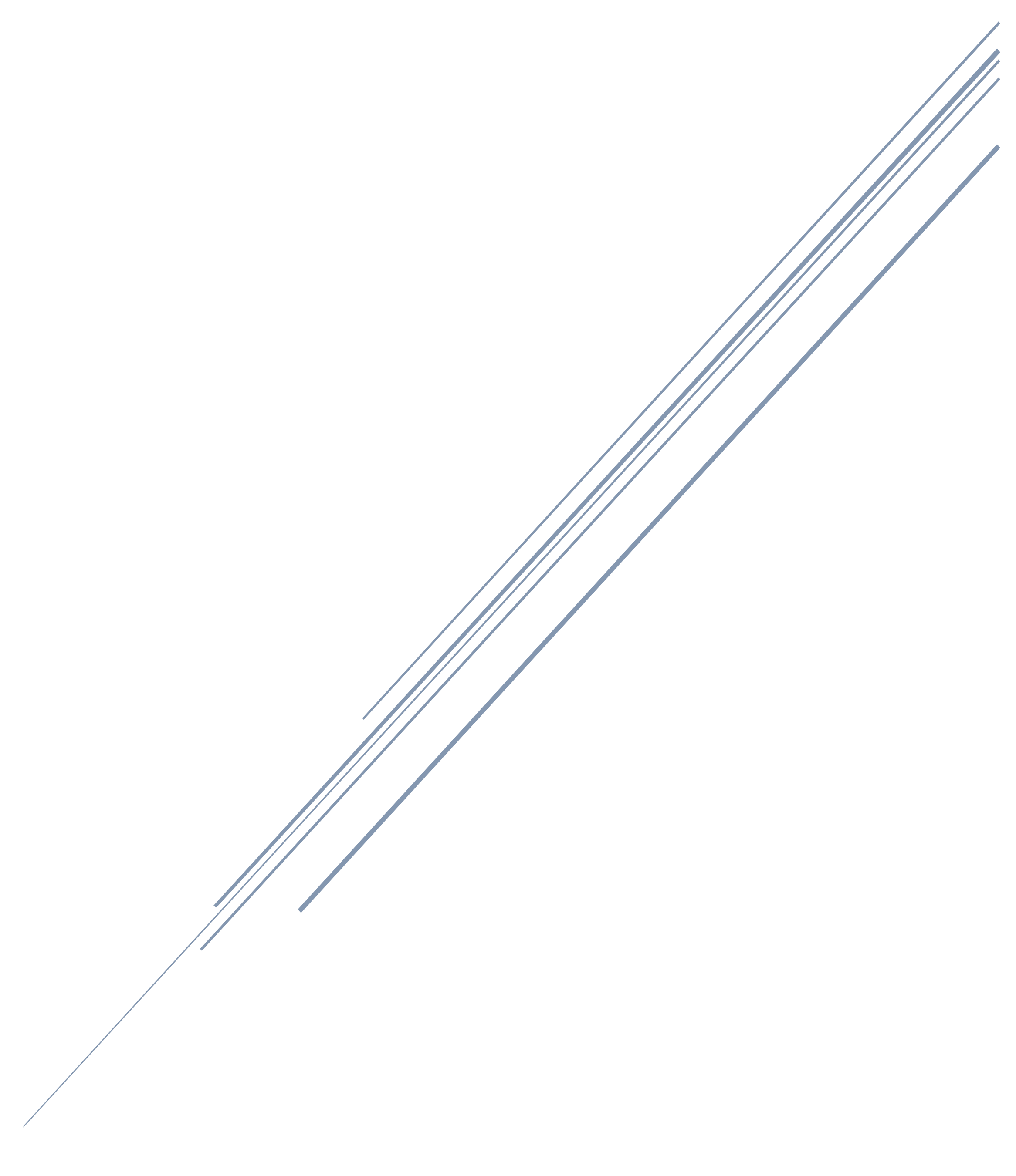
Node.jsで作る定期・APゲーム

Author: ('ω'   )<) @siroi_sakana



本書について

本書は「定期ゲ Advent Calendar 2019」の7日目の記事として書かれたものです。情報は2019年12月1日時点のものであり、この日付から大きく日数が経過している場合、情報が古くなっている可能性があります。

また、本書は「jQueryを使ったホームページを1から作成したことがある方」あるいはそれと同レベルの知識を持つ方をメインターゲットとして執筆されています。HTML、CSS、JavaScriptの基本的な書き方や構文、知識などについて本書では取り扱いません。これらの基礎をまだ習得できていない場合、入門書や入門サイトなどを読んでから本書を読み進めることをおすすめします。

なお、本書は無料で公開されていますがVPSの契約など開発環境を整えるにあたってお金がかかります。その点には留意しておいてください。本書に記載された内容によって何らかの不利益を被った場合でも筆者は一切の責任を負いません。また、筆者は専門家でもなんでもない素人です。間違いなどがあるかもしれませんのでその点はご了承ください。

本書で記述されているソースコードのライセンスについてはCC0とします。利用、改変、再公開などにあたって特別許諾や著作者表示などは必要ありません。ソースコード以外の部分については適用されないため注意してください。

目次

Chapter 1 11
1.1 準備が必要なもの 12
1.2 Webアプリケーション 13
1.2.1 定期・APゲームとWebアプリ 13
1.2.2 近代的なWebアプリ開発 13
1.3 ライブラリとライセンス 14
1.3.1 ライブラリとは 14
1.3.2 ライセンスとは 14
1.3.3 MITライセンス 14
1.3.4 BSDライセンス 14
1.3.5 Apache License 2.0 15
1.3.6 GPLライセンス 15
1.3.7 LGPLライセンス 16
1.3.8 その他のライセンス 16
1.3.9 ライブラリの例 16
Chapter 2 18
2.1 Windowsの環境構築 19
2.1.1 Google Chrome 19
2.1.2 Google Chromeの拡張機能のインストール 19
2.1.3 Tera Term 20
2.1.4 Visual Studio Code 20
2.1.5 Visual Studio Codeの拡張機能のインストール 21
2.1.6 MongoDB Compass 21
2.2 Linux/CentOS7の基礎 23
2.2.1 Linuxとは 23
2.2.2 CentOS7とは 23
2.2.3 ユーザーの種類とグループ 23
2.2.4 パーミッション 23
2.2.5 ポート 24
2.2.6 ファイアウォール 26
2.2.7 CLI 27
2.3 Linux/CentOS7のコマンド 28
2.3.1 pwd カレントディレクトリの表示 28
2.3.2 cd ディレクトリ移動 28
2.3.3 mkdir ディレクトリ作成 28
2.3.4 ls ファイル一覧 29

2.3.5 su ユーザー切り替え 30
2.3.6 vi テキストエディタ 30
2.3.7 rm ファイル削除 31
2.3.8 chown 所有者・所有グループの変更 31
2.3.9 chmod パーミッションの変更 31
2.3.10 yum ソフトウェアの管理 32
2.3.11 systemctl サービスの管理 32
2.4 VPSの契約と初期設定 33
2.4.1 VPSの選定 33
2.4.2 VPSを借りる(インスタンスを作る) 33
2.4.3 VPSに接続する 34
2.4.4 全体アップデート 37
2.4.5 Redisのインストール 37
2.4.6 Node.jsのインストール 38
2.4.7 Nginxのインストール 38
2.4.8 MongoDBのインストール 39
2.4.9 再起動 39
2.5 基礎的なセキュリティ設定 41
2.5.1 一般ユーザーの作成 41
2.5.2 rootログインの禁止 44
2.5.3 SSHポートの変更 45
2.5.4 補足:途中で接続が切れてしまった場合 46
2.5.5 MongoDBのセキュリティ設定 46
2.6 Webサーバー設定 50
2.6.1 Nginxとは 50
2.6.2 httpポートの開放 50
2.6.3 ドメインの基礎知識 50
2.6.4 ドメインの取得(無料編) 51
2.6.5 ドメインの取得(有料編) 52
2.6.6 ドメインの設定 53
2.6.7 https化(SSL/TLSの設定) 55
2.6.8 httpでのnginxの設定 58
2.7 手順書 60
Chapter 3 64
3.1 HTML5 65
3.1.1 HTML5とは 65
3.1.2 HTML4からの変更点 65
3.1.3 セマンティックなタグ 66

3.1.4 メディア関連のタグ 67
3.1.5 新しく追加されたフォーム 68
3.1.6 HTML5.1以降で追加されたタグと属性 69
3.1.7 viewport 69
3.2 ES2015(ES6) 70
3.2.1 ES2015(ES6)とは 70
3.2.2 const/let 70
3.2.3 テンプレートリテラル 71
3.2.4 デフォルト引数 71
3.2.5 クラス 71
3.2.6 アロー関数 76
3.2.7 Array 77
3.2.8 分割代入 80
3.2.9 Promise 82
3.2.10 async/await 86
3.3 CSS3とCSS関連技術 87
3.3.1 CSS3とは 87
3.3.2 flexbox 87
3.3.3 calc() 87
3.3.4 その他追加されたプロパティ・値 88
3.3.5 疑似クラス 91
3.3.6 transition 92
3.3.7 疑似要素 93
3.3.8 メディアクエリ 93
3.3.9 CSSフレームワーク 94
3.3.10 Sass(SCSS) 95
3.4 Node.js 99
3.4.1 Node.jsとは 99
3.4.2 シングルスレッド 99
3.4.3 ノンブロッキングIO 99
3.4.4 環境を整える 99
3.4.5 Hello, World! 101
3.4.6 ライブラリのインストールと使用法 102
3.4.7 ファイルの分割 104
3.5 フロントエンドの基礎 106
3.5.1 フロントエンドとは 106
3.5.2 3大フレームワーク 106
3.5.3 Nuxt.jsとは 106

3.5.4 拡張機能のインストール ... 108
3.5.5 リバースプロキシとその設定 ... 108
3.5.6 Nuxt.jsの初期設定 ... 109
3.5.7 Vue.jsの基礎 ... 112
3.5.8 v-bind ... 114
3.5.9 v-if / v-else-if / v-else ... 114
3.5.10 v-on ... 116
3.5.11 v-for ... 117
3.5.12 フィルター ... 118
3.5.13 双方向算出プロパティ ... 119
3.5.14 DOMの参照 ... 120
3.5.15 新規ページの作成とルーティング ... 121
3.5.16 コンポーネントの作成と読み込み ... 128
3.5.17 親から子への値の受け渡し ... 130
3.5.18 カスタムイベント(子から親への受け渡し) ... 134
3.5.19 v-modelを使った子から親への値の受け渡し ... 136
3.5.20 Vue.jsのライフサイクル ... 139
3.5.21 SCSSの利用 / CSSのスコープ ... 139
3.5.22 assets ... 141
3.5.23 layouts ... 142
3.5.24 ページタイトルの設定 ... 143
3.5.25 Vuex ... 144
3.5.26 middleware ... 147
3.5.27 plugins ... 147
3.5.28 Vue.js devtoolsの使い方 ... 148
3.5.29 公式ガイド ... 149
3.6 バックエンドの基礎 ... 150
3.6.1 バックエンドとは ... 150
3.6.2 リバースプロキシの追加 ... 150
3.6.3 Expressの基礎 ... 150
3.6.4 ルーティングとレスポンス ... 152
3.6.5 ミドルウェア関数 ... 154
3.6.6 エラーコード ... 155
3.7 データベースの基礎 ... 157
3.7.1 データベースとは ... 157
3.7.2 SQLとNoSQL ... 157
3.7.3 MongoDBとは ... 157
3.7.4 データベースの作成 ... 157

3.7.5 スキーマ 158
3.7.6 mongooseを使ったMongoDBへの接続 160
3.7.7 データ保存 160
3.7.8 データ検索 163
3.7.9 プロジェクション 165
3.7.10 複雑な検索 166
3.7.11 データ更新 167
3.7.12 データ削除 169
3.7.13 ObjectIdとpopulate 169
3.7.14 取得したドキュメントの編集 171
3.7.15 MongoDB Compassの使い方 172
Chapter 4 176
4.1 仕様策定 / プロジェクトの作成 177
4.1.1 仕様の決定 177
4.1.2 プロジェクトフォルダーの作成 177
4.1.3 フロントエンドの初期設定 178
4.1.4 バックエンドの初期設定 181
4.1.5 データベースの作成 183
4.1.6 Nginxの設定変更 184
4.1.7 プログラムの実行 184
4.1.8 補足：ENOSPCエラーが出る場合 185
4.1.9 補足：EADDRINUSEエラーが出る場合 186
4.2 トップページ / メニュー 187
4.2.1 トップページとレイアウトの変更 187
4.2.2 メニューの作成と読み込み 190
4.3 キャラクター登録 194
4.3.1 キャラクタースキーマの作成 194
4.3.2 キャラクター登録APIの作成 197
4.3.3 ページの作成 198
4.3.4 ENoの設定 206
4.3.5 パスワードのハッシュ化 208
4.3.6 クライアントの改良 210
4.4 ログインの仕組みと実装 213
4.4.1 ログインの仕組み 213
4.4.2 ログインAPIの実装 213
4.4.3 クライアント側のログイン処理の実装 219
4.4.4 自動ログイン 222
4.4.5 ログインが必要なページ 223

4.4.6 ログアウト 224
4.4.7 非ログイン状態のメニュー 225
4.4.8 ログイン / 登録時のリダイレクト 226
4.5 ホーム 229
4.5.1 ホームAPI 229
4.5.2 APIの結果を受け取ってページを表示 230
4.6 キャラクターリスト 240
4.6.1 キャラクターリストAPIの作成 240
4.6.2 キャラクターリストの作成 242
4.6.3 キャラクターページAPIの作成 246
4.6.4 キャラクターページの作成 248
4.7 キャラクター設定 251
4.7.1 装飾システムの作成 251
4.7.2 キャラクター設定APIの作成 252
4.7.3 キャラクター設定ページの作成 253
4.7.4 XSS 261
4.7.5 CSRF 264
4.8 お気に入り / ブロック / ミュート 267
4.8.1 APIの作成 267
4.8.2 キャラクターページの変更 271
4.9 トークルーム 281
4.9.1 トークルームのスキーマ設計 281
4.9.2 初期化処理 285
4.9.3 ENo設定処理の変更 288
4.9.4 トークルーム作成APIの作成 289
4.9.5 トークルーム作成ページの作成 290
4.9.6 トークルームAPIの作成 298
4.9.7 ダイス機能 302
4.9.8 発言APIの作成 303
4.9.9 発言削除APIの作成 305
4.9.10 トークルームページの作成 306
4.9.11 トークルームリストAPIの作成 317
4.9.12 トークルームリストページの作成 319
4.9.13 トークルーム編集APIの作成 323
4.9.14 トークルーム削除APIの作成 324
4.9.15 トークルーム編集ページ 325
4.10 設定ページ 330
4.10.1 キャラクター削除APIの作成 330

4.10.2　設定ページの作成 330
4.11　戦闘・基礎編 335
4.11.1　戦闘の仕様の決定 335
4.11.2　戦闘アルゴリズム・基礎編 335
4.11.3　戦闘プログラムの作成 337
4.11.4　ユニットクラスの作成 339
4.11.5　バトルクラスの作成 340
4.11.6　戦闘の実行 344
4.12　戦闘・応用編 346
4.12.1　ATK・AGIの組み込み 346
4.12.2　ヘイトシステム 349
4.12.3　その他のステータスの組み込み 352
4.12.4　アクティブスキルの構造 358
4.12.5　スキルエフェクト 359
4.12.6　アクティブスキル 363
4.12.7　スキルの追加 371
4.12.8　状態異常 375
4.12.9　パッシブスキル 379
4.12.10　ライブラリ化 384
4.12.11　battle.jsの最終形 391
4.12.12　skill.jsの最終形 401
4.13　戦闘設定 417
4.13.1　戦闘設定APIの作成 417
4.13.2　戦闘設定ページの作成 419
4.13.3　プロフィールの改変 427
4.14　探索戦（AP） 433
4.14.1　EJS 433
4.14.2　敵キャラクターの作成 438
4.14.3　ステージの作成 440
4.14.4　探索戦ログの生成 443
4.14.5　探索戦ログの保存 445
4.14.6　探索戦APIの作成 446
4.14.7　Nginxの設定変更 452
4.14.8　APの配布 452
4.14.9　探索戦ページの作成 454
4.14.10　ログ検索API 462
4.14.11　ログ検索ページ 464
4.14.12　初期化処理の追加 469

4.15 物語戦(定期更新) ... 470
4.15.1 テンプレートとステージの作成 ... 470
4.15.2 行動宣言APIの作成 ... 477
4.15.3 行動宣言ページの作成 ... 481
4.15.4 定期更新の仕様 ... 485
4.15.5 宣言内容の保持 ... 485
4.15.6 更新 ... 488
4.15.7 再更新 ... 493
4.15.8 結果の確定と報酬の付与 ... 493
4.15.9 各処理の呼び出し ... 495
4.15.10 Nginxの設定変更 ... 496
4.15.11 キャラクターページからの更新結果参照 ... 497
4.15.12 初期化設定の変更 ... 509
4.16 管理者用ページ ... 510
4.16.1 問い合わせページ ... 510
4.16.2 管理者しかアクセスできないAPI ... 514
4.16.3 問い合わせの表示 ... 515
4.16.4 お知らせAPI ... 521
4.16.5 お知らせページ ... 523
4.17 公開前にするべきこと ... 532
4.17.1 ルールブックの作成 ... 532
4.17.2 アイコンの作成 ... 532
4.17.3 ビルドとプロダクションモード ... 533
4.17.4 公開用サーバーのレンタルとセットアップ ... 533
4.17.5 PM2 ... 536
4.17.6 バックアップの定期実行 ... 542
4.17.7 その他設定 ... 543
4.17.8 開発用サーバーのアクセス制限 ... 544
4.17.9 更新の反映手順 ... 547
Chapter 5 ... 548
5.1 あとがき ... 549
5.2 おまけ ... 550
5.2.1 PageSpeed Insights ... 550
5.2.2 Can I use... ... 550
5.2.3 metaタグ ... 550
5.2.4 フォント ... 552
5.2.5 Webフォント ... 555
5.2.6 色 ... 556

5.2.7 配色 563
5.2.8 カラーユニバーサルデザイン 566
5.2.9 論理ピクセル / 物理ピクセル 569
5.2.10 変数の命名規則 569
5.2.11 CSSの設計手法 570
5.2.12 TypeScript（AltJS） 573
5.2.13 Webhook 574
5.2.14 正規表現 574
5.2.15 SELinux 576

Chapter 1
はじめに

はじめるにあたって必要なもの、
Webアプリ開発を取り巻く環境や、
その周辺知識について取り扱います。

1.1 準備が必要なもの

本書の内容を進めるにあたって必要なものは以下のとおりです。

パソコン(Windows 10 64bit版)

Macなどでも製作可能かと思われますが本書ではWindows 10 64bit版で解説を進めます。

インターネット環境

VPSの契約/クレジットカード等支払い手段

定期・APゲームの開発・公開のためのサーバーを立てるため**VPS**(Virtual Private Server)を契約する必要があります。契約の手順は後ほど解説します。本書では**ConoHa VPS**を利用して解説を進めます。KVM環境でCentOS7が用意されているVPSであればConoHa VPS以外でも構わないかと思われます。

なお、KAGOYA CLOUD／2 OpenVZなど**OpenVZ環境ではこの後の手順で失敗することを確認しています。**よく分かっていない場合、OpenVZ環境での開発・公開はおすすめできません。

ConoHa VPSを利用する場合、利用できる支払い手段は以下の通りです(2019年12月1日現在)。

- クレジットカード
- ConoHaチャージ(先払い)
    - コンビニ支払い
    - 銀行振込
    - PayPal etc.
- ConoHaカード(プリペイドカード)

電話番号(有料ドメインを取得する場合)

有料のドメインを取得する場合、電話/SMSによる自動認証が必要になります。(ムームードメインの場合)

Google検索に頼る力

本書はほぼこれ単体で最後まで読み進められるように構成されていますが、本書の内容から進んで独自の要素などを作る・改変するとなった場合、必ずどこかで詰まることがあると思われます。そうなった場合に自己解決する力、すなわち検索する力は大切になってきます。

特に周囲に聞くことが難しい個人開発ではこれができるかどうかが一番大切です。そうでなくとも、記憶が曖昧なままにプログラミングをするとバグの原因となる場合があります。少しでもわからないことがあったら検索するという習慣をつけましょう。

高校卒業程度の英語力もしくは自動翻訳の読解力

プログラミングにおいて、問題に突き当たって検索しても日本語情報が出てこないということはよくあります。特に公式ドキュメントなどは大抵の場合英語で書かれているので、自力で問題に対処するにあたって英語を読める、もしくは自動翻訳を使えるというのは大切なスキルです。

1.2 Webアプリケーション

1.2.1 定期・APゲームとWebアプリ

一般に定期・APゲームといった場合、ブラウザでゲームサイトにアクセスし、オリジナルのキャラクターを登録して交流したり、行動を宣言して戦闘などを行ったりするブラウザゲームのことを指します。（勿論ブラウザを使用しないものもありますが、ブラウザを使用するものが一般的でしょう。）

また、Webアプリケーションとは何かというとブラウザから利用可能なアプリケーションやサービスなどを指します。つまり、一般に定期・APゲームはWebアプリです。これはどういうことかというと、定期・APゲームはWebアプリのノウハウを用いて開発できるということです。

本書は近代的なWebアプリ開発技術を有効活用しながら定期・APゲームを開発することを目標とします。それにあたって、まずは近代的なWebアプリケーション開発を取り巻く環境について学びます。

1.2.2 近代的なWebアプリ開発

近年、スマートフォンの爆発的な普及などによりインターネットは一般人にもより身近なものになり、ウェブページの表示環境もパソコンのみならずタブレットやスマホなど多彩なものになっていきました。それだけではなく、ウェブページは単に情報を表示するだけではなく、アプリケーションとしても用いられるようになっていきます。

その流れの中で、旧来の技術だけでは様々なニーズに対応するのが難しくなっていきました。そして、jQuery（JavaScriptをより便利に扱えるようにしてくれる多機能ライブラリ）だけに頼り切るのではなく、目的にあった様々なライブラリを取捨選択して使っていくのが当たり前になっていきました。

1.3 ライブラリとライセンス

1.3.1　ライブラリとは

ライブラリとは、簡潔に言うと汎用性の高い便利機能をひとまとめにしたものです。jQueryもライブラリの一種です。先述の通り、近代的なWebアプリ開発において欠かせないものであり、本書も数々のライブラリを活用しながら開発を進めていきます。

1.3.2　ライセンスとは

ライセンスとは、ライブラリを利用するにあたって従わなければならない条件のことです。プログラムの世界では、ライブラリの開発者個々人が独自に利用許諾条件を書くのではなく、様々な組織によって作られた様々なライセンスの中から適用したいライセンスを選んで適用するのが一般的です（もちろん個々人によって定められる場合もあります）。続く内容でよく利用されているライセンスを紹介していきます。

1.3.3　MITライセンス

MITライセンスはマサチューセッツ工科大学によって作られたライセンスです。もっともよく利用されているライセンスであり、本書で取り扱うライブラリもほとんどがMITライセンスの下で頒布されているものです。

MITライセンスを利用した場合に従うべき条件は「ソースコードや別のファイルに著作権表示を行うこと」「このソフトウェアを利用したことで何か問題が起こった場合でもライブラリの作者は責任を負わないこと」の2つであり、この条件に従う限り誰でも自由にソフトウェアの再配布、変更、商用利用などができます。なお、これは意訳であり細かなニュアンスや内容などについてはかならず原文に目を通してください。MITライセンスの原文は以下のURLから確認できます。

The MIT License | Open Source Initiative
https://opensource.org/licenses/mit-license.php

Open Source Group JapanによるMITライセンスの日本語訳は以下のURLより参照できます。あくまで日本語訳は参考であるという点に留意してください。

licenses/MIT_license - Open Source Group Japan Wiki - Open Source Group Japan - OSDN
https://ja.osdn.net/projects/opensource/wiki/licenses%2FMIT_license

1.3.4　BSDライセンス

BSDライセンスはカリフォルニア大学によって作られたライセンスで、BSD-4-Clause、BSD-3-Clause、BSD-2-Clauseの3つのバージョンがあります。それぞれ4つ、3つ、2つの条文からなるライセンスで、それぞれ細かな違いがあります。なお、条文が少ないほど条件が緩く、新しいバージョンです。

それぞれ以下のような条件となっています。なお、これは意訳であり細かなニュアンスや内容などについてはかならず原文に目を通してください。

BSD-2-Clause

MITライセンスと似たようなライセンスです。「ソースコード形式で再配布する場合著作権表示や免責条項などを保持すること」「バイナリ形式で再配布する場合著作権表示や免責条項などを別ファイルなどによって表示すること」という2つの条文からなり、免責で「作者は一切責任を負わない」と指定されています。この条件と免責に従う限り誰でも自由にソフトウェアの再配布、変更、商用利用などができます。

BSD-3-Clause

BSD-2-Clauseの条件に加え、作者の名前を販売促進などに用いてはならない、という条件が加わっています。これはどういうことかというと、「あの○○さんの作ったライブラリを使った素晴らしい製品」というような宣伝をしてはならないということです。

BSD-4-Clause

BSD-3-Clauseの条件に加え、初期貢献者への指定の謝辞を表示しなければならない、という条件が加わっています（宣伝条項）。この性質により、BSD-4-Clauseは後述するGPLライセンスのライブラリとは共存できません。

1.3.5 Apache License 2.0

Apache License 2.0のライセンスが適用されているものを利用するだけであれば、「Apache License 2.0の下で頒布されている成果物が含まれています。」というような表示を行うことで利用することができます。

また、Apache License 2.0は商標や特許に関しての条項などが用意されており法律を意識した内容になっている他、ライセンスを一度適用した場合取り消しができなくなるなど、作者・利用者双方に様々な配慮がなされたライセンスと言えます。

Apache License 2.0はMITライセンスやBSDライセンスに比べると条文が長く、ここでは紹介しきれないため条項については各自で読んでください。以下が原文になります。

Apache License, Version 2.0
http://www.apache.org/licenses/LICENSE-2.0

Open Source Group JapanによるApache License 2.0の日本語訳は以下のURLより参照できます。あくまで日本語訳は参考であるという点に留意してください。

licenses/Apache_License_2.0 - Open Source Group Japan Wiki - Open Source Group Japan - OSDN
https://ja.osdn.net/projects/opensource/wiki/licenses%2FApache_License_2.0

1.3.6 GPLライセンス

GPLはオープンソースコードライセンスです。GPLライセンスの最大の特徴として、このライセンスが適用されているライブラリを同梱した場合、あなたの書いたソースコードも全てGPLとしてオープンソースで公開する必要があ

ります。そうして公開されたソースコードがまた別の人によって同梱され利用された場合、そのソースコードもGPLとして公開しなければなりません。

　このような派生物にも同一のライセンスを適用しなければならないといった考えのことを**コピーレフト**と呼びます。これに対し、MITライセンスやBSDライセンスのような派生物に同一のライセンスの使用を強制しないという考えのことを**非コピーレフト**といいます。

　ソースコードを全て開示しなければならないのは定期・APゲームとして致命的と思われるので、このライセンスが適用されたライブラリを使用するのは定期・APゲームにおいてはあまりおすすめできません。また、BSD-4-Clauseのところでも軽く触れましたがコピーレフトなどの定めによりGPLとは共存できないライセンスも存在します。GPLを利用する場合、そのような事情も考慮に入れる必要があります。

1.3.7　LGPLライセンス

　LGPLライセンスは、ざっくりと言ってしまうとGPLライセンスの影響範囲を縮小させたものです。LGPLのライブラリを同梱したからといってあなたの書いたソースコードをLGPLとして開示する必要はありません。ただし、LPGLのライブラリそのものを改変した場合、改変した後のライブラリについてはLGPLに基づいてオープンソースで公開する必要があります。このようなもののことを**準コピーレフト**と呼びます。

　また、それに加えてリバースエンジニアリングを許可する必要があります。リバースエンジニアリングとはソフトウェアに対して解析等を行うことを指します。定期・APゲームではこの部分がネックになるかどうかは物によるといった所でしょうか。よく分かっていない場合、このライセンスも避けたほうが無難でしょう。もとより、**よく分かっていないライセンスは使用するべきではありません。**

1.3.8　その他のライセンス

　その他、時々見るライセンスを簡単に紹介していきます。

Unlicensed/CC0

　UnlicensedもCC0も、著作権をほぼ放棄しパブリックドメインとほぼ同一の条件で配布しているライセンスです。

ISCライセンス

　MITと同じようなライセンスです。

SIL Open Font License

　これはソースコードではなくフォントに適用されるライセンスです。単体での再配布はできませんが、ソフトウェアへの同梱であれば可能です。フォント自体を改変した場合は改変した旨の明記が必要で、かつ、元フォント名を含めないファイル名でSIL Open Font Licenseで公開しなければなりません。

1.3.9　ライブラリの例

　ここでは今回使用するライブラリを軽く紹介していきます。

Vue.js

UIを構築するためのライブラリです。MITライセンスで公開されています。

Nuxt.js

Vue.jsをサーバーサイドレンダリングするためのライブラリです。MITライセンスで公開されています。

Express

Node.jsでブラウザからアクセスを受け取るためのライブラリです。MITライセンスで公開されています。

mongoose

Node.jsからMongoDBというデータベースを扱うためのライブラリです。MITライセンスで公開されています。

Passport.js

ログインなどの仕組みを扱うためのライブラリです。MITライセンスで公開されています。

Chapter 2

環境構築編

必要なソフトウェア等のインストールを行い、
開発のために必要な環境を構築していきます。
また、VPSの契約などの他、
環境構築に必要な基礎知識の学習も行います。

2.1 Windowsの環境構築

まずはWindowsに必要なソフトウェアをダウンロード＆インストールします。

2.1.1 Google Chrome

Google ChromeとはGoogleが開発しているブラウザです。2019年現在、ブラウザシェア率ナンバー1なのでこれを読んでいる方もChromeを利用されている方が多いのではないでしょうか。数多くの先進的な機能が実装されているブラウザで、今もなお新しい機能が追加され続けています。

同じように先進的機能が実装されているブラウザにはFirefoxがありますが、Chromeの方が開発ツールの機能が豊富なため開発にはChromeを使います。解説もChromeでの開発を前提として行っていきます。

ChromeはGoogleの公式ページからダウンロードします。

Google Chrome ウェブブラウザ
https://www.google.com/intl/ja_jp/chrome/

「Chromeをダウンロード」をクリックすると利用規約が書かれたウィンドウが出るので同意できる場合は「同意してインストール」をクリックしましょう。インストーラーがダウンロードされるので実行してください。インストールは自動で行われます。Chromeのウィンドウが表示されればインストール完了です。

2.1.2 Google Chromeの拡張機能のインストール

次に、開発する際に使用する拡張機能をChromeにインストールしていきます。この先のURLはChromeで開いてください。開いたページでそれぞれ「Chromeに追加」ボタンを押すとインストールできます。

HTTP/2 and SPDY indicator
https://chrome.google.com/webstore/detail/http2-and-spdy-indicator/mpbpobfflnpcgagjijhmgnchggcjblin

HTTP/2やSPDYが使われているかどうかひと目で分かるようになるツールです。URLバーの隣に雷マークが表示されていればインストール成功です。アイコンの意味はそれぞれ以下のとおりです。

- HTTP/2、SPDYが使われていない通信。
- HTTP/2が使われている通信。
- SPDYが使われている通信。

Vue.js devtools
https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd

Vue.jsが使われているかどうかの確認などができます。URLバーの隣に灰色もしくは緑色のVのようなアイコンが表示されていればインストール成功です。Vue.jsの公式ガイドなど、Vue.jsを使用しているページを開くとアイコンが緑色になります。

はじめに —— Vue.js
https://jp.vuejs.org/v2/guide/

Vue.jsが使われていないページ。
Vue.jsが使われているページ。

開発ツールからVue.jsの詳細な状態確認をできるようにする機能もあり、こちらがメインなのですがこの機能については実際に使用する際に解説します。

2.1.3 Tera Term

Tera Termは様々な機能を備えたフリーのターミナルエミュレーターです。物理学者の寺西 高さんが開発していたものをTeraTerm Projectが開発を引き継いだものになります。ここで扱うレベルにおいてはサーバーを遠隔操作するためのソフトウェアにいろいろ便利な機能がついたものと思っておけば問題ありません。

ダウンロードはOSDNというサイトから行います。以下のURLを開き、taraterm-○○.exeと書かれたリンクをクリックします。○○にはバージョン番号が入ります(2019/11/1現在、最新のバージョンは4.104)。

ダウンロードファイル一覧 - Tera Term – OSDN
https://ja.osdn.net/projects/ttssh2/releases/

ページが切り替わり、自動でインストーラーのダウンロードが始まるのでダウンロードが終わったらインストーラーを実行しインストールを進めてください。コンポーネントの選択、追加タスクの選択は初期設定のままで特に問題ありません。

2.1.4 Visual Studio Code

Visual Studio CodeはMicrosoftが開発しているソースコードエディタで、JavaScriptなどの動的型付け言語に強みを持ちます。スッキリした見た目で、拡張機能も充実しています。Microsoftが開発しているソースコードエディタには他にもVisual Studioというものがあるのですが、そちらとはまた別物です。本書では使いやすさを重視してVisual Studio Codeを使用します。

ではVisual Studio Codeをダウンロードしていきましょう。下記リンクにアクセスし、Visual Studio Code公式ページを開きます。サイトは英語ですがソフトウェアは日本語化できるのでご安心ください。

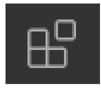
Visual Studio Code - Code Editing. Redefined
https://code.visualstudio.com/

ページを開いたらまずは「Download for Windows Stable Build」と書かれたところをクリックします。ページが切り替わり、自動でインストーラーのダウンロードが始まります。自動でダウンロードが始まらない場合、direct download link.と書かれたところをクリックしてください。

ダウンロードが終わったらインストーラーを実行しインストールを進めてください。設定は初期設定のままで特に問題ありません。インストールが終わったら一度PCを再起動してください。

2.1.5 Visual Studio Codeの拡張機能のインストール

PCの再起動が終わったらインストールしたVisual Studio Codeを起動します。画面左にある下記のアイコンをクリックし、EXTENSIONSというメニューを表示します。



EXTENSIONSのアイコン

開いたメニューの中に「Search Extensions in Marketplace」という検索バーがあるので、ここから拡張機能を検索してインストールしていきます。ここでは拡張機能を4つインストールします。それぞれタイトルを検索バーに入力して検索しインストールしてください。タイトルは『』で囲まれている部分です。

『Japanese Language Pack for Visual Studio Code』(Microsoft)
Microsoft公式による日本語化のための拡張機能です。インストール後Visual Studio Codeを閉じて再起動すると日本語化されます。

『Remote - SSH』(Microsoft)
Microsoft公式によるSSH接続経由で遠隔開発をするための拡張機能です。

『Vetur』(Pine Wu)
Vue.jsのソースコードを扱いやすくしてくれる拡張機能です。

『EJS language support』(DigitalDrainstem)
EJSというテンプレートエンジンのテンプレートを扱いやすくしてくれる拡張機能です。

2.1.6 MongoDB Compass

データベースの一種であるMongoDBを管理、操作するためのツールです。MongoDBの開発を行っているMongoDB Inc.により提供されています。MongoDBのデータを閲覧したり、編集したり、遠隔操作したりなどができます。MongoDB Compassは以下のURLよりダウンロードします。

Compass | MongoDB
https://www.mongodb.com/products/compass

　ページを開いたら中央の「Try it now」と書かれた緑色のボタンをクリックします。ダウンロード内容を尋ねる画面になるので、Versionは「(stable)」と末尾に書かれているもの、Platformsは「Windows 64-bit (7+) (MSI)」を選択してその下の「Download」というボタンをクリックしましょう。インストーラーがダウンロードされるので実行してインストールを行います。設定はデフォルトのままで特に問題ありません。

2.2 Linux/CentOS7の基礎

2.2.1 Linuxとは

VPSの契約とその環境構築に移る前に、まずLinuxの基礎の基礎的な知識を学習します。**Linux**はOSの一種です。サーバー用途等に広く使われており、今回定期・APゲームを設置する環境もLinuxの上に作られます。

少し話が脇道にそれますが、OSというものは単体では使い物にならない物です。Windowsを例に上げてみましょう。フォルダーやファイルを開けるのはエクスプローラーというソフトウェアのおかげです。ウェブページが見られるのはブラウザのおかげですし、曲を聞いたり動画を再生したり絵を描いたり、そのほか様々な機能はソフトウェアあってのものです。それらがなければWindowsは全く使い物にならないことでしょう。

Linuxも同様にソフトウェアがあって初めて使えるようになります。利用しやすいよう、Linuxとソフトウェア等を同梱しまとめ上げたものを**Linuxディストリビューション**と呼びます。Linuxディストリビューションのことを指してLinuxと呼ぶ場合も多々あります(**広義のLinux**)。

Linuxディストリビューションには様々な種類がありますが、主なものには以下のようなものがあります。

Debian系	Debian
	Ubuntu
Red Hat系	Red Hat Enterprise Linux
	CentOS

2.2.2 CentOS7とは

CentOS7はLinuxディストリビューションの1つです。今回作る定期・APゲームはCentOS7の上で開発を進めていきます。安定性が高くサポート期間が非常に長いのが特徴で、その特性からサーバー用途として人気があります。なおサポート終了日は2024年6月30日が予定されています。**Red Hat Enterprise Linux**(RHEL)という業務用Linuxディストリビューションから商用ソフトなどを取り除いたもので、Red Hat系に分類されます。

なお2019月11月1日現在、より新しいバージョンであるCentOS8が出ていますがまだ出たばかりのバージョンであり資料も少ないのでここでは取り扱いません。

2.2.3 ユーザーの種類とグループ

Linuxには主にrootユーザーと一般ユーザーの2種類があります。Windowsでいうところの管理者とそうでないユーザーと同じようなものです。ソフトウェアのインストールや管理などはrootユーザーで行い、開発や定期・APゲームの実行には一般ユーザーを用います。

また、Linuxにはグループという機能があり複数人でLinuxの管理・利用を行う際に役立ちます。ただ、今回はおそらく個人開発と思われるのでグループについては取り扱いません。そういうものがあるのだな、という程度に思っておいてください。

2.2.4 パーミッション

Linuxにはパーミッションという概念があります。Linuxではユーザーを「所有者」「所有グループ」「それ以外」に

分け、ファイルごとに読み込み権限、書き込み権限、実行権限をそれぞれ付与します。

パーミッションの表現方法には主に2種類の方法があります。1つ目は文字で表現する方式で、「rwx---r-x」というように表現されます。3文字ずつがセットになっていて、読み込み可能であれば1文字目は r 、書き込み可能であれば2文字目は w 、実行可能であれば3文字目は x になります。そうでない場合は - になります。r-xとなっている場合は読み込み、実行が可能で書き込みは不可ということです。これが3つセットになっており、1〜3文字目は所有者に対してのパーミッション、4〜6文字目は所有グループに対してのパーミッション、7〜9文字目はそのどちらでもないユーザーに対してのパーミッションになります。

実際に例を上げてみましょう。「rwxr-xr--」という表示になっていたとします。この場合「所有者は読み込み・書き込み・実行が可能、所有グループは読み込み・実行が可能、そのどちらでもないユーザーは読み込みのみ可能」という意味になります。

2つ目は数字で表現する方式です。理解しやすさの問題で後での紹介になりましたが、こちらの方が一般的に使われています。「705」「704」「654」などのように3文字の数字で表現されます。これは文字表現を短く表現したものになっています。

「rwxr-xr--」を例に上げてみましょう。これは「111101100」という2進数として見ることができます。これを8進数での表現に変更すると「754」になります。これが数字表現になります。つまり各桁は4、2、1の足し算の結果になっていて、+4されていれば読み込み権限、+2されていれば書き込み権限、+1されていれば実行権限です。

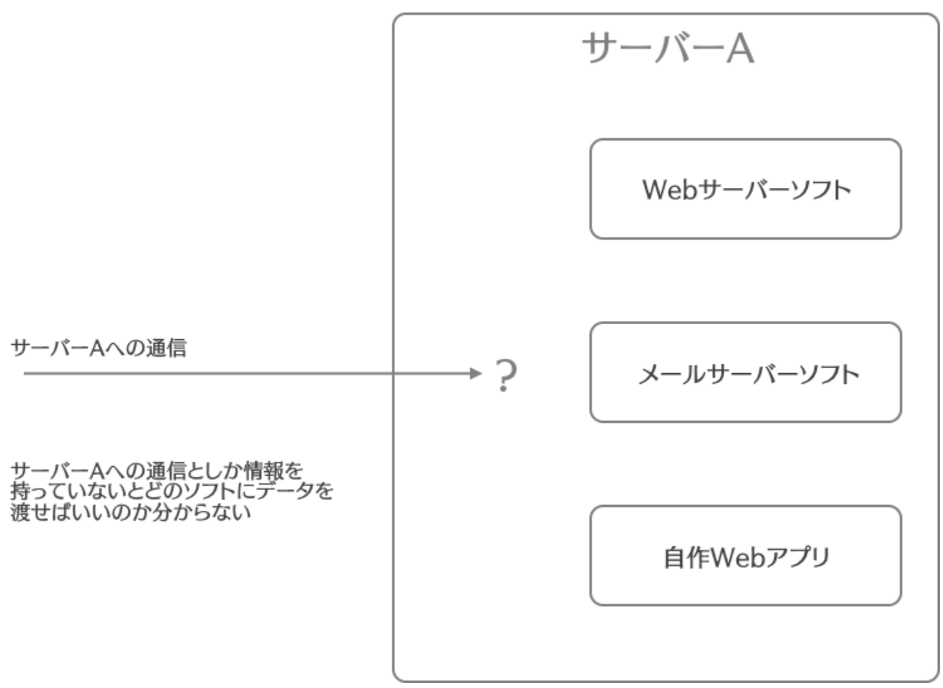
理屈としてはこうなりますが、分かりづらい場合見方さえ覚えてしまえば数字ごとに意味を覚えたり都度調べたりしても構いません。見方としては1文字目が所有者のパーミッション、2文字目が所有グループのパーミッション、3文字目がそのどちらでもないユーザーのパーミッションで、数字ごとに許可されている権限は以下の表のようになります。

0	何も権限が付与されていない状態
1	実行
2	書き込み
3	書き込み、実行
4	読み込み
5	読み込み、実行
6	読み込み、書き込み
7	読み込み、書き込み、実行

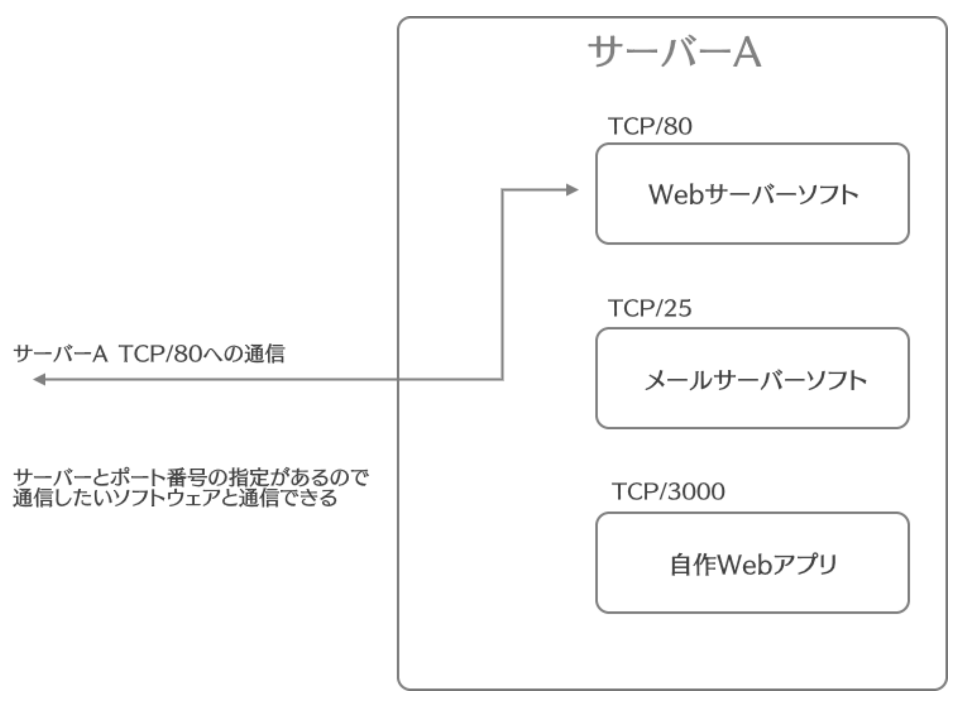
「705」を例に取ると、「所有者は読み込み・書き込み・実行が可能、所有グループにはなんの権限もなく、そのどちらでもないユーザーは読み込み・実行が可能」という意味になります。

2.2.5 ポート

サーバーでは多数のソフトウェアが同時に動作しており、ソフトウェアによってはサーバーの外部と通信を行っています。そのためソフトウェアがサーバーの外部と通信するとき、このサーバーのどのソフトウェアにデータを送るのか?という識別のための情報がないとデータを届けたいソフトウェアに届けることができません。



そこで利用するのが**ポート**という仕組みです。現実でもマンションやアパートなどで部屋ごとに部屋番号を振り分けそこに手紙や荷物を送る際はマンションやアパートの住所＋部屋番号を宛先とするように、ネットワーク通信の世界でも通信を行うソフトウェアに番号を振り分けデータを送る際はどのサーバーのどのポートに通信をするかを指定します。この番号のことを**ポート番号**といいます。ポート番号は0から65535まであり、その通信の方法によって**TCP**と**UDP**に別れます。ポート番号があることによりサーバーのどのソフトウェアにデータを送るかを指定できるようになり、通信したいソフトウェアと通信できるようになります。



実際の通信では通信内容を小分けに分割して通信を行っています。この通信の単位のことを**パケット**と呼びます。現実で多くの荷物を送る際はいくらかのダンボールに分けて送りますがそれと同じことです。ちなみにパケットは英語で小包を意味する単語です。

TCPとUDPではパケットの送り方が違っていて、TCPはパケットがしっかりと送られているか検証しながら通信を行っていく通信方法、UCPはそういった処理を行わずに通信を行う通信方法です。TCPは信頼性が高く、UDPは信頼性が低くなる代わりに高速です。基本的にはTCPを使います。

さて、ポート番号を使うときはソフトウェアごとにどのポート番号を使うかを取り決めて使用します。ポート番号はその範囲ごとに大まかに使用方法の指針が取り決められており、特に0〜1023番は**ウェルノウンポート番号**と呼ばれ広く使われる通信データを扱います。Linuxではウェルノウンポート番号を使用するにはroot権限が必要になります。一般的にウェルノウンポート番号は以下のように使用されます（主要なものの例）。

TCP/22	SSH	セキュアな遠隔操作を行う
TCP/80	HTTP	HTTP通信を行う
TCP/443	HTTPS	HTTPS通信を行う

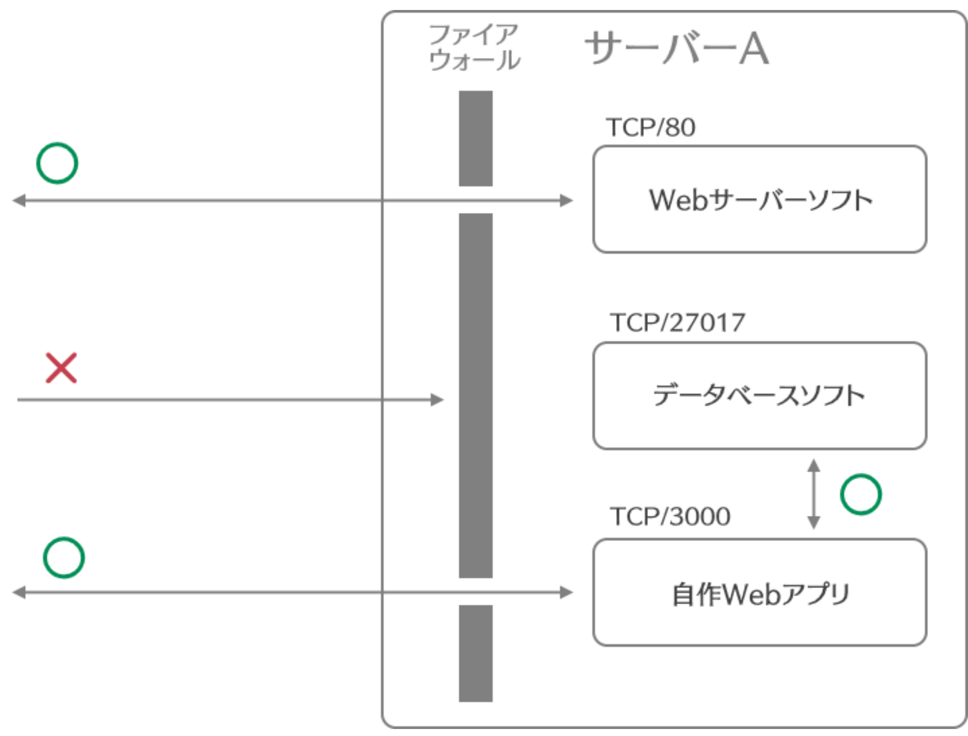
Webサーバー用ソフトウェアなど特別なものを除き基本的にはウェルノウンポート番号以外のポート番号を使用することになります。

2.2.6　ファイアウォール

外部から通信が行えるということはつまり外部から不正アクセスを試みることができるということでもあります。なので通常通信の内容をフィルタリングし不正なデータは弾きます。このような仕組みのことを**ファイアウォール**といいます。

ファイアウォールは基本的には全ての通信をブロックするようになっていて、ここから通信を許可したいポートだけを許可設定していきます。これを一般に**ポート開放**と呼びます。（なおVPSなどでは遠隔操作の通信ができないと操作がそもそもできないので、そのためのポートだけは最初から開放されいるのが一般的です。）

これにより外部からアクセスさせたいソフトウェアだけに絞って通信を許可させることができます。またサーバー内での通信に対してはファイアウォールでのフィルタリングは行われないので、サーバー内の他ソフトウェアとだけ通信を行いたい場合はポート開放を行わないことでそれが実現できます。



図の例ではTCP/80とTCP/3000だけポート開放を行っています。これにより「Webサーバーソフト」と「自作Webアプリ」は外部と通信ができるようになります。「データベースソフト」についてはポート開放を行っていないので外部から通信を行うことはできませんが、自作Webアプリなどからは通信を行って情報を受け取ることが可能です。

2.2.7　CLI

多くの場合、サーバー用Linuxは**CLI**（キャラクターユーザーインターフェース）によって操作することになります。映画やアニメなどでハッカーが真っ黒の背景に文字だけ表示されている画面とにらめっこしているシーンがよくありますが、あのような操作画面のことです。

```
[                    ]# ping google.com
PING google.com (216.58.199.238) 56(84) bytes of data.
64 bytes from kix05s02-in-f14.1e100.net (216.58.199.238): icmp_seq=1 ttl=54 time
=2.39 ms
64 bytes from kix05s02-in-f14.1e100.net (216.58.199.238): icmp_seq=2 ttl=54 time
=2.47 ms
64 bytes from kix05s02-in-f14.1e100.net (216.58.199.238): icmp_seq=3 ttl=54 time
=2.38 ms
64 bytes from kix05s02-in-f14.1e100.net (216.58.199.238): icmp_seq=4 ttl=54 time
=2.40 ms
^C
--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3002ms
rtt min/avg/max/mdev = 2.381/2.413/2.478/0.038 ms
```

CLIの例

CLIではテキストによってコマンドを入力しコンピューターを操作します。コマンドを入力する画面のことを**コンソール**と呼びます。例に挙げた画像では「ping google.com」というコマンドを入力し実行しています。

2.3 Linux/CentOS7のコマンド

コマンドには数多くの種類があるのですが、ここでは定期・APゲーム開発の上で使う基礎的なコマンドのみ扱います。

2.3.1 pwd カレントディレクトリの表示

Linuxにはカレントディレクトリ(作業ディレクトリ)という概念があります。ディレクトリとはWindowsでいうフォルダーのことと考えて差し支えありません。カレントディレクトリとは現在作業中のディレクトリのことを指します。カレントディレクトリが存在することでいちいちフルパスで表記しなくても現在のディレクトリのファイルを操作できたりします。pwdコマンドはカレントディレクトリを絶対パスで表示してくれるコマンドです。なお、passwordではなくprint working directoryの略です。

2.3.2 cd ディレクトリ移動

cdは作業ディレクトリを移動するコマンドです。cdコマンドの後に相対パスもしくは絶対パスを指定することで作業ディレクトリを移動することができます。以下がその例です。

cd /var/log

絶対パスを利用したディレクトリ移動です。この例では/var/logというディレクトリに移動しています。なお、Windowsとは異なり絶対パスはC:¥のようなドライブ名ではなく/で始まります。

cd ..

相対パスを利用したディレクトリ移動の例です。この例では1つ上のディレクトリに移動しています。現在のディレクトリが例えば/var/logだとすると、実行後の作業ディレクトリは/varになります。

cd log

相対パスを利用したディレクトリ移動の例です。この例ではlogディレクトリの中に移動しています。現在のディレクトリが例えば/varだとすると、実行後の作業ディレクトリは/var/logになります。

cd ~

~ と書いた場合、Linuxではホームディレクトリを指します。ホームディレクトリとはユーザーごとに用意されたディレクトリのことでほとんどの場合/home/(ユーザー名)というディレクトリに存在します。例えばfoobarというユーザーがこのコマンドを実行したとすると、/home/foobarというディレクトリに移動します(表記の上では ~ と表示されることがほとんどです)。なお、単に「cd」というコマンドを実行した場合はこのコマンドと同様の意味になります。

2.3.3 mkdir ディレクトリ作成

新たにディレクトリを作成します。Windowsでいう新規フォルダー作成のようなものです。例えばカレントディレクトリが「/home/foobar」であるときに「mkdir hoge」を実行すると「/home/foobar」内に新しく「hoge」ディレクトリ

が作成され「/home/foobar/hoge」というディレクトリになります。

2.3.4　ls ファイル一覧

カレントディレクトリの中のファイルとディレクトリの一覧を確認できます。「ls」とだけ実行した場合単にファイル名とディレクトリ名の一覧のみを表示するコマンドです。これ単体で使うことはあまりなく、オプションと一緒に使われることが多いです。

lオプション

ファイル・ディレクトリ名の一覧に加え、その詳細情報を表示するオプションです。lsコマンドでは「ls -l」というように-(ハイフン)の後にオプションを指定します。lオプションはよく使うので覚える場合は「ls -l」というようにセットで覚えましょう。実行すると図のような表示になり、それぞれの表示は以下のような意味になります。

①種別とパーミッション

1文字目は種別を表しておりdであればディレクトリを、-であればファイルを表しています。また、2～10文字目はパーミッションを表します。

②所有者

そのファイル/ディレクトリの所有者と所有グループを表します。左側が所有者で右側が所有グループです。

③ファイル/ディレクトリサイズ

そのファイル/ディレクトリのサイズです。Windowsとは異なり、ディレクトリのサイズはその中に入っているファイルなどを含めたサイズではなく、ディレクトリそのもののサイズを表します。

④更新日時

そのファイル/ディレクトリの更新日時です。こちらもWindowsとは異なり、ディレクトリの更新日時はディレクトリそのものが更新された日時を表します。中身のファイルを変更したとしてもその上位ディレクトリの更新日時は変更されません。

⑤ファイル/ディレクトリ名

ファイルとディレクトリにつけられた名前です。ターミナルによっては属性などによって見た目などを変えてくれている場合もあります。Tera Termの場合、ディレクトリは太字で表示されるなどの機能があります。

aオプション

Linuxでは先頭が.（ピリオド）から始まるファイルやディレクトリは隠しファイル、隠しディレクトリとして扱われます。aオプションもつけてlsコマンドを実行することで隠しファイル、隠しディレクトリも含めて情報を表示できるようになります。lオプションと併用する場合「ls -la」になります。

2.3.5 su ユーザー切り替え

ユーザーを切り替えるためのコマンドで、「su (ユーザー名)」とすることで指定のユーザーに切り替わります。「su hoge」ならhogeというユーザーに切り替えるという意味になります。切り替える際にはパスワードが必要になります。ただし、rootユーザーはパスワードなしに下位のユーザーに切り替えることが可能です。なお、単に「su」と実行した場合はrootユーザーへの切り替えという意味になります。

2.3.6 vi テキストエディタ

viはテキストエディタを起動するコマンドです。「vi (ファイル名)」というように編集するファイルを指定します。指定のファイルが存在しない場合、新たにファイルを作成します。

なお、Windowsのメモ帳などと違ってテキストエディタを立ち上げたからといって**そのまま編集できるわけではありません。**viにはモードというものが存在します。主に使うのはコマンドモードと入力モードです。立ち上げてすぐはコマンドモードになっています。コマンドモードでは各種コマンドを入力して操作を行います。主なコマンドは以下の通りです。

:q

保存せずに終了します。変更したにも関わらず保存せずに終了しようとすると警告が出るので、その際は「:q!」というコマンドを実行します。

:wq

上書き保存して終了します。指定のファイルが存在しない場合、指定の名前で保存します。

/検索文字列

指定の文字列を検索します。

コマンドモードでは編集はできません。編集をする場合入力モードに移行します。入力モードに移行するためにはコマンドモードの画面でiキーを押します。すると画面左下に「INSERT」というように表示され、ファイルの編集が可能になります。

なお、右クリックでコピーしてあるテキストをペーストすることができます。入力モードを離脱してコマンドモードに戻りたい場合ESCキーを押します。それを踏まえて、新規テキストファイルの作成から編集、保存までをまとめると以下のようになります。

・「vi 作成したいファイル名」を実行
・テキストエディタが立ち上がるのでiキーを押して入力モードへ
・内容を入力する
・入力が終わったらESCキーを押してコマンドモードへ
・「:wq」を実行し、保存して終了する

2.3.7 rm ファイル削除

rmコマンドはファイルやディレクトリを削除するためのコマンドです。なお、注意点としてWindowsと違い「ごみ箱」が存在しないので、一度消去したファイルは二度と戻ってきません。rmコマンドを実行する際は本当にそのファイルが消していいものか、スペルミスなどをしていないかなどをしっかり確認してから実行するようにしてください。

「rm 削除したいファイル名」とすることで実行できます。ディレクトリを削除したい場合、後述するrオプションと併用する必要があります。オプションをつける場合はlsコマンド同様に「rm -オプション 削除したいファイル/ディレクトリ名」というように実行します。

削除する際には本当に削除していいか確認が入ります。削除していい場合は「y」、削除をキャンセルする場合は「n」を入力しましょう。

rオプション

ディレクトリを再帰的に削除します。どういうことかというと、Windowsでいうとフォルダーの中身ごと削除するということです。ディレクトリを削除したい場合はrオプションを使う必要があります。なお、削除の際は1ファイルごとに確認が行われるため削除するファイル数が多くなる場合はfオプションとの併用をおすすめします。

fオプション

本当に削除していいか確認なしで削除するオプションです。削除しなければならないものが多い場合に使います。当然このオプションを使用した場合確認なしに削除されることになるので、本当に削除していいのか、スペルミスなどをしていないかなどをしっかりチェックする必要があります。一度消えたファイルは二度と戻ってきません。

2.3.8 chown 所有者・所有グループの変更

ファイルやディレクトリの所有者や所有グループを変更します。ディレクトリの所有者を変更する場合、Rオプションで再帰的に（中身のディレクトリやファイルも含めて）所有者を変更できます。例えばhogeというファイルの所有者、所有グループをfoobarに変更したい場合「chown foobar:foobar hoge」というようにします。hogeがディレクトリであり中身も含めて所有者を変更したい場合「chown -R foobar:foobar hoge」というように行います。

2.3.9 chmod パーミッションの変更

ファイルやディレクトリのパーミッションを変更します。パーミッションは数字形式で指定します。例えばhogeというファイルのパーミッションを「所有者、所有グループ、それ以外のユーザー全てに対し全てのパーミッションを付与」とするのであれば、「chmod 777 hoge」というようにします。また、hogeがディレクトリであり中身も含めてパーミッションを変更したい場合には「chmod -R 777 hoge」というように行います。

2.3.10 yum ソフトウェアの管理

CentOS7のソフトウェアを管理するためのコマンドです。以下のようにして利用します。yオプションを使うことで確認の際自動的にyesが選択されるようになります。

yum install (ソフトウェア名)

指定のソフトウェアをインストールします。そのソフトウェアの動作に必要なソフトウェアも同時にインストールされます。

yum update (ソフトウェア名)

指定のソフトウェアをアップデートします。単に「yum update」と実行した場合、インストールしてあるソフトウェア全てをアップデートします。

yum uninstall (ソフトウェア名)

指定のソフトウェアをアンインストールします。

なお、ソフトウェアはリポジトリという場所からインストールされます。今あるリポジトリにインストールしたいソフトウェアがない場合、yumコマンドからリポジトリをインストーリしたり、設定ファイルを記述したりしてリポジトリを追加します。

2.3.11 systemctl サービスの管理

CentOS7でサービスを管理するためのコマンドです。Webサーバーを立てたりデータベースを稼働させたりは全てサービスという仕組みによって行われます。systemctlコマンドではサービスの起動、停止、自動起動設定などを行うことができます。以下のように利用します。

systemctl start (サービス名)	サービスの起動
systemctl stop (サービス名)	サービスの停止
systemctl restart (サービス名)	サービスの再起動
systemctl enable (サービス名)	サーバーが起動した際にサービスが自動起動するようにする
systemctl disable (サービス名)	自動起動を停止させる

「systemctl enable (サービス名)」としても再起動するまでは自動起動は行われないので、サービスをインストールしてすぐに利用したい場合などには「systemctl start (サービス名)」を実行する必要があることに注意してください。

2.4 VPSの契約と初期設定

2.4.1 VPSの選定

今回はConoHa VPSを使って解説してきます。なお、ConoHa VPSはアダルト系コンテンツを設置できないため、アダルト要素を含む定期・APゲームを作りたい場合はConoHa VPSではなくアダルトコンテンツが許可されているVPSを選ぶようにしましょう(クラウドVPS by GMOなど)。ConoHa VPS以外でもKVM環境かつCentOS7が利用できるVPSであればどこでも問題ないと思います。ただし、多少手順が変わったりすることはあるかもしれません。

契約する際は契約内容などをしっかりと確認してから契約を行ってください。なお、本書の内容により何らかの不利益を被ったとしても筆者は一切の責任を負いません。契約は自己責任の下で行ってください。

アカウントの作り方、契約方法などはConoHa VPS側で丁寧に説明されているため本書で改めて解説するということはしません。続きの解説はアカウントを作成し契約が完了したところから解説をしていきます。ConoHa VPSは以下のURLからアクセスできます。

ConoHa VPS
https://www.conoha.jp/vps/

2.4.2 VPSを借りる(インスタンスを作る)

契約が完了したらConoHa VPSにログインしましょう。ログインするとConoHa VPSのダッシュボードに移動します。左メニューの一番上に「サーバー追加」があるので、そちらをクリックします。

リージョンは「東京」、サービスは「VPS」を選択します。スペックは512MB以外ならどれでも構いませんが、コスパの関係から開発中は「1GB」もしくは「2GB」をおすすめします。スペックは後から変更できるのでどちらを選んでも構いません。イメージタイプはCentOSを選択し、下のプルダウンメニューから「7.7(64bit)」を選択しましょう。これが今回利用するCentOS7の最新版になります。

次にrootパスワードを設定します。このパスワードは非常に重要なものなので、英数字や記号をしっかり混ぜたランダムなパスワードを使用しましょう。VPSはよくハッキングを狙われており、何十万回も不正アクセスを試みられます。**試みられるかもしれないではありません。必ず試みられます**。そのため、簡単なパスワードを利用していた場合VPSが乗っ取られたり犯罪行為の踏み台にされたりする場合があります。このとき設定したパスワードはしっかり管理するようにしてください。

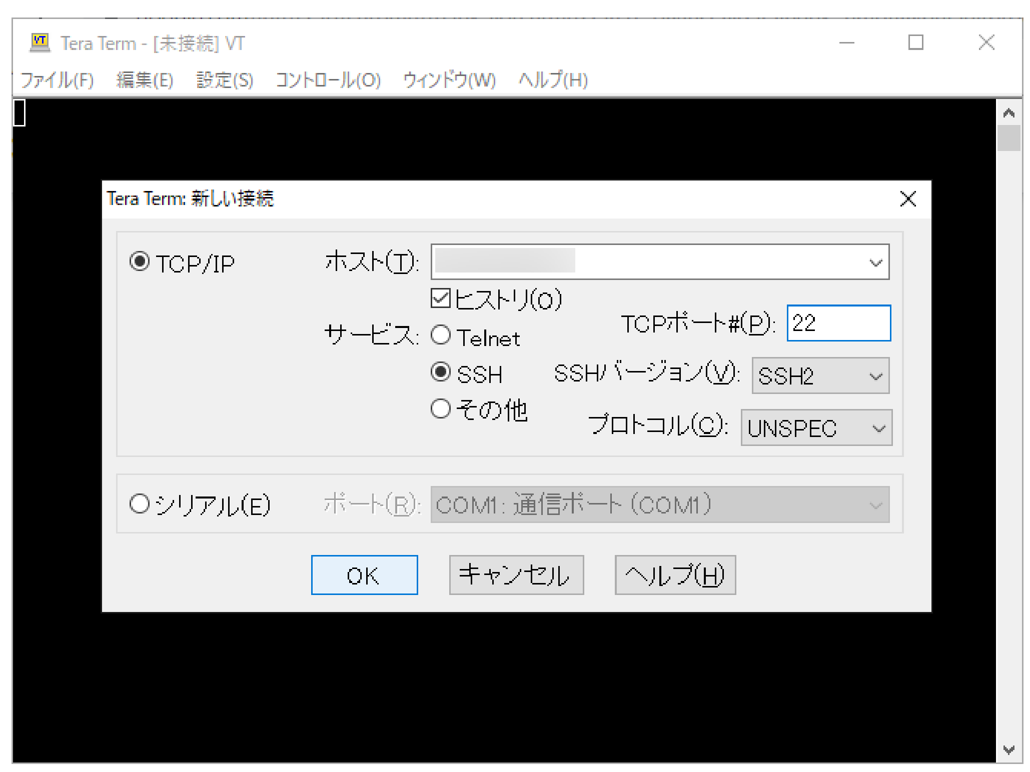
最後にネームタグを設定します。VPSにつける管理のための名前のことです。これは外部から見られるものではないので「Teiki_Game_Developing_VPS」など分かりやすい名前をつけましょう。設定が完了したら右側にある「追加」ボタンを押しましょう。待機画面になるのでしばらく待つとVPSが作成されます。なお、VPSは**サーバー削除しない限りたとえ停止したとしても課金され続ける**ので注意してください。

2.4.3　VPSに接続する

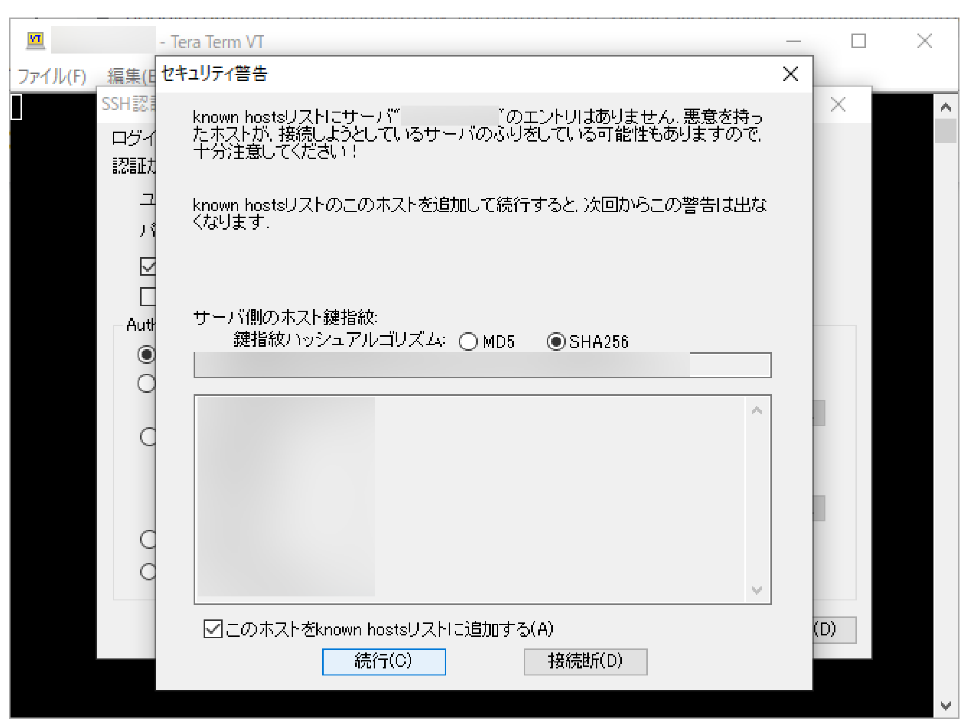
まずVPS情報を確認しましょう。登録が完了したらサーバーリストが表示されているはずです。(一度閉じてしまった場合などはConoHa VPSのコントロール左メニューから「サーバー」を選択してください。)

サーバーリストから先程レンタルしたサーバーのネームタグをクリックして詳細画面に移りましょう。ConoHa VPSでは最初からVPSが起動した状態になっているはずです。その画面から「ネットワーク情報」の中の「IPアドレス」の欄に書かれているIPアドレスを確認しておきます。

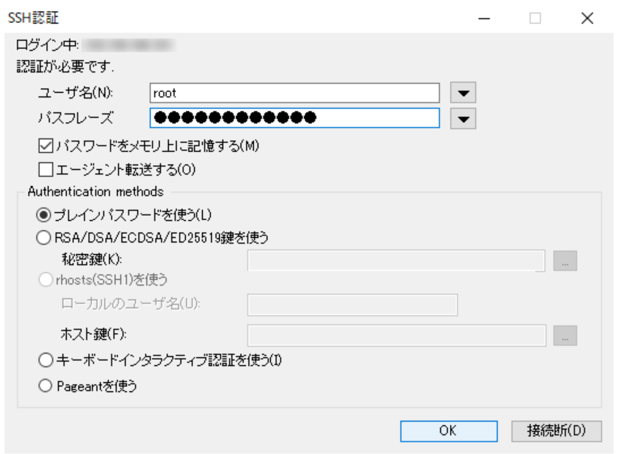
ではログインしてみましょう。PC上でTera Termを起動します。起動後すぐに「Tera Term: 新しい接続」というウィンドウが出てくるので、「ホスト」と書かれた所に先程確認したIPアドレスを入力します。その他の欄についてはそのままで構いません。入力できたら「OK」を押します。



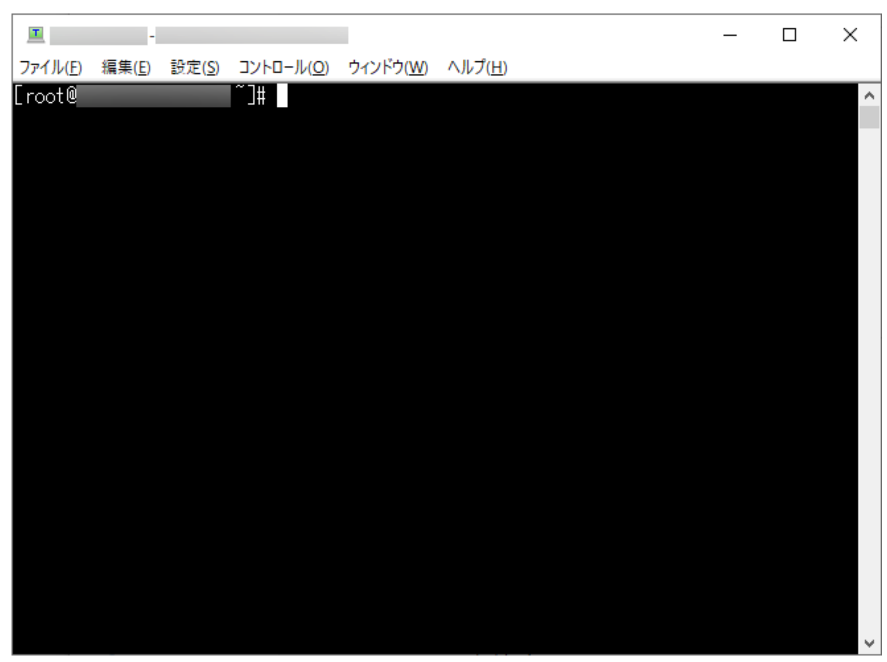
その後、セキュリティ警告が出ます。この警告は接続したことがないサーバーに接続しようとすると出るものなので、「このホストをknown hostsリストに追加する」にチェックを入れた状態で「続行」を押しましょう。これで次回以降警告が表示されなくなります。



するとSSH認証画面に移行します。まず、ユーザー名に「root」と入力します。パスフレーズには『VPSを借りる（インスタンスを作る）』で設定したrootパスワードを入力しましょう。なお、ペーストする場合はCtrl+Vではなく右クリックメニューを使うかShift+Insertを押す必要あります。入力できたら「Authentication methods」から「プレインパスワードを使う」を選択し、OKを押します。



接続に成功したら次のような画面になります。



2.4.4　全体アップデート

VPSに接続できたら環境を整えていきましょう。まずはインストールされているソフトウェアを全て最新バージョンにアップデートします。コンソールに「yum update -y」と入力し、エンターを押してコマンドを実行しましょう。なお、コピー＆ペーストをする場合、右クリックでペーストすることが可能です。

コマンドを実行するとアップデート処理が進むのでしばらく待ちます。なお、この処理は数分程度かかる場合があったり途中で止まって見えることがあったりしますが気長に待ちましょう。「Complete!」という表示が出ていれば全体アップデートは完了です。

2.4.5　Redisのインストール

まずリポジトリを追加します。リポジトリとは簡単に言えばソフトウェアのダウンロード元のことです。Redisは**EPELリポジトリ**にて配布されているのでEPELリポジトリをインストールします。「yum -y install epel-release」を実行してください。「Complete!」と表示されればEPELリポジトリのインストールは完了です。（すでにEPELリポジトリがインストールされている場合もあり、その場合は「Package ○○ already installed and latest version Nothing to do」と表示されます。この場合そのまま次の手順に進みましょう。）

EPELリポジトリのインストールが完了したらRedisをインストールします。「yum -y install redis」を実行してインストールしてください。インストール完了後、「redis-cli --version」「redis-server --version」と入力しインストールできているかチェックします。画像のようにバージョン情報が帰ってくればインストール成功です。

```
[root@              ~]# redis-cli --version
redis-cli 3.2.12
[root@              ~]# redis-server --version
Redis server v=3.2.12 sha=00000000:0 malloc=jemalloc-3.6.0 bits=64 build=7897e7d
0e13773f
```

インストールできていればサービスを有効化しましょう。これによってVPSが起動した際に自動的にRedisが起動し利用できるようになります。「systemctl enable redis」を実行します。「Created symlink…」というように帰っ

てくるか何も帰ってこなければ有効化できています。(ややこしいのですがsystemctl enableではエラーメッセージが帰ってこない限り有効化できています。)

なお、ここでインストールされるRedisは最新版ではありませんが、今回の要件では支障はないためインストールされたバージョンを使用します。

2.4.6　Node.jsのインストール

今回は最新のLTS(推奨バージョン)であるバージョン12をインストールします。インストールするにあたってリポジトリを追加します。「curl -sL https://rpm.nodesource.com/setup_12.x | bash -」を実行してください。

次に「yum -y install nodejs」とコマンドを入力して実行しましょう。Node.jsのインストールが始まります。インストールが完了したら「node -v」「npm -v」というコマンドをそれぞれ実行してみましょう。バージョン情報が帰ってくればインストール成功です。(インストールした時期によって画像とバージョンが若干違う可能性があります。)

2.4.7　Nginxのインストール

まずリポジトリを作成します。「vi /etc/yum.repos.d/nginx.repo」と入力します。viコマンドによってテキストエディタが起動するので入力モードにしてからレポジトリの内容を記述します。以下の内容をコピー＆ペーストしましょう。ペーストは右クリックで行うことができます。入力が終わったらESCキーを押してコマンドモードにし「:wq」と入力して保存して終了します。

```
[nginx-stable]
name=nginx stable repo
baseurl=http://nginx.org/packages/centos/$releasever/$basearch/
gpgcheck=1
enabled=1
gpgkey=https://nginx.org/keys/nginx_signing.key
```

リポジトリの作成が終わったら「yum -y install nginx」を実行しインストールします。終わったらインストールできているか確認してみましょう。「nginx -v」を実行しバージョン情報が帰ってきていればインストール成功です(インストールされるバージョンは画像のバージョンと異なる場合があります)。

インストールできていれば「systemctl enable nginx」を実行しサービスを有効化します。

2.4.8　MongoDBのインストール

Nginx同様にリポジトリを作成します。「vi /etc/yum.repos.d/mongodb-org-4.2.repo」を実行してテキストエディタを起動し、以下の内容を入力して保存します。

```
[mongodb-org-4.2]
name=MongoDB Repository
baseurl=https://repo.mongodb.org/yum/redhat/$releasever/mongodb-org/4.2/x86_64/
gpgcheck=1
enabled=1
gpgkey=https://www.mongodb.org/static/pgp/server-4.2.asc
```

リポジトリが作成できたら「yum -y install mongodb-org」を実行してインストールします。インストールが完了したら「mongo --version」と入力しインストールできているか確認します。バージョン情報が帰ってくれれば成功です。「systemctl enable mongod」を実行しMongoDBのサービスを有効化しましょう。mongoやmongodbではなくmongodなので注意してください。

2.4.9　再起動

ここまで設定したら「reboot」というコマンドを入力してください。VPSが再起動されます。接続が切れるのでTera Termも閉じられます。少し待ってから再度Tera Termを起動しもう一度接続しましょう。再接続したら以下のコマンドをそれぞれ入力してください。

「systemctl status nginx」
「systemctl status mongod」
「systemctl status redis」

サービスの実行状況を確認することができます。結果の表示がすべてactiveになっていれば正常に自動起動設定ができています。

```
[root@            ~]# systemctl status nginx
nginx.service - nginx - high performance web server
   Loaded: loaded (/usr/lib/systemd/system/nginx.service; enabled)
   Active: active (running) since Tue 2019-10-01 08:35:49 JST; 1min 15s ago
     Docs: http://nginx.org/en/docs/
  Process: 348 ExecStart=/usr/sbin/nginx -c /etc/nginx/nginx.conf (code=exited, status=0/SUCCESS)
 Main PID: 365 (nginx)
   CGroup: /system.slice/nginx.service
           ├─365 nginx: master process /usr/sbin/nginx -c /etc/nginx/nginx.conf
           └─366 nginx: worker process

Oct 01 08:35:49                              systemd[1]: Starting nginx - high performance web server...
Oct 01 08:35:49                              systemd[1]: Started nginx - high performance web server.
[root@            ~]# systemctl status mongod
mongod.service - MongoDB Database Server
   Loaded: loaded (/usr/lib/systemd/system/mongod.service; enabled)
   Active: active (running) since Tue 2019-10-01 08:35:50 JST; 1min 19s ago
     Docs: https://docs.mongodb.org/manual
  Process: 364 ExecStart=/usr/bin/mongod $OPTIONS (code=exited, status=0/SUCCESS)
  Process: 361 ExecStartPre=/usr/bin/chmod 0755 /var/run/mongodb (code=exited, status=0/SUCCESS)
  Process: 356 ExecStartPre=/usr/bin/chown mongod:mongod /var/run/mongodb (code=exited, status=0/SUCCESS)
  Process: 350 ExecStartPre=/usr/bin/mkdir -p /var/run/mongodb (code=exited, status=0/SUCCESS)
 Main PID: 368 (mongod)
   CGroup: /system.slice/mongod.service
           └─368 /usr/bin/mongod -f /etc/mongod.conf

Oct 01 08:35:49                              mongod[364]: about to fork child process, waiting until server is ready for connections.
Oct 01 08:35:49                              mongod[364]: forked process: 368
Oct 01 08:35:50                              mongod[364]: child process started successfully, parent exiting
Oct 01 08:35:50                              systemd[1]: Started MongoDB Database Server.
[root@            ~]# systemctl status redis
redis.service - Redis persistent key-value database
   Loaded: loaded (/usr/lib/systemd/system/redis.service; enabled)
  Drop-In: /etc/systemd/system/redis.service.d
           └─limit.conf
   Active: active (running) since Tue 2019-10-01 08:35:49 JST; 1min 30s ago
 Main PID: 347 (redis-server)
   CGroup: /system.slice/redis.service
           └─347 /usr/bin/redis-server 127.0.0.1:6379

Oct 01 08:35:49                              systemd[1]: Starting Redis persistent key-value database...
Oct 01 08:35:49                              systemd[1]: Started Redis persistent key-value database.
```

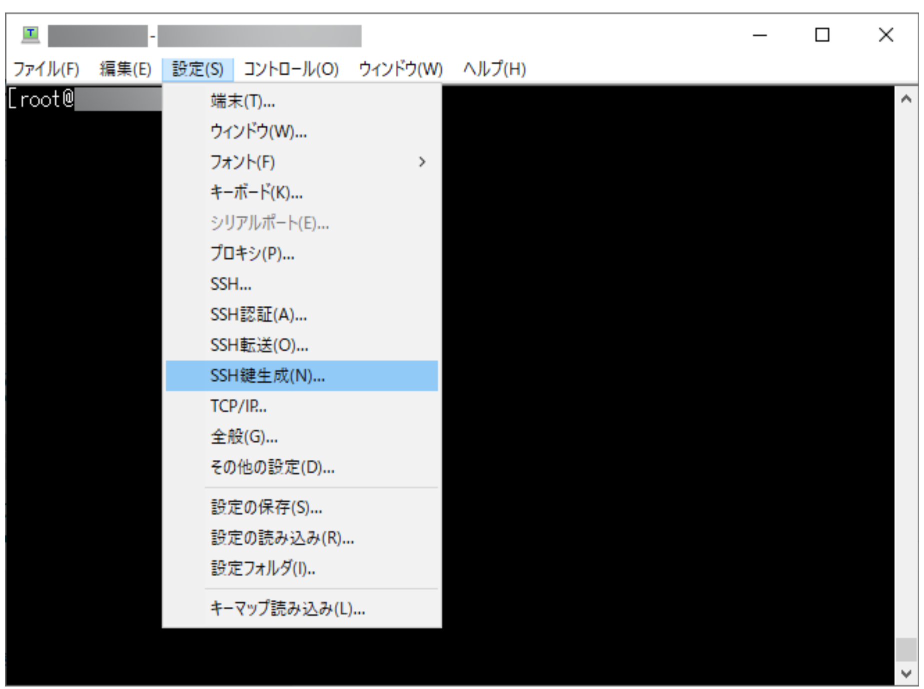
2.5 基礎的なセキュリティ設定

2.5.1　一般ユーザーの作成

　Linuxではrootユーザーはありとあらゆる権限を持っています。そのためセキュリティや操作ミスのことを考えるとrootユーザーで何でもやってしまうのは好ましくありません。rootユーザーの使用は本当に必要な場合だけにして普段は一般ユーザーを使うのが普通です。なのでまずはそのための一般ユーザーを作りログインできるようにしていきましょう。

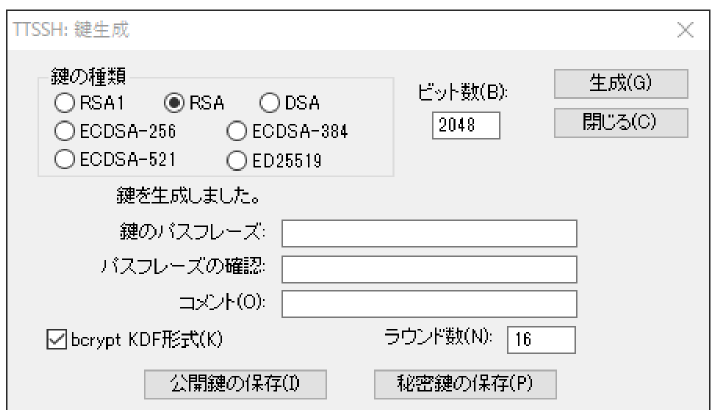
　まず作成するユーザーがログインするときに使う鍵になるファイルを作ります。現実でもセキュリティのために玄関の扉に錠前をつけ錠前に合う鍵でのみ入れるようにするように、サーバーでも錠前にあたるファイルをセットしてそれに合致する鍵ファイルでのみログインできるようにします。

　鍵になるファイルはクライアント側（Tera Term側）で作成します。まずはTera Termの上メニューの「設定」から「SSH鍵生成」を選択します。



　「TTSSH: 鍵生成」というウィンドウが出てくるので「生成」ボタンを押してください。なお鍵の種類、ビット数はデフォルト設定のままで問題ありません（RSA、2048）。

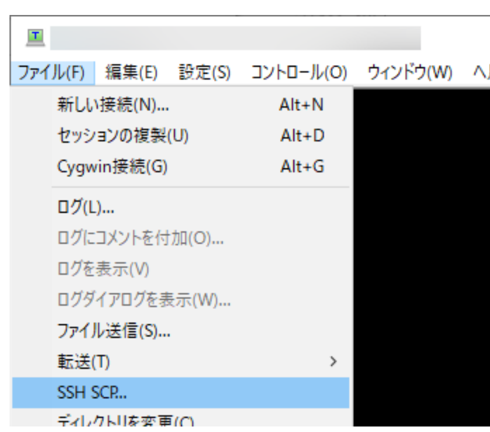
少し待つと鍵のパスフレーズなどが入力できるようになります。鍵のパスフレーズというのはSSH鍵を使うためのパスワードです。万が一SSH鍵が流出してしまった際にもパスワードがわからない限りSSH鍵が使えないようになります。ただし、**ここは空に設定します**。Visual Studio Codeで遠隔操作する際に支障が出るためです。コメントは空にしても構いません。



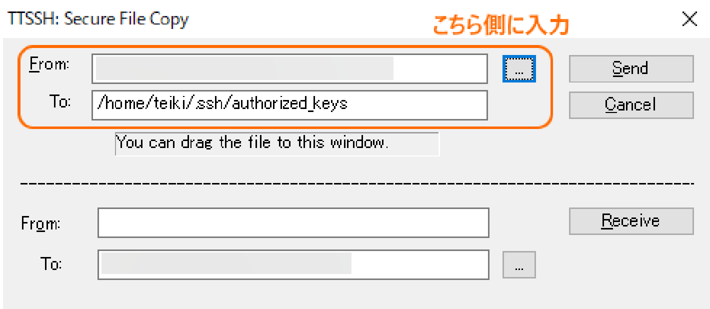
ここまで入力したら「公開鍵の保存」「秘密鍵の保存」と押しそれぞれ保存します。**これは両方行います**。間違えて片方しか保存していなかった場合、この項目の最初からやり直してください。なお、秘密鍵の保存の際に警告が表示されますが「はい」を押して続行します。保存する際はどちらが公開鍵でどちらが秘密鍵か覚えておきましょう。デフォルト名のままであれば公開鍵が「id_rsa.pub」、秘密鍵が「id_rsa」になります。保存し終わったら「TTSSH: 鍵生成」ウィンドウを閉じます。

それでは実際にユーザーを作成していきましょう。ここではteikiというユーザーを作成することにします。以降のコマンドなどは作成するユーザー名によって適宜読み替えてください。分かりやすくするため、読み替えるべき場所は背景色をこのように変えてあります。

Tera Termのコンソールに戻り、「useradd teiki」というコマンドを入力します。次に、「cd /home/teiki」を実行しteikiユーザーのホームディレクトリに移動します。次に公開鍵を保存するためのディレクトリを作成します。「mkdir .ssh」を実行してください。次にTera Termの上メニューの「ファイル」から「SSH SCP」を選択します。

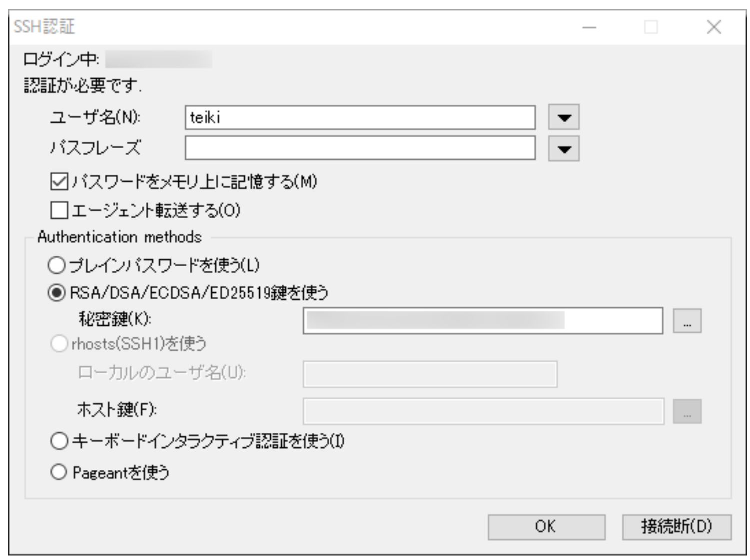


「TTSSH: Secure File Copy」というウィンドウが出てくるので、**上側**の「From:」と書かれた入力欄の右の「...」をクリックし先程保存した公開鍵ファイルを選びます(**秘密鍵ではなく公開鍵の方です**)。その下の「To:」と書かれた部分には「/home/teiki/.ssh/authorized_keys」と入力してください。ここまで入力したら「Send」をクリックしましょう。VPSに公開鍵ファイルが送信されます。



次にアップロードした公開鍵のパーミッションを変更します。このファイルはteikiユーザー以外が使用する必要はないので「chmod -R 700 .ssh」と入力しパーミッションを限定します。次に公開鍵ファイルの所有者を変更します。rootユーザーで作業した関係で公開鍵の所有者がrootになっているので「chown -R teiki:teiki .ssh」を実行し所有者をteikiに変更します。

ここまで設定したら実際にteikiユーザーでログインできるか確かめてみましょう。新しくTera Termを起動します。SSH認証のところでユーザー名に「teiki」、パスフレーズには何も指定せず「Authentication methods」は「RSA/DSA/ECDSA/ED25519鍵を使う」を選び、保存した秘密鍵を選択してください。(**公開鍵ではなく秘密鍵の方です**)。ファイル名をデフォルトから変えている場合、選択ウィンドウの右下の「秘密鍵ファイル」と書かれたところを「すべてのファイル」に変更しないと出てこないことがあります。



「OK」を押して接続されたら正常に設定できています。teikiユーザーでログインしているのでコンソールの左の部分が「[teiki@…… ~]$」となっているはずです。

```
[teiki@            ~]$
```

2.5.2　rootログインの禁止

teikiユーザーでログインできたらここからrootユーザーになってみましょう。「su」を実行します。パスワードを尋ねられるのでrootパスワードを入力します。ペーストする場合は右クリックします。(パスワードを入力しても見た目が変わりませんが入力できています。)以下のような感じになりrootユーザーになっていることがわかります。

```
[root@            teiki]#
```

rootユーザーに戻ったらrootユーザーとパスワードでの直接的なログインを禁止するように設定変更しましょう。先述の通り不正アクセスが非常によく試みられるためログイン処理はなるべく厳格にします。さて、CentOS7では遠隔操作でのログインはsshdというサービスにより管理されているのでその設定を変更します。

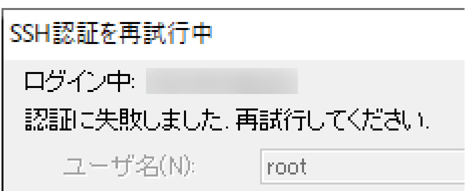
まずは「vi /etc/ssh/sshd_config」を実行します。表の左に書いてある行を見つけ、右の内容に書き換えて保存してください。

変更前	変更後
PermitRootLogin yes	PermitRootLogin no
PasswordAuthentication yes	PasswordAuthentication no
#ClientAliveInterval 0	ClientAliveInterval 60
#ClientAliveCountMax 3	ClientAliveCountMax 3

設定内容は以下のようになっています。ログイン処理を厳格にしている他、定期的に接続チェックを行って接続が勝手に切れないようにしています。

・rootでのログインを禁止する。
・パスワードでのログインを禁止する。
・SSH接続のチェックを60秒ごとに行う。
・3回連続でSSH接続のチェックに失敗した場合切断する。

保存したら「systemctl restart sshd」を実行します。そうするとsshdが再起動され設定が反映されます。実際にrootで直接ログインできなくなっているか試してみましょう。一度Tera Termを閉じ再度起動して『VPSに接続する』と同様の手順でログインを試みます。ログイン**できなく**なっていれば正常に設定できています。接続断をクリックしてログインを中止しましょう。



次にこの状態からrootにログインするまでをチェックします。もう一度Tera Termを再起動し秘密鍵を使ってteikiユーザーでログインします。ログインしたら「su」を実行しましょう。パスワードの入力を求められるので、rootパスワードを入力します。ペーストする場合は右クリックです。rootユーザーになることができれば成功です。

2.5.3　SSHポートの変更

この項目ではroot権限が必要になるのでrootユーザーでログインしておいてください。

今までやってきた遠隔操作でサーバーにログインしたりコマンドを実行したりといった通信には**SSH**というものが利用されています。SSHは初期設定では22番ポートが使用されています。そのため乗っ取りを狙って22番ポートに攻撃が集中するので、その対策としてSSHで使用するポートを変更します。

1024～49151番ポートのうち他のソフトウェアとかぶらないポートの中からランダムに選びましょう。他のソフトウェアが利用しているポートは以下のURLが参考になります。今回はここでは例としてポートを16815にします。設定するポート番号に適宜読み替えてください。

TCPやUDPにおけるポート番号の一覧 - Wikipedia
https://ja.wikipedia.org/wiki/TCP%E3%82%84UDP%E3%81%AB%E3%81%8A%E3%81%91%E3%82%8B%E3%83%9D%E3%83%BC%E3%83%88%E7%95%AA%E5%8F%B7%E3%81%AE%E4%B8%80%E8%A6%A7

まず「vi /etc/ssh/sshd_config」を実行します。最初の方に「#Port 22」と書かれた行があるので、その下の行に「Port 16815」を追記し上書き保存します。保存したら「systemctl restart sshd」を実行しましょう。設定が反

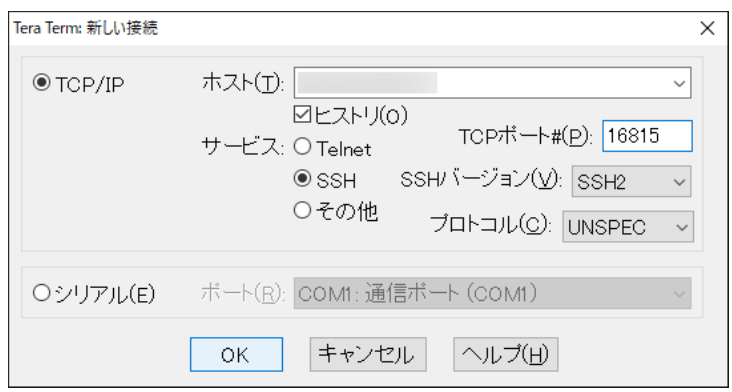
映されます。

なお、これ以降この項目が終わる前にTera Termの接続を切ってしまうとログインできなくなってしまいます。その時は『補足：途中で接続が切れてしまった場合』を参照してください。

次はSSHのために開放するポート番号を変更しましょう。CentOS7では開放するポート番号の設定は**firewalld**で行います。まず、初期のSSHポートの接続を無効化します。「firewall-cmd --permanent --remove-service=ssh」を実行しましょう。次に16815ポートを開放していきましょう。まずは元の設定ファイルをコピーします。「cp /usr/lib/firewalld/services/ssh.xml /etc/firewalld/services/ssh-16815.xml」を実行してください。

実行したら、設定ファイルを編集するため「vi /etc/firewalld/services/ssh-16815.xml」を実行します。テキストエディタが立ち上がるので、「<port protocol="tcp" port="22"/>」となっている所を「<port protocol="tcp" port="16815"/>」というように書き換えてください。書き換え終わったら保存して終了し、「firewall-cmd --permanent --add-service=ssh-16815」を実行します。

最後に「firewall-cmd --reload」を実行しましょう。これで設定が反映され、SSHのポート設定が完了します。Tera Termを終了しもう一度接続してみましょう。TCPポートが22のままでは接続できなくなっているはずです。TCPポートの欄を16815に書き換えると接続できるようになります。



なおポートが違うと別の接続先扱いになるので再びセキュリティ警告が表示されますが、初回同様に許可してください。その後はteikiユーザーでログインしたときと同じ方法でログインできます。

2.5.4　補足：途中で接続が切れてしまった場合

ConoHa VPSではブラウザ経由でもコンソールにアクセスすることができます。なお、ConoHa VPS以外でもほとんどのVPSサービスでは同様のことが可能かと思われます。ConoHa VPSコントロールパネルにログインし、左メニューのサーバーからレンタルしているVPSを選択、「コンソール」と書かれたボタンをクリックします。

新しいウィンドウでコンソールへ接続されログインが求められるのでまずはユーザー名を入力します。今回の場合は「root」と入力します。次にパスワードが求められるのでrootパスワードを入力しましょう。ログインに成功したらVPSの操作ができるようになるので続きの処理を行いましょう。

2.5.5　MongoDBのセキュリティ設定

まずMongoDBの管理ユーザーを作りましょう。これはCentOS7のユーザーとは全く別個のもので、MongoDB内でのみ使われるユーザーです。最初に「mongo」というコマンドを実行します。するとMongoDBのコンソールに移行します。いくつかWARNINGが出ていますが無視して構いません。

```
[teiki@            ~]$ mongo
MongoDB shell version v4.2.1
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("                                    ") }
MongoDB server version: 4.2.1
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
        http://docs.mongodb.org/
Questions? Try the support group
        http://groups.google.com/group/mongodb-user
Server has startup warnings:
2019-11-15T20:10:57.197+0900 I  STORAGE  [initandlisten]
2019-11-15T20:10:57.197+0900 I  STORAGE  [initandlisten] ** WARNING: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine
2019-11-15T20:10:57.197+0900 I  STORAGE  [initandlisten] **          See http://dochub.mongodb.org/core/prodnotes-filesystem
2019-11-15T20:10:58.140+0900 I  CONTROL  [initandlisten]
2019-11-15T20:10:58.140+0900 I  CONTROL  [initandlisten] ** WARNING: Access control is not enabled for the database.
2019-11-15T20:10:58.140+0900 I  CONTROL  [initandlisten] **          Read and write access to data and configuration is unrestricted.
2019-11-15T20:10:58.140+0900 I  CONTROL  [initandlisten]
2019-11-15T20:10:58.141+0900 I  CONTROL  [initandlisten]
2019-11-15T20:10:58.141+0900 I  CONTROL  [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/enabled is 'always'.
2019-11-15T20:10:58.141+0900 I  CONTROL  [initandlisten] **        We suggest setting it to 'never'
2019-11-15T20:10:58.141+0900 I  CONTROL  [initandlisten]
2019-11-15T20:10:58.141+0900 I  CONTROL  [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/defrag is 'always'.
2019-11-15T20:10:58.141+0900 I  CONTROL  [initandlisten] **        We suggest setting it to 'never'
2019-11-15T20:10:58.141+0900 I  CONTROL  [initandlisten]
---
Enable MongoDB's free cloud-based monitoring service, which will then receive and display
metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---

> 
```

次に「use admin」と入力しましょう。MongoDBの管理者用データベースにスイッチします。ここで管理ユーザーの名前とパスワードを決めてください。ここでは例として管理ユーザーの名前を「admin」、パスワードを「q%sv|N#IE99i」とします。適宜読み替えてください。分かりやすくするため、読み替えるべき場所は背景色をそれぞれ変えてあります。

注意：MongoDBは以降の設定によりインターネットからアクセス可能になります。
このパスワードは絶対に使用せず、独自かつ十分な強度をもったものを使用してください。

まずは管理ユーザーを作成します。以下のコマンドを実行してください。（最後のdb:"admin"はユーザー名ではなくMongoDBの管理者用データベースのことを指します。変更しないでください。）

```
db.createUser({user:"admin", pwd:"q%sv|N#IE99i", roles:[{ role:"userAdminAnyDatabase", db:"admin" }]})
```

これで管理ユーザーが作成されます。作成したら試しに管理ユーザーでMongoDBにログインしてみましょう。「db.auth("admin", "q%sv|N#IE99i")」と入力してください。「1」と帰ってくればログイン成功です。ここまで実行したらMongoDBのコンソールから離脱しましょう。「exit」と入力するか「Ctrl+C」を押すと離脱できます。

```
> use admin
switched to db admin
> db.createUser({user:"admin", pwd:"q%sv|N#IE99i", roles:[{ role:"userAdminAnyDatabase", db:"admin" }]})
Successfully added user: {
        "user" : "admin",
        "roles" : [
                {
                        "role" : "userAdminAnyDatabase",
                        "db" : "admin"
                }
        ]
}
> db.auth("admin", "q%sv|N#IE99i")
1
```

管理ユーザーが作成できたらセキュリティ認証を追加します。ここからはroot権限が必要になるのでrootユーザーではない場合rootユーザーに切り替えてください。「vi /etc/mongod.conf」を実行してください。「#security:」と書かれている次の行の場所に以下の内容を追記します。

```
security:
  authorization: enabled
```

SSHと同様にデフォルトのポートも変更しましょう。ここでは例としてポートを34189に変更します。設定するポートによって適宜読み替えてください。

```
net:
  port: 27017
  bindIp: 127.0.0.1  #(省略)
```

と書かれているところを以下のように書き換えます。

```
net:
  port: 34189
  bindIp: 0.0.0.0
```

ここまで設定したら上書き保存して終了し「systemctl restart mongod」と入力します。MongoDBが再起動され設定が反映されます。なお、これ以降コンソールからMongoDBにアクセスする際は「mongo」ではなく「mongo --port 34189」と入力する必要があります。

次にMongoDB Compassから利用できるようにするため、MongoDBをインターネットからアクセスできるようにします。**必ずMongoDBのセキュリティ設定を行ってからこの手順を行ってください。**今回のMongoDBの利用ポートは34189なのでこれを開放します。「firewall-cmd --zone=public --add-port=34189/tcp --permanent」を実行してください。次に、「firewall-cmd --reload」を実行すれば設定が反映されMongoDBがインター

ネットからアクセス可能になります。

2.6 Webサーバー設定

2.6.1　Nginxとは

本書ではWebサーバーを立てるのにNginxを利用します。NginxとはWebサーバーソフトウェアの1つで、Webサーバーを立てるにあたって必要な様々な機能が搭載されています。似たようなものにApacheがありますが、Nginxの方がより高速で軽量です。また、設定ファイルの記述方法もApacheより直感的です。

2.6.2　httpポートの開放

Nginxを利用するにあたってまずはhttp通信用のポートを開放しましょう。firewalldにはhttp通信用の設定ファイルがすでに用意されているので、「firewall-cmd --permanent --add-service=http」を実行するだけでポート開放ができます。実行したら「firewall-cmd --reload」をして設定を反映しましょう。

インストール時に行った設定によってNginxはすでに起動しているので、これでサーバーにアクセスできるようになります。ためしにブラウザを起動し「http://(レンタルしているVPSのIP)/」にアクセスしてみましょう。「Welcome to nginx!」というページが表示されていればOKです。

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

2.6.3　ドメインの基礎知識

ドメインとはインターネット上の住所のようなものです。IPアドレスは人間にとって非常に分かりづらいもののため、それを人間に分かりやすくするために作られました。「http://www.example.com/」を例に取ると、ドメイン名は以下の強調表示された部分のことを指します。

https://www.**example.com**/

「www」などの部分は**ホスト名**といいます。また、「www.example.com」などのようにホスト名とドメイン名を合わせたものを**FQDN**（Fully Qualified Domain Name：完全修飾ドメイン名）と呼びます。ドメインを取得する場合はホスト名を含めたものではなく、ドメイン名のみを取得します。例えば「http://teikigame.example.com/」というようなURLにしたい場合は「example.com」を取得するようにするということです。

また、ドメインはネームサーバーというサーバーによって管理されます。ドメインを取得したら利用するサービスが公開しているネームサーバーに管理を任せるのが一般的です。ConoHaのネームサーバーは以下のURLから確認することができます。2019年11月1日現在は「ns-a1.conoha.io」「ns-a2.conoha.io」がネームサーバーのようです。続く解説ではネームサーバーがこれらのFQDNであることを前提に解説していますが、もし変更されていた場合や違うVPSを利用している場合などは適宜読み替えてください。

ネームサーバーとは｜ConoHa WINGサポート
https://support.conoha.jp/w/%E3%83%8D%E3%83%BC%E3%83%A0%E3%82%B5%E3%83%BC%E3%83%90%E3%83%BC%E3%81%A8%E3%81%AF/

次の項目では実際にドメインを取得していきます。無料編と有料編に別れているのでどちらかに進みましょう。無料で取得できるドメインにはそれなりに制約なども存在するので一長一短です。お好きな方を選んでください。

2.6.4　ドメインの取得（無料編）

ドメインはFreenomというサービスを利用することにより無料で取得することもできます。しかし、無料のドメインはいろいろとデメリットが多いです。使えるドメインの種類が少なかったり、規約が英語であったり、個人情報をこのサイトに渡す必要があったり（個人情報が必要なのは有料ドメインも一緒ですが信頼性の問題として）、更新手続きが面倒だったり、突然使えなくなってしまった事例もあったりします。そのため利用するとしても無料ドメインの利用は開発中のみに留め、実際に定期・APゲームを公開する際は有料ドメインを利用することをおすすめします。それでは実際にドメインを取得してみましょう。まずFreenomにアクセスします。

Freenom - 誰でも利用できる名前
https://www.freenom.com/ja/index.html

「新しい無料ドメインを探します」という欄があるので、取得したいドメインを入力して「利用可能状況をチェックします」を押しましょう。いろいろな種類のトップレベルドメインと取得可能であるかどうかのリストが出てきます。（無料でないものもあるので注意してください。）「利用不可」と表示されている場合、すでにそのドメインは他の人に取得されてしまっているので別のドメインを考えましょう。

「USD 0.00　今すぐ入手！」と書かれているものは無料で取得可能です。取得するドメインが決まったら、「今すぐ入手」を押しましょう。「今すぐ入手」をクリックすると「選択」に変化します。その状態で右上の「チェックアウト」をクリックしましょう。取得処理画面に移ります。なお、日本語化されているのはここまででここからは英語になります。

「Domain」は取得するドメインです。取得したいドメインかどうか念の為もう一度確認しておきましょう。問題がなければ「Use your new domain」のところから「Use DNS」を選び、さらにその中から「Use your own DNS」を選択してNameserverのところに「ns-a1.conoha.io」「ns-a2.conoha.io」をそれぞれ入力してください。「IP Address」のところは入力しなくても構いません。

「Period」はドメインの利用期間です。選択肢の中から選べます。料金も書かれているので（@ FREEと書かれているものが無料）、@ FREEとなっているもののうち最長のものを選べば問題ないでしょう。今回は「12 Months @ FREE」を選択します。なお、利用期間が切れる15日前から更新処理が行えるようになります。

ここまで設定したら右下の「Continue」を押します。ここでログインするかアカウントを新規登録する必要があります。おそらくアカウントを持っていないと思うので登録していきましょう。「Please enter your email address and click verify to continue to the next step」と書かれている所の下にある「Enter Your Email Address」の所に登録したいメールアドレスを入力します。入力したら「Verify My Email Address」をクリックします。

すると「Verification link sent to your email (～～～～). The link is valid for only 24 hours. Go

to your email inbox and click on the link.」と表示されたページに移行します。メールアドレスの確認メールが届いているはずなのでそのメールを開いて、その中の「Before completing your order, please confirm your email address by pressing the following link:」と書かれている所のURLを開いてください。届いていない場合、迷惑メールフォルダーなどに入っていないか確認してみてください。

URLをクリックするとアカウントの開設&個人情報の入力画面に移ります。それぞれの項目の意味は以下のとおりです。すべてローマ字で記入してください。

First Name	名前
Last Name	名字
Company Name	会社名(必須ではないようです)
Address 1	市より下の住所
Zip Code	郵便番号(必須ではないようです)
City	市
Country	国
State/Region	都道府県
Phone Number	電話番号(必須ではないようです)
Email Address	メールアドレス
Password	使用するパスワード
Confirm Password	使用するパスワード(再入力)

最後にTerms & Conditionsを確認し、同意する場合はチェックを入れて「Complete Order」をクリックしましょう。取得処理が完了します。取得したドメインを管理する場合以下のURLにアクセスします。

Client Area - Freenom
https://my.freenom.com/clientarea.php

登録したメールアドレスとパスワードを入力し「Login」をクリックしましょう。ログインに成功したら右上メニューの「Services」より「My Domains」をクリックします。すると取得したドメインの一覧が表示されるので「Manage Domain」をクリックしましょう。移動した先のページのManagement Toolsから管理ができます。なお、ドメインを解約する場合もここから行えます。

2.6.5 ドメインの取得(有料編)

すでに設定したいドメインが決まっている場合、この段階から有料のドメインを取得するのも悪くないでしょう。ドメインの取得は早い物勝ちなので開発などをしている間に他の人に取得されてしまう可能性があるからです。

さて、有料ドメイン取得サービスには色々ありますがどこを選んでも大差はないと思います。今回はムームードメインを例に取得方法を解説します。まずムームードメインにアクセスしましょう。なお、ムームードメインで登録する場合SMSもしくは自動音声電話による認証が必要になります。

ムームードメイン
https://muumuu-domain.com/

「欲しいドメインを入力」という欄があるので、取得したいドメインを入力して「検索する」を押しましょう。いろいろな種類のトップレベルドメインと取得可能であるかどうかのリストが出てきます。表示されている料金は初年度の税抜料金です(2年目以降の料金は異なる場合があります)。「取得できません」と表示されている場合、すでにそのドメインは他の人に取得されてしまっているので別のドメインを考えましょう。

取得するドメインが決まったら、「カートに追加」ボタンを押しましょう。ボタンが「お申し込みへ」に変わるのでクリックします。ここでログインするかアカウントを新規登録する必要があります。おそらくアカウントを持っていないと思うので登録していきましょう。

「新規登録」をクリックします。新規ユーザー登録画面になるので、ムームーID(メールアドレス)と書かれたところに登録するメールアドレスを、パスワードのところに登録するパスワードを入力しましょう。「利用規約」を確認し、同意できれば「同意して本人確認へ」をクリックします。

「SMS認証による本人確認」という画面になるので、「確認方法の受け取り方法」でSMSを選択し、電話番号を入力します。SMSが利用できない場合は自動音声確認を選択してください。入力したら「認証コードを送信する」をクリックします。認証コードが届くので、「認証コード入力」画面にて届いたコードを入力してください。入力したら「本人確認をして登録する」をクリックします。登録の続きを行います。ムームードメインのトップページに戻ってください。

戻ったら右上メニューの人型のアイコンをクリックします。すると、「個人情報に不足部分があります。ユーザー登録情報変更より個人情報を登録してください。」と出て「ユーザー登録情報変更」画面に移動します。「お客様情報(弊社管理用)」と書かれた所に個人情報を記入し、「更新」をクリックします。「ユーザー登録情報を修正してもよろしいですか?」と確認が出るので「OK」を押しましょう。「ユーザー登録情報を変更いたしました」と出れば登録完了です。「OK」を押しましょう。

右上メニューのカートアイコンをクリックします。出てきたウィンドウから「お申込みへ」を選択しましょう。実際にドメインの取得に移ります。WHOIS公開情報は「弊社の情報を代理公開する」、ネームサーバー(DNS)は「GMOペパボ以外のサービス」を選択し、「ns-a1.conoha.io」「ns-a2.conoha.io」をそれぞれ選択してください。

オプションサービスのところはすべてチェックを外して構いません。契約年数を選び、支払情報を入力して「次のステップへ」をクリックします。連携サービスを勧められますがこれも無視して構いません。再び「次のステップへ」をクリックします。

契約内容の確認に移ります。もう一度契約内容を確認しましょう。また利用規約への同意を求められるので確認の上で同意できればチェックし、「取得する」をクリックします。「ドメインを取得中です。」と出るのでしばらく待ちましょう。「取得が完了しました。」と出れば取得完了です。

取得したドメインを管理する場合、右上メニューの人型アイコンをクリックすることでコントロールパネルに移動できます。

2.6.6　ドメインの設定

ドメインを取得したら設定を行いましょう。今回は「siroisakana.com」というドメインを取得したものとして解説します。分かりやすいよう読み替えるべき部分についてはこのように背景色を変えているので、適宜読み替えてくださ

い。ドメインの設定のことを**DNSレコード**と呼びます。

まずはConoHa VPSコントロールパネルにログインします。左メニューに「DNS」というものがあるので開き、「＋ドメイン」と書かれたボタンをクリックします。管理するドメインを聞かれるので「siroisakana.com」と入力しましょう。GeoDNSは無効で構いません。入力したら保存を押します。

ドメインリストにドメインが追加されるので、「〉 siroisakana.com」となっている所をクリックし展開します。右の鉛筆ボタンをクリックし、さらに出てきた＋ボタンをクリックします。今回は開発用サーバーということで「dev.siroisakana.com」というFQDNを作りましょう。設定内容を入力できるようになっているので、「A(通常)」「dev」「3600」「(レンタルしているVPSのIP)」を入力し保存を押しましょう。

「http://dev.siroisakana.com/」にアクセスしてみてください。運が良ければすでにアクセスできるかもしれませんし、まだ設定が反映されておらずアクセスできないかもしれません。

DNSレコードの設定の反映には時間がかかるので気長に待ちましょう。数分程度で反映されることもあれば最大72時間かかることもあります。(とはいえたいてい早ければ数分程度、長くても2,3時間程度で反映されることが多いです。)待っている間に本書を読み進めるのも悪くないかもしれません。時間を開けて何度かアクセスしながら反映されているか確認しましょう。

2.6.7　https化(SSL/TLSの設定)

ここではWebサイトをhttps化する方法について解説します。https通信を利用することで通信経路上での盗聴が困難になりセキュリティが向上する他、http2を利用できるようになり通信が高速化するなどのメリットがあります。ただし、https通信を利用する場合httpsのサイト内でhttpによる画像などが表示できなくなる恐れがあります(混在コンテンツのブロック)。定期・APゲームはhttpによる外部の画像を参照することが多いので、https化する場合はそのあたりを考慮に入れる必要があるでしょう。https化しない場合、次の項目の『httpでのnginxの設定』に進んでください。

https化にあたって、まずはSSL/TLSについて簡単に解説します。SSL/TLSとは通信を暗号化する技術のことで、**証明書**というものを利用します。(なお証明書という名前ですが認証用のファイルであり実際に書類のような形状で存在するわけではありません。)証明書は**認証局**という機関によって発行されます。認証局に証明書を発行してもらう場合ほとんどは有料になりますが、**Let's encrypt**という認証局では無料かつ自動で証明書を発行してもらえます。世の中に暗号化通信を広めるために設置された認証局であり、数々の大企業がスポンサーになっているため将来性も問題ありません。今回はこの認証局を利用してhttps化を行います。

それでは実際にhttps化していきましょう。ここからの作業にはroot権限が必要になるのでrootユーザーでログインしておいてください。まずは、Let's encryptで認証するためのソフトウェアであるcertbotをインストールします。certbotのあるEPELリポジトリはすでにインストールしてあるので、そのまま「yum -y install certbot」を実行します。少し待つとcertbotがインストールされます。

インストールが終わったらLet's encryptから証明書を取得しましょう。今回は「dev.siroisakana.com」の証明書を作ります。なお、ここでは実際にドメインを取得しているかどうか確認されるので、設定したドメインがまだ反映されていない場合はこの項目は後回しにしてください。『Node.js』の『ノンブロッキングIO』のあたりまでは進めます。ドメインが反映されていれば以下のコマンドを実行しましょう。

```
certbot certonly --webroot -w /usr/share/nginx/html -d dev.siroisakana.com
```

初回実行の際はメールアドレスの入力や規約への同意が必要になります。まず「Enter email address ～～」とメールアドレスの入力が求められるので入力します。次に規約への同意が求められるので表示されたURLの内容を読み、同意できそうであれば「A」と入力します。次にLet's encryptのキャンペーン等のメールを配信してもいいかの確認がされます。今回はNoということで「N」を入力します。ドメインがLet's encryptによって確認されるので少し待ちましょう。「Congratulations!」と出れば証明書の取得に成功しています。

```
[root@                    ]# certbot certonly --webroot -w /usr/share/nginx/html -d dev.siroisakana.com
Saving debug log to /var/log/letsencrypt/letsencrypt.log
Plugins selected: Authenticator webroot, Installer None
Enter email address (used for urgent renewal and security notices) (Enter 'c' to
cancel):
Starting new HTTPS connection (1): acme-v02.api.letsencrypt.org

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Please read the Terms of Service at
https://letsencrypt.org/documents/LE-SA-v1.2-November-15-2017.pdf. You must
agree in order to register with the ACME server at
https://acme-v02.api.letsencrypt.org/directory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(A)gree/(C)ancel: A

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Would you be willing to share your email address with the Electronic Frontier
Foundation, a founding partner of the Let's Encrypt project and the non-profit
organization that develops Certbot? We'd like to send you email about our work
encrypting the web, EFF news, campaigns, and ways to support digital freedom.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(Y)es/(N)o: N
Obtaining a new certificate
Performing the following challenges:
http-01 challenge for dev.siroisakana.com
Using the webroot path /usr/share/nginx/html for all unmatched domains.
Waiting for verification...
Cleaning up challenges

IMPORTANT NOTES:
 - Congratulations! Your certificate and chain have been saved at:
   /etc/letsencrypt/live/dev.siroisakana.com/fullchain.pem
   Your key file has been saved at:
   /etc/letsencrypt/live/dev.siroisakana.com/privkey.pem
   Your cert will expire on 2020-02-13. To obtain a new or tweaked
   version of this certificate in the future, simply run certbot
   again. To non-interactively renew *all* of your certificates, run
   "certbot renew"
 - Your account credentials have been saved in your Certbot
   configuration directory at /etc/letsencrypt. You should make a
   secure backup of this folder now. This configuration directory will
   also contain certificates and private keys obtained by Certbot so
   making regular backups of this folder is ideal.
 - If you like Certbot, please consider supporting our work by:

   Donating to ISRG / Let's Encrypt:   https://letsencrypt.org/donate
   Donating to EFF:                    https://eff.org/donate-le
```

証明書が取得できたらNginxの設定を変更します。「vi /etc/nginx/conf.d/default.conf」を実行します。ほぼすべての設定を変えることになるので入力モードにしたら一度すべての内容を消してしまいましょう。その上で以下の内容を記述します。

/etc/nginx/conf.d/default.conf

```
server {
    listen       80;
    server_name  dev.siroisakana.com;

    location / {
        return 301 https://$host$request_uri;
    }
}
```

```
server {
    listen       443 ssl http2;
    server_name  dev.siroisakana.com;

    ssl_stapling            on;
    ssl_stapling_verify     on;
    ssl_certificate         /etc/letsencrypt/live/dev.siroisakana.com/fullchain.pem;
    ssl_certificate_key     /etc/letsencrypt/live/dev.siroisakana.com/privkey.pem;
    ssl_trusted_certificate /etc/letsencrypt/live/dev.siroisakana.com/chain.pem;

    gzip            on;
    gzip_types      text/css text/javascript application/x-javascript application/javascript application/jso
n;
    gzip_min_length  1k;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }
}
```

設定の内容は以下のようになっています。

・「dev.siroisakana.com」というFQDNで応答する
・HTTPによる通信はすべてHTTPSに転送する
・HTTPSを有効化する
・可能であればHTTP/2を利用する
・OCSPを有効化する
・送信するファイルが1k以上で指定のファイルタイプであればgzipを有効化する
・/user/share/nginx/htmlをドキュメントルートとする

記述が終わったら上書きして保存し「nginx -s reload」を実行しましょう。nginxの設定が更新されます。次に、https通信ができるようにファイアーウォールの設定を変更しましょう。httpsもhttp同様に設定ファイルが予め用意されているので「firewall-cmd --permanent --add-service=https」を実行し、「firewall-cmd --reload」も実行して設定を反映しましょう。

設定が完了したら「https://dev.siroisakana.com/」にアクセスしてみましょう。httpではなくhttpsです。「Welcome to nginx!」と表示されアドレスバー左の表示が鍵マークになっていればhttps化に成功しています（Google Chromeの場合）。

Let's encryptから取得した証明書は3ヶ月で失効してしまいます。3ヶ月ごとに更新手続きをするのは手間になってしまうので、自動更新を設定しましょう。自動更新にはcrontabというコマンドを利用します。crontabはスケジュールした時間になると予め設定しておいたコマンドを実行してくれます。では実際にcrontabの自動更新設定を行いましょう。「crontab -e」を実行します。テキストエディタが立ち上がるので、viコマンドの時と同様に編集します。以下の内容を記入してください。

```
0 4 * * * certbot renew --webroot-path /usr/share/nginx/html --post-hook "systemctl reload nginx"
```

編集が完了したら保存して終了してください。「crontab: installing new crontab」と表示されていればcrontabが設定できています。この設定では毎日午前4:00に自動更新処理が行われます。とはいえ毎日証明書が更新されるわけではなく、certbotではデフォルトで有効期限が残り30日未満の証明書のみ更新されるようになっているため残り日数が30日未満の場合のみ証明書が更新されます。

2.6.8 httpでのnginxの設定

ここではhttps化しない場合のnginxの設定について記述しています。すでに『https化(SSL/TLSの設定)』を行っている場合はこの項目は飛ばしてください。また、設定したドメインがまだ反映されていない場合はこの項目は後回しにしてください。『Node.js』の『ノンブロッキングIO』のあたりまでは進めます。なおこの項目はrootユーザーで行うのでrootではない場合はrootでログインしてください。

さて、httpでの通信を効率化するためにnginxの設定を変更しましょう。「vi /etc/nginx/conf.d/default.conf」を実行します。ほぼすべての設定を変えることになるので入力モードにしたら一度すべての内容を消してしまいましょう。その上で以下の内容を記述します。

```
/etc/nginx/conf.d/default.conf
server {
    listen       80;
    server_name  dev.siroisakana.com;

    gzip            on;
    gzip_types      text/css text/javascript application/x-javascript application/javascript application/jso
n;
    gzip_min_length  1k;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }
}
```

設定の内容は以下のようになっています。

- 「dev.siroisakana.com」というFQDNで応答する
- 送信するファイルが1k以上で指定のファイルタイプであればgzipを有効化する

・/user/share/nginx/htmlをドキュメントルートとする

保存して終了したら「nginx -s reload」を実行しましょう。nginxの設定が更新されます。

2.7 手順書

ここではVPSサーバー側の環境を整えるための手順をひとまとめにしてあります。何らかの事情で初期化をしたり、2つ目以降のサーバーの環境を整えたりする場合に参考にしてください。そうでない方はこの項目は飛ばして構いません。なお、解説した手順とは若干順番などが異なる場合がありますが問題ありません。なお、ドメインはすでに取得しておりネームサーバーの設定も済んでいるものとして進めます。また、読み替えるべき部分は以下のように背景色を変えてあります。

teiki … 使用するユーザー名
16815 … SSHで使用するポート番号
admin … MongoDBの管理ユーザー名
q%sv|N#IE99i … MongoDBの管理ユーザーのパスワード
34189 … MongoDBで使用するポート番号
siroisakana.com … 使用するドメイン名

rootでログイン
「yum update -y」を実行
「yum -y install epel-release」を実行
「curl -sL https://rpm.nodesource.com/setup_12.x | bash -」を実行
「vi /etc/yum.repos.d/nginx.repo」を実行、以下の内容を記述して保存

```
[nginx-stable]
name=nginx stable repo
baseurl=http://nginx.org/packages/centos/$releasever/$basearch/
gpgcheck=1
enabled=1
gpgkey=https://nginx.org/keys/nginx_signing.key
```

「vi /etc/yum.repos.d/mongodb-org-4.2.repo」を実行、以下の内容を記述して保存

```
[mongodb-org-4.2]
name=MongoDB Repository
baseurl=https://repo.mongodb.org/yum/redhat/$releasever/mongodb-org/4.2/x86_64/
gpgcheck=1
enabled=1
gpgkey=https://www.mongodb.org/static/pgp/server-4.2.asc
```

「yum -y install redis nodejs nginx mongodb-org」を実行
「systemctl enable redis」を実行

「systemctl enable nginx」を実行
「systemctl enable mongod」を実行
「reboot」を実行
rootで再度ログイン
Tera Termの上部メニューの「設定」から「SSH鍵生成」を選択
鍵の種類「RSA」ビット数「2048」を選択し「生成」をクリック
鍵のパスフレーズ、パスフレーズの確認、コメントを全て空にする
「公開鍵の保存」「秘密鍵の保存」でそれぞれ保存
コンソールに戻り「useradd teiki」を実行
「cd /home/teiki/」を実行
「mkdir .ssh」を実行
Tera Termの上部メニューの「ファイル」から「SSH SCP」を選択
上側のFrom横の「...」で公開鍵ファイルを選択
上側のToに「/home/teiki/.ssh/authorized_keys」を入力、「Send」をクリック
コンソールに戻り「chmod -R 700 .ssh」を実行
「chown -R teiki:teiki .ssh」を実行
「vi /etc/ssh/sshd_config」を実行、以下の表に従って書き換え保存

変更前	変更後
#Port 22	Port 16815
PermitRootLogin yes	PermitRootLogin no
PasswordAuthentication yes	PasswordAuthentication no
#ClientAliveInterval 0	ClientAliveInterval 60
#ClientAliveCountMax 3	ClientAliveCountMax 3

「firewall-cmd --permanent --remove-service=ssh」を実行
「cp /usr/lib/firewalld/services/ssh.xml /etc/firewalld/services/ssh-16815.xml」を実行
「vi /etc/firewalld/services/ssh-16815.xml」を実行
「<port protocol="tcp" port="22"/>」を「<port protocol="tcp" port="16815"/>」に書き換え保存
「mongo」を実行
「db.createUser({user:"admin", pwd:"q%sv|N#IE99i", roles:[{ role:"userAdminAnyDatabase", db:"admin" }]})」を実行
Ctrl+Cを入力
「vi /etc/mongod.conf」を実行
「#security:」と書かれている次の行の場所に以下を追記

```
security:
  authorization: enabled
```

「net:」と書かれているところを以下のように書き換え保存

```
net:
  port: 34189
  bindIp: 0.0.0.0
```

「systemctl restart mongod」を実行
「firewall-cmd --permanent --add-service=ssh-16815」を実行
「firewall-cmd --zone=public --add-port=34189/tcp --permanent」を実行
「firewall-cmd --permanent --add-service=http」を実行
「firewall-cmd --reload」を実行
「systemctl restart sshd」を実行
https化しない場合は「httpsにしない場合」に、https化する場合は「httpsにする場合」に進む

httpsにしない場合

「vi /etc/nginx/conf.d/default.conf」を実行、以下の内容を記述して保存

/etc/nginx/conf.d/default.conf

```
server {
    listen       80;
    server_name  dev.siroisakana.com;

    gzip            on;
    gzip_types      text/css text/javascript application/x-javascript application/javascript application/jso
n;
    gzip_min_length  1k;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }
}
```

「nginx -s reload」を実行

httpsにする場合

「yum -y install certbot」を実行
「certbot certonly --webroot -w /usr/share/nginx/html -d dev.siroisakana.com」を実行

メールアドレスを入力
規約に同意する場合「A」を入力
メール配信の確認に「N」を入力
「vi /etc/nginx/conf.d/default.conf」を実行、以下の内容を記述して保存

/etc/nginx/conf.d/default.conf

```
server {
    listen       80;
    server_name  dev.siroisakana.com;

    location / {
        return 301 https://$host$request_uri;
    }
}

server {
    listen       443 ssl http2;
    server_name  dev.siroisakana.com;

    ssl_stapling            on;
    ssl_stapling_verify     on;
    ssl_certificate         /etc/letsencrypt/live/dev.siroisakana.com/fullchain.pem;
    ssl_certificate_key     /etc/letsencrypt/live/dev.siroisakana.com/privkey.pem;
    ssl_trusted_certificate  /etc/letsencrypt/live/dev.siroisakana.com/chain.pem;

    gzip            on;
    gzip_types      text/css text/javascript application/x-javascript application/javascript application/jso
n;
    gzip_min_length  1k;

    location / {
        root    /usr/share/nginx/html;
        index   index.html index.htm;
    }
}
```

「nginx -s reload」を実行
「firewall-cmd --permanent --add-service=https」を実行
「firewall-cmd --reload」を実行
「crontab -e」を実行、以下の内容を記述して保存

```
0 4 * * * certbot renew --webroot-path /usr/share/nginx/html --post-hook "systemctl reload nginx"
```

Chapter 3

基礎学習編

定期・APゲームを開発するのに使用する
様々な技術の基礎部分を学習します。

3.1 HTML5

3.1.1 HTML5とは

HTML5は2014年10月28日に勧告されたHTMLの新しいバージョンです。HTML4に比べメディア、セマンティクスのサポートが強化されている他、デザインと構造の分離が進んでいます。マイナーバージョンも存在し、HTML 5.1は2016年11月1日に、HTML 5.2は2017年12月14日に勧告されています。なお、HTML 5.1以降はInternet Explorer 11では利用できません。

3.1.2 HTML4からの変更点

ここではHTML4から変わった点を紹介していきます。

DOCTYPE宣言

HTML5ではDOCTYPE宣言がかなりシンプルになりました。

```html
<!-- HTML4 -->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<!-- HTML5 -->
<!DOCTYPE html>
```

言語の指定

metaタグではなくhtmlタグの属性で指定するようになりました。

```html
<!-- HTML4 -->
<meta http-equiv="content-language" content="ja">

<!-- HTML5 -->
<html lang="ja">
```

文字コードの指定

こちらもシンプルになり、より短く記述できるようになりました。

```html
<!-- HTML4 -->
<meta http-equiv="content-type" content="text/html; charset=UTF-8">

<!-- HTML5 -->
<meta charset="UTF-8">
```

style、scriptタグ

　type="text/css"やtype="text/javascript"などタイプ指定を書いていましたが、それぞれデフォルト値になったためCSS、JavaScriptの場合は不要になりました。

HTML
```
<!-- HTML4 -->
<style type="text/css">
</style>
<script type="text/javascript">
</script>

<!-- HTML5 -->
<style>
</style>
<script>
</script>
```

まとめると以下のようになります。

HTML5
```
<!DOCTYPE html>
<html lang="ja">
  <head>
    <meta charset="UTF-8">
    <title>タイトル</title>
    <style>
      /* ここにCSSを記述 */
    </style>
  </head>
  <body>
    <!-- ここに内容を記述 -->
  </body>
  <script>
    /* ここにJavaScriptを記述 */
  </script>
</html>
```

3.1.3　セマンティックなタグ

　HTML4ではヘッダーもフッターもサイドバーも<div>で記述していましたがHTML5からは<header>、<footer>、<nav>など用途に応じた適切なタグを使うようになりました。HTMLで構造だけではなく意味も扱うようにしようという**セマンティック・ウェブ**の思想の下に実装されており、それらの関連技術をセマンティックHTMLなどと言ったりします。タグが持つ意味が違うだけで表示上は<div>と同じです。

　これらのタグを使うことで検索エンジンが意味を読み取りやすくなったり、視覚障害者の方などが読み上げ機能を使う際により適切な読み上げがなされるようになったります。また、ソースコードを読んだときにどこがヘッダーでどこがメニューなのかなどがわかりやすくなります。なるべく使うようにしましょう。HTMLの仕様を決定していた**W3C**(World Wide Web Consortium)は<div>を使うのは他のタグが適切でない場合のみにするよう勧告し

ています。
　よく使われるセマンティックなタグは以下のとおりです。

<main></main>
　ページのメイン要素であることを示すタグです。

<aside></aside>
　メインコンテンツとの関連性が薄い要素を示すタグです。補足やSNSボタンなどにはこのタグを使います。

<article></article>
　内容が単体で完結する場合に使用するセクションです。記事やコメントに対して使います。

<section></section>
　セクションに対して使用します。記事のうちの章・説・項などに使います。内容が単体で完結しているかどうかで<article>と使い分けます。

<nav></nav>
　ナビゲーションを示すタグです。メニューなどに対して使います。

<header></header>
　ヘッダーであることを示すタグです。

<footer></footer>
　フッターであることを示すタグです。

3.1.4　メディア関連のタグ

　HTML5からはHTMLタグで動画や音声を配置できるようになりました。また、<canvas>タグが実装されJavaScriptなどから動的に図形などを表示できるようになりました。

<audio></audio>
　オーディィオプレイヤーを配置できます。ブラウザによって対応している音声ファイルの形式が違うので注意が必要です。autoplay属性による自動再生なども用意されていますが、自動再生は制限されている場合が多いです。

<video></video>
　ビデオプレイヤーを配置できます。<audio>タグと同様に対応している動画形式はブラウザによって違い、自動再生もありますが制限されている場合が多いです。

<source>

<audio>や<video>要素などの中で子要素として使用します。このタグを使うことで複数のメディアファイルを指定することができます。複数のメディアファイルを指定することでブラウザがある形式に対応していない場合でも代替の形式を用意することができます。

<canvas>

JavaScriptなどから図形などを描くためのキャンバスを配置します。canvasを扱うためのJSフレームワークはいくつかあります。詳しく扱うことはしませんが、有名なものには**Pixi.js**や**CreateJS**などがあります。

3.1.5　新しく追加されたフォーム

HTML5から様々な種類のフォームが追加されました。その中から代表的なものを紹介します。

<input type="url">

URLを入力するフォームです。

<input type="email">

メールアドレスを入力するフォームです。

<input type="search">

検索バーに使うフォームです。

<input type="range">

数値の範囲などを直感的に入力できるフォームです。ブラウザにもよりますがスライドバーなどが表示されます。

また、属性も追加されました。

placeholder

入力時に表示されるヒントを指定します。

required

入力必須であることを示す属性です。同時にinputのtypeがemailなどの場合、submitの際にその入力内容が妥当であるかなどの検証も行われます。

pattern

入力内容の検証が行われる際、チェックに使用される正規表現を指定できます。

autocomplete/list

入力補完を行うための属性です。autocomplateにonを指定しlistに<datalist>タグで候補を指定することで簡単に入力補完機能が実装できます。

3.1.6　HTML5.1以降で追加されたタグと属性

srcset属性

（デバイスピクセルについては「おまけ」の「デバイスピクセル」に簡単な解説があります。）

pictureタグ

例えばPCは基本的に横長の画面ですがスマートフォンは縦長です。このような場合全く同じ画像を表示することが適切ではない場合があります。そういうときに使うのがpictureタグになります。srcset属性はデバイスピクセルの違いによって解像度を分ける場合に、pictureタグは画面サイズによって表示する画像を別のものにする場合に使うようにします。

3.1.7　viewport

画面の仮想的なサイズおよび拡大率を設定するためのmetaタグです。例えば以下のように設定したとします。

```
<meta name="viewport" content="width=360,initial-scale=1">
```

この例では画面の横幅が360pxとして設定され、初期拡大率は1（等倍）です。JavaScriptやCSSからも画面の横幅が360pxとして扱われます。また、width=device-widthとすることで仮想的なサイズとデバイスの実際の解像度を合わせせることができます。多くの場合、以下のように指定しておけば問題ないでしょう。

```
<meta name="viewport" content="width=device-width,initial-scale=1">
```

3.2 ES2015(ES6)

3.2.1　ES2015(ES6)とは

ES2015(ECMAScript 2015)とはJavaScriptの共通規格のことです。2015年にリリースされたためこのような名称となっています。各ブラウザなどはこの規格に従ってJavaScriptエンジンを実装しています。(共通規格に従っているとはいえ、ブラウザごとに仕様が異なる部分もあります。)当初はECMAScript 6th editionという名称だったため**ES6**という呼び方をされることが多く、本書においてもES6という呼称を用います。なお、これに対して従来のJavaScriptの記法を**ES5**と呼んだりします。

ここでは、ES6で新たに追加された記法や機能などを紹介していきます。ES6の記法はすでに開発が停止しているInternet Explorer 11では使えないので注意が必要です。(なお、ES6の記法をレガシーブラウザで使えるようにするための技術も存在します。)

3.2.2　const/let

const、**let**はvarに代わる新たな変数の宣言です。letは同スコープ内で変数の重複宣言ができない、という特徴があります。そのため誤って同じ変数を2回宣言してしまい、わけのわからないバグが生まれてしまったというような事態を防ぐことができます。constはそれに加えて変数に再代入ができない、という特徴を持ちます。他の言語を知っている方からすればすこし違和感があるかもしれませんが、constは再代入が不可能なだけでconst宣言されたオブジェクトのプロパティは変更可能です。constを利用することで意図しない再代入によるバグを防ぐことができます。実際に例を見てみましょう。

```
var hensuOne = 1;
var hensuOne = 2; // 同じ変数を2度宣言しているがエラーにはならない

let hensuTwo = 1;
hensuTwo = 2;     // letは再代入ができるためエラーにならない
let hensuTwo = 3; // 同名の変数を宣言しているためエラーになる

const hensuThree = 1;
hensuThree = 2;       // constは再代入が不可能なためエラーになる
const hensuThree = 3; // 同名の変数を宣言しているためエラーになる

const constObject = {
  propOne: 1,
  propTwo: 2
};

constObject.propOne = 3; // constObjectそのものに再代入しているわけではないためエラーにならない
constObject = 4;         // これは再代入のためエラーになる
```

ES6では原則として変数の宣言にはconstを使用し、再代入をしなければならない場合にのみletを使用します。

3.2.3　テンプレートリテラル

例えば変数の内容に従って「私は〇〇月〇〇日生まれで、〇〇歳です。」という文を表示したいとしましょう。今までは以下のように記述する必要がありました。

```
console.log('私は' + month + '月' + date + '日生まれで、' + age + '歳です。');
```

もちろんこれでも記述することはできるのですが少し煩雑で見づらいです。そこで、より簡潔かつ明瞭な記述方法として生まれたのが**テンプレートリテラル**です。テンプレートリテラルは`（バッククォート）を用いて記述します。テンプレートリテラル内では${～～}と記述することで連結表示ができます。上の例文はテンプレートリテラルでは以下のように記述することができます。

```
console.log(`私は${month}月${date}日生まれで、${age}歳です。`);
```

3.2.4　デフォルト引数

ES6では関数の引数にデフォルト値を指定することができるようになりました。デフォルト値は引数の後に=（イコール）で指定します。実際に例を見てみましょう。

```
const saySomething = function (str = 'ほげほげ') {
  console.log(str);
  return;
};

saySomething('ふーばー'); // ふーばー
saySomething(); // ほげほげ
```

saySomething関数は受け取ったstrをそのままconsole.logで表示するだけの関数です。ただし、strのデフォルト値は'ほげほげ'なのでなにも値を渡していない場合strは'ほげほげ'になります。1回目のsaySomething('ふーばー')ではstrに'ふーばー'が代入され、「ふーばー」と表示されます。2回めのsaySomething()ではstrが受け取る値が何もないため、strは'ほげほげ'になります。そのため、「ほげほげ」と表示されます。

なお、undefinedを渡した場合にもデフォルト引数は適用されます。以下の例では「ほげほげ」が表示されるということです。

```
const foobar = undefined;
saySomething(foobar);
```

3.2.5　クラス

クラスはオブジェクトの設計図のようなもので、同じようなオブジェクトを複数作りたい場合に役立ちます。クラス

では初期値の宣言や初期化に必要な値の受け取りにはコンストラクタを使用します。コンストラクタとはクラスからオブジェクトが生成される際に最初に呼び出されるメソッド(関数)のことです。クラスは以下のように記述します。以下の例では名前・HPを受け取って名前・最大HP・現在HPを初期化している他、名前と現在HPを表示するメソッドsayを定義しています。

```
class Unit {
  constructor(name, hp) {
    this.name = name; // 名前
    this.maxHp = hp; // 最大HP
    this.currentHp = hp; // 現在HP
  }

  say(){
    console.log(`私の名前は${this.name}。現在HPは${this.currentHp}。`);
  }
}
```

クラスからオブジェクトを生成するときはnew ClassName()というように行います。この処理のことを**インスタンス化**といい、このようにして生成されたオブジェクトのことを**インスタンス**と呼びます。インスタンス化の際は()の中でコンストラクタに渡す値を指定します。上の例のUnitクラスであれば、以下のようにインスタンスを生成し使用できます。

```
const taro = new Unit('太郎', 100); //名前が太郎でHPが100のユニットを生成しtaroに代入
const hanako = new Unit('花子', 200); //名前が花子でHPが200のユニットを生成しhanakoに代入

taro.say(); // 私の名前は太郎。現在HPは100。
hanako.say(); // 私の名前は花子。現在HPは200。
```

また、クラスの機能を受け継いで新しいクラスを作ることができます。これを**継承**といい、extendsの後に継承元クラスを書くことで継承することができます。クラスの継承では継承元のクラスを**親クラス**、それに対して継承した側のクラスのことを**子クラス**と呼びます。また、親の親の……クラスのことを**先祖クラス**といったり、子の子の……クラスのことを**孫クラス**や**子孫クラス**と呼んだりします。子クラスでは親クラスのメソッドをsuper.(メソッド名)で呼び出すことができます。コンストラクタの場合メソッド名は不要です。ここでは例としてUnitクラスにMPとcastメソッドを追加しsayでの発言内容を追加したCasterUnitクラスを作ってみます。

```
class CasterUnit extends Unit {
  constructor(name, hp, mp) {
    super(name, hp); // 親クラスのコンストラクタを呼び出す
    this.maxMp = mp; // 最大MP
    this.currentMp = mp; // 現在MP
  }

  say(){
```

```
    super.say(); // 親クラスのsayメソッドを呼び出す
    console.log(`魔法も使えるよ。現在MPは${this.currentMp}。`);
  }

  cast() {
    console.log(`${this.name}は魔法を唱えた！`);
  }
}

const smith = new CasterUnit('スミス', 50, 200);
smith.say(); // 私の名前はスミス。現在HPは50。魔法も使えるよ。現在MPは200。
smith.cast(); // スミスは魔法を唱えた！
```

オブジェクトはinstanceofによってそのクラスのオブジェクトかどうかを判定することができます。これは先祖クラスに対しても有効です。

```
const checkIsUnit = function(obj) {
  if (obj instanceof Unit) {
    console.log('これはUnitです。');
  } else {
    console.log('これはUnitではありません。');
  }

  if (obj instanceof CasterUnit) {
    console.log('これはCasterUnitです。');
  } else {
    console.log('これはCasterUnitではありません。');
  }
}

checkIsUnit(taro);
// これはUnitです。
// これはCasterUnitではありません。

checkIsUnit(smith);
// これはUnitです。
// これはCasterUnitです。
```

クラスでは外部からアクセスする必要のない変数やメソッドの名前には先頭に_(アンダースコア)をつけるのが一般的です。外部からは先頭にアンダースコアのついた変数やメソッドにアクセスしないようにすることで意図しない再代入などを防ぐことができます。以下のように記述します。

```
class HimitsuUnit {
  constructor(name) {
    this._name = name;
  }

  _say() {
    console.log(`私の名前は${this._name}。`)
```

```
  }
}

const himitsu = new HimitsuUnit('ヒミツ');

console.log(himitsu._name); // これはやらないようにする
himitsu._say(); // これもやらないようにする
```

このように外部からアクセスできない変数やメソッドのことを**プライベート変数**や**プライベートメソッド**、**プライベートフィールド**などと言ったりします。勘のいい方は表現などから察しているかもしれませんが、アンダースコアによるネーミングはJavaScriptではプライベートフィールドを実現できないのを運用によってカバーしているだけであり、実際にはやろうと思えば外部からアクセスすることができてしまいます。そのためアンダースコア表現は実際にはプライベートフィールドではありません。

プライベートフィールドを正しく実現する機能もあるのですが、最近できたばかりでまだ正式なものではなく(2019年11月1日現在仕様は完成しているものの実装やフィードバックを求めている状態)利用できるブラウザや処理系もかなり限られているため本書では取り扱いません。

クラスにはまだまだ機能があります。次は**ゲッター**と**セッター**について学びます。ゲッター/セッターとはプロパティのような振る舞いをする関数のことです。なんのことだかよくわからないと思うので具体的な例を上げてみましょう。以下はゲッターとセッターを利用したクラスの例です。

```
class Kazu {
  constructor(value) {
    this.value = value;
  }

  get number() {
    return this.value;
  }

  set number(value) {
    this.value = value;
  }

  get five() {
    return 5;
  }
}

const kazu = new Kazu(10);

console.log(kazu.number); // 10
kazu.number = 20;
console.log(kazu.number); // 20
console.log(kazu.five);   // 5
```

kazuはvalue以外の値を持っていないはずですがkazu.numberやkazu.fiveであたかも指定の値があるかの

ようにアクセスできています。ゲッターは指定の値を取得するときに呼び出され、returnで値を返すことであたかも指定の値があるかのように振る舞わせることができる関数です。メソッド名の前にgetというキーワードをつけることで宣言します。それに対しセッターは指定の値を設定するときに呼び出される関数で、代入しようとした内容が渡されて処理を行うことが出来ます。こちらはsetというキーワードをつけます。

　ゲッターとセッターは関数なのでいろいろと自由なことができます。例えば以下のようにゲッターを利用してある値を返すときに補正をかけて値を返すことが出来ます。

```
class Kazu {
  constructor(value) {
    this.value = value;
  }

  get doubledNumber() {
    return this.value * 2;
  }
}

const kazu = new Kazu(10);
console.log(kazu.doubledNumber); // 20
```

　この性質は定期・APゲームの戦闘エンジンを組むときに例えばある状態異常中はステータス値に補正をかけたい、といった場合などに役立つでしょう。

　最後は**静的メソッド**です。基本的にクラスを利用するときはnewでインスタンスを作ってから利用することになるのですが、静的メソッドについてはインスタンス化する前に呼び出すことができます。逆にインスタンス化した後は呼び出すことはできません。静的メソッドはメソッド名の前にstaticというキーワードをつけることで宣言します。以下のような感じになります。

```
class Seihoukei {
  static calcMenseki(ippen) {
    return ippen * ippen;
  }
}

// クラスから呼び出す
console.log(Seihoukei.calcMenseki(10)); // 100

// インスタンスから呼び出す
const seihoukei = new Seihoukei();
console.log(seihoukei.calcMenseki(10)); // エラー。静的メソッドはインスタンスからは呼び出せない。
```

　静的メソッドはインスタンスではなくクラス自体に持たせたい情報などを管理するときに有用です。定期・APゲームの開発であれば例えばスキルの習得条件を定義するときに静的メソッドで定義しておくとスキルを使うわけではないのにわざわざスキルをインスタンス化しなくても習得できるか判断できるようになるので便利でしょう。

クラスは定期・APゲームの戦闘エンジンを組む際に多用することになるので本書で紹介したクラスの使い方は頭に入れておきましょう。

3.2.6　アロー関数

従来、関数を宣言する場合以下のように書いていました。

```
const doubleNumber = function(num) {
  return num * 2;
};
```

アロー関数を使うとこれを以下のように書くことができます。

```
const doubleNumber = (num) => {
  return num * 2;
};
```

また、アロー関数は{}の中身が1文だけであればreturnと{}を省略することができます。引数についても1つだけであれば()を省略できます。つまり、上の式は以下のようにも書くことができます。

```
const doubleNumber = num => num * 2;
```

これだけではアロー関数はfunction()を縮めただけの書き方に見えますが、アロー関数はそれ以外に重要な機能があります。それは**アロー関数にはthisを束縛する性質がある**ということです。より分かりやすく言うと、アロー関数内のthisはそのアロー関数が宣言された場所のthisを参照します。実際にどういうことなのか見ていきましょう。

```
class OuterClass {
  constructor() {
    this.message = 'thisはouterClassを参照しています。';

    this.normalFunction = function() {
      console.log(this.message);
    };
    this.arrowFunction = () => {
      console.log(this.message);
    };

    this.innerObject = {
      message: 'thisはinnerObjectを参照しています。',
      normalFunction: this.normalFunction,
      arrowFunction: this.arrowFunction
```

```
    };
  }
};

const outerClass = new OuterClass();
outerClass.innerObject.normalFunction(); //thisはinnerObjectを参照しています。
outerClass.innerObject.arrowFunction(); //thisはouterClassを参照しています。
```

実際にコードの流れを見てみましょう。まずOuterClassというクラスを宣言しています。そしてOuterClassのコンストラクタ内でmessageとnormalFunction、arrowFunctionをそれぞれ宣言しています。次にinnerObjectを宣言しています。innerObjectにもmessageが存在しており、normalFunctionとarrowFunctionはOuterClassのものが代入されています。そしてコードの最後でinnerObject内の関数をそれぞれ実行しています。

結果を見てみると、normalFunctionでは「thisはinnerObjectを参照しています。」という表示がなされます。innerObject内ではthisはinnerObjectを指すのでnormalFunctionではinnerObjectのmessageの内容が表示されています。arrowFunctionの方も見てみましょう。arrowFunctionでは「thisはouterClassを参照しています。」という表示がなされます。アロー関数のthisは宣言された場所のthisを参照するので、宣言された場所であるOuterClassのmessageを参照しているというわけです。

function()とアロー関数を使い分けることでthisが示す内容をある程度コントロールすることができます。この性質は後々利用することになるので頭の片隅に置いておいてください。よくわからない場合、thisの参照先を宣言した所のthisから変えたい場合はfunction()を使い、変えたくない場合はアロー関数を使うという理解でも支障ないかと思います。

3.2.7　Array

ArrayはJavaScriptにおける配列を扱うためのオブジェクトです。配列に入った値の参照方法など、Arrayに関する基本的な使い方については最低限知っているものとして解説は行いません。ここではArrayの知っておくと便利なメソッドについて紹介します。なおこれらについてはES5の時点ですでに存在していたものですが、よく使われる割に入門書では触れられないことも多いためここで解説しておきます。

array.map(関数)

mapは元の配列arrayから新たな配列を作成するためのメソッドです。引数に指定する関数では配列に入ったそれぞれの値をインデックス0から順番に受け取ることができ、その戻り値が新たな配列となります。ただし言葉で説明しても分かりづらいと思われるので、実際の例を見ながら学んでいきましょう。

以下は数値が入った元の配列oldArrayからそれぞれの値を2倍にした新しい配列newArrayを生成する例です。mapに「数値を受け取ってその2倍の値を返す」関数を設定してあります。この関数がoldArrayのそれぞれの要素に対して呼び出されることでnewArrayが生成されているというわけです。

```
const oldArray = [1, 3, 5];

const newArray = oldArray.map(function(num) {
  return num * 2;
});
```

```
console.log(newArray);
// [2, 6, 10];
```

オブジェクトの配列からある値だけを取り出した配列を作る際にも使えます。例えば、キャラクターデータの配列からENoだけの配列を作りたい場合以下のようなコードで実現できます。

```
const oldArray = [{
  eno:  1,
  name: '一郎'
}, {
  eno:  2,
  name: '二郎'

}, {
  eno:  4,
  name: '四郎'
}];

const newArray = oldArray.map(function(character) {
  return character.eno;
});

console.log(newArray);
// [1, 2, 4];
```

アロー関数の略記と組み合わせることでさらに処理を簡潔に記述することができます。以下は上記2つの処理をそれぞれアロー関数で略記したものです。このようにひと目でわかりやすく明解になるためこのような場面では積極的にアロー関数を使うと良いでしょう。

```
const newArray = oldArray.map(num => num * 2);
const newArray = oldArray.map(character => character.eno);
```

array.filter(関数)

filterは元の配列arrayから関数に従って指定の要素のみを抽出した新たな配列を作成するためのメソッドです。引数に指定する関数で抽出したい要素でtrueを返すことによって、その要素を取り出す対象に指定できます。例を見てみましょう。以下はoldArrayから「10未満の値のみを取り出す」操作を行っています。

```
const oldArray = [1, 11, 8, 14, 10, 3];
const newArray = oldArray.filter(num => num < 10);

console.log(newArray);
// [1, 8, 3]
```

array.reduce(関数)

reduceは元の配列arrayから新たな値を作成するためのメソッドです。mapでは関数の戻り値が新たな配列の要素になっていましたが、reduceでは関数の戻り値は次の処理に渡されるようになっており、最終的な関数の戻り値が結果の値となります。例を見てみましょう。以下はarrayの要素の合計値を計算する処理です。

```
const array  = [1, 2, 3, 4];
const result = array.reduce((acc, value) => acc + value);

console.log(result);
// 10
```

処理を追っていきましょう。accが前回の戻り値、valueが配列から受け取る値です。まずreduceの関数には配列の0、1番目の値であるacc=1、value=2が渡されacc+valueの結果である3が戻り値となります。次に再びreduceの関数が呼ばれ、前回の戻り値である3をaccとして受け取り、配列の2番目の値である3を受け取ります。足した結果は6ですからまた次の処理にこの値が渡され、最終的に10という合計値が得られるというわけです。

応用すれば最大値や最小値も得ることができます。Math.maxとMath.minは渡した値の中から最大値と最小値をそれぞれ得るメソッドです。これをreduceで配列の要素に対して順次に呼び出すことによって配列の中の最大値や最小値を得ることができます。

```
const array  = [8, 13, 5, 21, 4];
const max = array.reduce((acc, value) => Math.max(acc, value));
const min = array.reduce((acc, value) => Math.min(acc, value));

console.log(`max:${max} min:${min}`);
// max:21 min:4
```

reduceは少し複雑ですので本書では使っていませんが、適切に使うことができるとコーディングがぐっと楽になるため覚えておいて損はないでしょう。なお、これは余談ですが配列の右から左(最後から最初)の方向に処理するreduceRightというメソッドも存在します。使い方はreduceと一緒です。

array.sort(関数)

sortは配列を関数の基準に従って並べ替えるための処理です。sortの関数では元の配列arrayの要素から一つずつ要素aと要素bを受け取り、要素aが要素bより先に来る場合は0未満、要素aが要素bより後に来る場合は0

より大きい値、要素a、bを入れ替える必要がない場合は0を返します。sortメソッドはその結果を元にarrayを並び替えます。

以下は配列の入った数値を昇順でソートする例です。a - bを戻り値とすることでaのほうが小さい場合戻り値が0未満となり、よりaが左に配置されることとなるので、これで昇順のソートが実現可能です。

```
const array = [8, 13, 5, 21, 4];

array.sort((a, b) => a - b);

console.log(array);
// [4, 5, 8, 13, 21]
```

降順にするには逆にb - aとします。これによりaのほうが小さい場合戻り値が0より大きくなるのでよりaが右に配置され降順ソートとなります。

```
const array = [8, 13, 5, 21, 4];

array.sort((a, b) => b - a);

console.log(array);
// [21, 13, 8, 5, 4]
```

sortで気をつけるべき点はこの処理は並び替えられた新たな配列を得るものではなく、元の配列arrayの順序を変更する処理だという点です。コードを見ていただければわかるのですが、実行前と後でarrayの値の順序が変わっています。

このように操作対象の配列を書き換えるようなメソッドのことを**破壊的メソッド**(破壊的処理)と呼び、mapやfilterのような操作対象を書き換えず新たな配列を生成するようなメソッドのことを**非破壊的メソッド**(非破壊的処理)と呼びます。配列のメソッドは破壊的メソッドと非破壊的メソッドの2つがあり、今使っているメソッドはどちらなのかを意識しないと思わぬバグを生むことがあるので注意しましょう。

3.2.8　分割代入

分割代入は配列の値やオブジェクトのプロパティを一つ一つの変数に取り出す構文です。まずは配列の例から見ていきましょう。以下の例では元の配列arrayから要素一つ一つをitem0, 1, 2に取り出しています。

```
const array = [1, 4, 9];

const [item0, item1, item2] = array;

console.log(`item0:${item0} item1:${item1} item2:${item2}`);
// item0:1 item1:4 item2:9
```

配列の中から一部のみを取り出したり、「...」を使って余剰の部分を別の配列として取り出すことも可能です。以下のように記述します。

```
const array = [1, 3, 5, 7, 9];

const [a0, a1, a2] = array;
console.log(`a0:${a0} a1:${a1} a2:${a2}`);
// a0:1 a1:3 a2:5

const [b0, , b2] = array;
console.log(`b0:${b0} b2:${b2}`);
// b0:1 b2:5

const [c0, c1, ...other] = array;
console.log(`c0:${c0} c1:${c1} other:${other}`);
// c0:1 c1:3 other:5,7,9
```

また、分割代入を利用すれば値を簡単に入れ替えることも可能です。

```
let a = 1;
let b = 2;

[a, b] = [b, a];

console.log(`a:${a} b:${b}`);
// a:2 b:1
```

次はオブジェクトの分割代入を見ていきましょう。オブジェクトの分割代入ではプロパティが同名のものを簡単に取り出すことが可能で、以下のように{}を使って記述します。以下の例ではcharacterからenoとnicknameを分割代入で取り出しています。

```
const character = {
  eno:      1,
  nickname: '太郎',
  fullname: '田中太郎'
};

const {eno, nickname} = character;
console.log(`eno:${eno} nickname:${nickname}`);
// eno:1 nickname:太郎
```

プロパティ名とは違う変数名を割り当てたい場合、以下のように : (コロン)を使って割り当てたい変数名を指定します。

```
const character = {
  eno:      1,
  nickname: '太郎',
  fullname: '田中太郎'
};

const {eno: bangou, nickname: namae} = character;
console.log(`bangou:${bangou} namae:${namae}`);
// bangou:1 namae:太郎
```

また、分割代入は関数の引数の指定でも利用することができます。以下のshowCharacterNickname関数では分割代入を利用し、受け取ったオブジェクトからnicknameのみを取り出して表示しています。

```
const character = {
  eno:      1,
  nickname: '太郎',
  fullname: '田中太郎'
};

const showCharacterNickname = function({nickname}) {
  console.log(`私の名前は${nickname}です。`);
}

showCharacterNickname(character);
// 私の名前は太郎です。
```

3.2.9 Promise

プログラムにとってインターネットから情報を取得する、ファイルを読み込む、データベースにアクセスする、といった処理は非常に時間のかかる処理で、これらの処理をいちいち待っているとプログラム全体の処理速度が低下してしまいます。JavaScriptではそれに対処するため、ある処理が完了したら関数を呼び出すといった方式を取っていました。このようにプログラム全体の流れとは別に実行される処理のことを**非同期処理**といい、非同期処理の方式のうちこのような方式のことを**コールバック**と呼びます。例えば「アクセスがあったらユーザー情報をデータベースから取得してその値を返す」だけの処理を書くとしましょう(以下例1)。例1をコールバックで記述する場合以下のようになるでしょう(関数名などは例示のための架空のものです)。

```
server.get((req, res) => {
  UserDatabase.find(req.user.id, (err, user) => { // ユーザー情報をデータベースから検索
    if (err) {
      return res.status(500).send(); // エラーが起きたら500を返す
    } else if (!user) {
      return res.status(404).send(); // ユーザーが見つからなかったら404を返す
    } else {
      return res.json(user); // ユーザー情報を返す
```

```
    }
  });
});
```

　極めて単純な処理であれば上記のようなコールバックでも問題ありませんでした。では、次のような場合はどうでしょうか。「ユーザー情報をデータベースから取得し、そのユーザーがブロックしていない/ブロックされていない部屋主のトークルームを取得、さらにそのトークルームのそれぞれの発言数を取得しトークルームデータと合わせて返却する」というようなトークルームリストを作るための処理(以下例2)をコールバックで書くと以下のようになります。

```
server.get((req, res) => {
  UserDatabase.find(req.user.id, (err, user) => {
    if (err) {
      return res.status(500).send();
    } else if (!user) {
      return res.status(404).send();
    } else {
      const dismissList = [].concat(user.blockList).concat(user.blockedList);
      RoomDatabase.find({roommaster: {not: dismissList}}, (err, roomList) => {
        if (err) {
          return res.status(500).send();
        } else {
          LogDatabase.counts(roomList, (err, logCounts) => {
            if (err) {
              return res.status(500).send();
            } else {
              return {
                talkRooms: roomList,
                logCounts: logCounts
              };
            }
          });
        }
      });
    }
  });
});
```

　ネストの深さ({}の深さのこと)がとてつもない量になってしまいました。特に最後の}の連続などは非常にわかりづらく見づらいです。エラー処理がいちいち挟まなければならないのも見づらくなっている要因でしょう。このようなもののことを俗に**コールバック地獄**と呼びます。そこで生まれたのがPromiseという概念です。Promiseは以下のように記述します。

```
new Promise((resolve, reject) => {
    // ここに内容を記述
});
```

　Promiseではresolveとrejectを用いて処理を進めていきます。これはそれぞれ関数になっていて、処理が完了したらresolve()を呼び出し、何らかのエラーが発生した場合はreject()を呼び出します。例1のデータ取得部分をPromiseで書いてみましょう。

```
new Promise((resolve, reject) => {
  UserDatabase.find(req.user.id, (err, user) => {
    if (err) {
      reject(err);
    } else if (!user) {
      reject('User Not Found');
    } else {
      resolve(user);
    }
  });
});
```

　これだけではまだメリットを実感できないかもしれません。Promiseにはさらに重要な機能があり、thenという関数を使うことで処理をつなげることができます。thenを使う場合、resolveに値を渡すことで次のPromiseでその値を受け継ぐことができます。また、エラー処理をcatchでまとめることができます。resolve同様、rejectに値を渡すことによりcatchに値を受け渡すことができます。例2をPromiseで書いてみましょう。

```
server.get((req, res) => {
  new Promise((resolve, reject) => {
    UserDatabase.find(req.user.id, (err, user) => {
      if (err) {
        reject(err);
      } else if (!user) {
        reject('User Not Found');
      } else {
        resolve(user);
      }
    });
  })().then(user => {
    return new Promise((resolve, reject) => {
      const dismissList = [].concat(user.blockList).concat(user.blockedList);
      RoomDatabase.find({roommaster: {not: dismissList}}, (err, roomList) => {
        if (err) {
          reject(err);
        } else {
          resolve(roomList);
        }
      }
    });
  }).then(roomList => {
    return new Promise((resolve, reject) => {
      LogDatabase.counts(roomList, (err, logCounts) => {
        if (err) {
          reject(err);
        } else {
```

```
          resolve({
            talkRooms: roomList,
            logCounts: logCounts
          });
        }
      });
    });
  }).then(result => {
    res.json(result);
  }).catch(err => {
    if (err == 'User Not Found') {
      res.status(404).send();
    } else {
      res.status(500).send();
    }
  })
});
```

まだ煩雑ですが、少なくとも最初よりはかなりよくなりました。UserDatabase.findなどをPromiseにしたものがあれば例2をさらに簡単に書くことができます。

```
server.get((req, res) => {
  UserDatabase.promiseFind(req.user.id)
  .then(user => {
    const dismissList = [].concat(user.blockList).concat(user.blockedList);
    return RoomDatabase.promiseFind({roommaster: {not: dismissList}});
  }).then(roomList => {
    return new Promise((resolve, reject) => {
      LogDatabase.counts(roomList, (err, logCounts) => {
        if (err) {
          reject(err);
        } else {
          resolve(res.json({
            talkRooms: roomList,
            logCounts: logCounts
          }));
        }
      });
    });
  }).catch(err => {
    if (err == 'User Not Found') {
      res.status(404).send();
    } else {
      res.status(500).send();
    }
  })
});
```

だいぶすっきりしたのではないでしょうか。最初に比べるとかなり改善され、一応実用に足る構文と言うことができるでしょう。しかし、非同期ではない処理に比べるとまだまだ煩雑です。そこで生まれたのがasync/awaitです。

3.2.10 async/await

　async/awaitはES2017でリリースされたモダンな非同期処理の記法です。ES6の機能ではありませんが、本書ではこの記法を多用するためここで紹介します。asyncをつけた関数は非同期処理になります。また、async関数内でawaitを使うことでPromiseの処理が終わるのを待つことができます。それに加え、async/awaitを使うことでtry〜catchを使ってエラーハンドリングができるようになります。例2をasync/awaitで記述してみましょう。

```
server.get(async (req, res) => {
  try {
    const user = await UserDatabase.promiseFind(req.user.id);
    if (!user) {
      return res.status(404).send();
    }

    const dismissList = [].concat(user.blockList).concat(user.blockedList);

    const roomList = await RoomDatabase.promiseFind({roommaster: {not: dismissList}});
    const logCounts = await LogDatabase.promiseCounts(roomList);

    return res.json({
      talkRooms: roomList,
      logCounts: logCounts
    });
  } catch(err) {
    return res.status(500).send();
  }
});
```

　見通しもよく簡潔で分かりやすいコードになったのではないでしょうか。このように非常に分かりやすく非同期処理を記述できるため、このような場合はasync/awaitを利用するのが一般的です。非同期処理を行うJavaScriptライブラリは大抵の場合Promiseを提供しており、本書において扱う内容においてはそれを使うだけのことがほとんどのため、Promiseそのものを書くことはほぼありません。しかし、async/awaitの根底にはPromiseがある、ということは覚えておいてください。

　なお、本書で取り扱うレベルにおいてはほぼ使わない知識なので詳しく取り扱うことはしませんが、asyncが実際に何をしているのかを軽く解説しておきます。asyncは内部的にはPromiseを生成しています。async関数は実はPromiseなのです。頭の片隅に入れておくと複数の非同期処理を同時に実行し、その全てが終わったら何か処理をする、というようなことをしたい場合に役立つでしょう。

3.3 CSS3とCSS関連技術

3.3.1 CSS3とは

CSS3(Cascading Style Sheets, level 3)とはCSSの新しいバージョンのことです。CSS3では様々な便利な機能が追加されています。その全てを紹介するのは難しいので、ここではCSS3で追加された機能のうち主なものを紹介していきます。なお、これらの内容を覚えきる必要は全くありません。どんな機能があるのかだけ把握しておき、使いたくなったときにまた調べるといった方法をおすすめします。

また紹介するCSS3の機能の一部については**CodePen**(HTML、CSS、JavaScriptなどを共有・表示できるサービス)上に作例を用意してあります。実際に開いて確認したり、書き換えたりしてみてください(書き換えても保存されて他の人が見られなくなったりといったことはありません)。

3.3.2 flexbox

flexboxはCSS3で追加された新しいレイアウトの指定方法およびそのためのプロパティです。他の要素や横幅に合わせて伸縮する要素や中央揃え、上揃え、下揃えなどを簡単に実現することができます。flexboxについては以下のサイトに非常に分かりやすい説明と例があるのでここで改めて解説はしません。flexboxは定期・APゲームを作る際に多用することになるので一度は目を通しておいてください。

日本語対応!CSS Flexboxのチートシートを作ったので配布します | Webクリエイターボックス
https://www.webcreatorbox.com/tech/css-flexbox-cheat-sheet

3.3.3 calc()

calc()は異なる単位同士の表現で演算したい場合などに使用します。例えば横幅を100%から50pxだけ引いた長さにしたい場合、「width: calc(100% - 50px);」とすることで実現できます。calcでは以下の演算子を用いて四則演算をすることができます。

+	加算
-	減算
*	乗算(引数のうち少なくとも1つは数値でなければならない)
/	除算(右側は0以外の数値でなければならない)

calc()内部で()やcalc()を入れ子にして使うこともできます。意味は算数で使う()と同じで計算順序の変更です。JavaScriptとは違い「width: calc(100%-50px);」のような表現は無効で、演算子の前後には空白を開ける必要があります(正確には空白が必要なのは+と-のみですが表現の統一のため*や/にも空白を開けることが推奨されています)。IE11にもcalc()が実装されていますがバグが多くIE11に対応する場合は注意が必要です。

作例:https://codepen.io/siroi_sakana/pen/QWWQNXZ

3.3.4 その他追加されたプロパティ・値

その他、CSS3で追加されたプロパティや値から主なものを紹介します。

opacity

要素の透明度を指定するプロパティです。透明度は0.0〜1.0の範囲で指定し0.0なら完全な透明、1.0なら完全な不透明です。

作例：https://codepen.io/siroi_sakana/pen/RwwQRNZ

rgb() / hsl()

RGBとHSLで色を指定できます。RGBは光の三原色による指定、HSLは色相・彩度・輝度による指定です。RGBの場合各色0〜255、HSLの場合色相は0〜360、彩度・輝度は0〜100%で指定しそれぞれ「rgb(128, 0, 255)」「hsl(290, 50%, 70%)」というようになります。

アルファチャンネル（透明度）の指定を追加した「rgba()」「hsla()」もあります。opacityと同様に0.0〜1.0で指定し0.0なら完全な透明、1.0なら完全な不透明です。「rgb(128, 0, 255, 0.3)」「hsl(290, 50%, 70%, 0.6)」というように指定します。

background-color: rgb(100, 0, 150);

background-color: hsl(180, 50%, 50%);

background-color: rgba(100, 0, 150, 0.2);

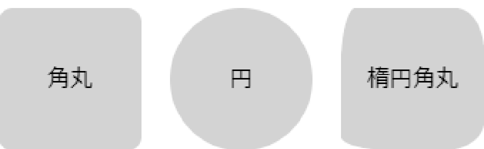
background-color: hsla(180, 50%, 50%, 0.5);

作例：https://codepen.io/siroi_sakana/pen/abbaGmg

border-radius

ボックス要素の角を丸めるプロパティです。「border-radius: 5px 10px 15px 20px;」というように左上、右上、右下、左下の順番で丸め方を指定します。なお、省略して2つにした場合は左上＆右下と右上＆左下というように解釈され、1つにした場合は全ての角が指定されます。「border-radius: 50%;」とすることで円形にすることもできます。

なお、使うことはあまりありませんが「border-radius: 5px 10px 15px 20px / 20px 15px 10px 5px;」というようにすることで角の丸め方を楕円形にすることもできます。スラッシュの前が水平方向の半径でスラッシュの後が垂直方向の半径を表します。



作例：https://codepen.io/siroi_sakana/pen/zYYRqQY

text-shadow

文字に影をつけることができます。水平方向の距離、垂直方向の距離、ぼかし半径、色という順番に指定します。ぼかし半径、色は省略が可能です。例えば「text-shadow: 2px 2px 1px gray;」というように指定します。

また、コンマで区切ることで複数の値を指定することができます。これを利用して8方向に影をつけ幅1pxの縁取り文字を作ることもできます（2px以上では表示が崩れることがあります）。縁取り文字は以下のようにして実現できます。

```
text-shadow:
  0px  1px black, 0    -1px black, /* 右    左 */
  -1px 0   black, 1px  0    black, /* 上    下 */
  1px  1px black, -1px -1px black, /* 右下 左上 */
  -1px 1px black, 1px  -1px black; /* 左下 右上 */
```

影付き文字
縁取り文字

作例：https://codepen.io/siroi_sakana/pen/LYYQNdV

box-sizing

「box-sizing: border-box;」を指定することでwidthやheightなどの指定がpadding、borderも込みの長さになります。なお、従来のpadding、borderを含めない方式は「box-sizing: content-box;」で指定します。

作例：https://codepen.io/siroi_sakana/pen/YzzeqRp

linear-gradient()

グラデーションを画像のような扱いで表示できます。例が多くここには載せきれないため例については作例を確認してください。

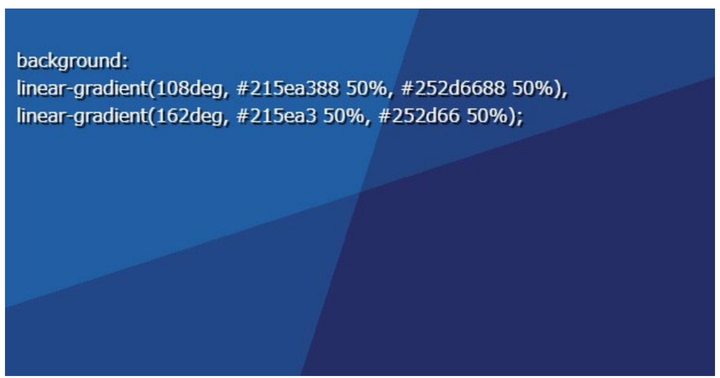
linear-gradient(red, blue)
赤から青にグラデーションする例
デフォルトでは上から下にグラデーションします

linear-gradient(#2f9170, #235f6b)
グラデーションの色はカラーコードによる指定もできます

作例：https://codepen.io/siroi_sakana/pen/ExxQZBX

background

背景を指定するためのプロパティです。背景は,(コンマ)で区切ることで複数指定することができます。複数指定した場合前に書いてある方が手前側に表示されます。



作例：https://codepen.io/siroi_sakana/pen/zYYmjrm

position: sticky

「position: sticky;」を指定した要素はブラウザ画面上の指定の位置に固定されます。スクロールしても位置を保ち続けるので、近年のサイトによくある上部メニューなどを実現することができます。なお、このようなメニューのことを**スティッキーヘッダー**と呼びます。

作例：https://codepen.io/siroi_sakana/pen/yLLvabg

transform

要素を変形させるためのプロパティです。「transform: rotate(30deg);」とすることで30°回転させられたり、「transform: scale(1.2, 1.5);」とすることで横1.2倍縦1.5倍に拡大できたり、「transform: skewX(30deg);」とすることでナナメ30°に傾斜させられたりします。また、これらは半角スペースで区切ることで複数適用することが可能です。コンマではないので注意してください。複数適用した場合手前から順番に変形が適用されます。順番が変わると形も変わるので複数適用する場合は順番に気を払うようにしましょう。

作例：https://codepen.io/siroi_sakana/pen/yLLRjqR

border-image

枠線に画像を使用するためのプロパティです。これを使うことでCSSだけでは表現しきれない複雑な枠線を表現できるようになります。仕様が複雑かつ本書では使用しないためここでは単に紹介に留めます。興味のある方は参考URLなどから調べてみてください。

CSS3のプロパティには他にもtransitionというものがあるのですが、これは後述する擬似クラスと併用されることが多いため擬似クラスの後で解説します。

3.3.5　疑似クラス

擬似クラスとは指定された要素が特定の状態にのみスタイルを適用させるセレクタのことです。「.hoge:hover」のように記述します。この例では「:hover」が擬似クラスであり意味合いとしては「.hogeにマウスオーバーしたとき」となります。主な擬似クラスには以下のようなものがあります。

:hover

指定の要素にマウスオーバーしたときに有効になる擬似クラスです。

:checked

チェックボックス（<input type="checkbox">）やラジオボタン（<input type="radio">）などの要素がチェックされている場合に有効になる擬似クラスです。

:disabled

無効な要素（disabledが設定されているinputなど）を表す擬似クラスです。有効な要素を表す「:enabled」もあります。

:nth-child(…)

兄弟要素の中で指定した位置のときに有効になる擬似クラスです。nthはn番目のという意味です（4thや5thの数字部分をnにしたもの）。…には位置を記述します。位置の指定方法には様々なものがあり、例としては以下のようなものがあります。

:nth-child(3)	3番目の要素
:nth-child(odd)	奇数番目の要素
:nth-child(even)	偶数番目の要素
:nth-child(5n)	5番目、10番目、15番目……の要素
:nth-child(5n+3)	8番目、13番目、18番目……の要素
:nth-child(-n+3)	最初の3つの要素

「:nth-child(1)」は「:first-child」とも書くことができます。また、「:nth-child(…)」の番号を数える順番を最後からにしたものである「:nth-last-child(…)」もあります。例えば「:nth-last-child(3)」であれば最後から数えて3番目の要素を表します。「:first-child」同様に「:nth-last-child(1)」は「:last-child」とも書けます。

兄弟要素に複数の型のタグが混在しており、そのうち特定の型のタグのみに「:nth-child(…)」を適用したい場合は「:nth-of-type(…)」を使用します。例えば「p:nth-of-type(even)」とすることで偶数番目のpタグに対してのみスタイルを適用させることができます。最後から数える「:nth-last-of-type(…)」も存在します。

:focus

要素がフォーカスされているときに有効になる擬似クラスです。例えば「input:focus」というようにすることで入力中のinput要素に対してスタイルを適用することができます。

:link/:visited

リンクの訪問状態を表す擬似クラスです。リンク先に未訪問であれば「:link」が、訪問済みであれば「:visited」が有効になります。ただし、プライバシー上の理由により「:visited」に対して使用できるスタイルには制限があります。制限については詳しくは以下のURLから確認することができます。

:visited - CSS: カスケーディングスタイルシート | MDN
https://developer.mozilla.org/ja/docs/Web/CSS/:visited

:not(…)

…に指定したセレクタが有効ではないときに有効になる擬似クラスです。例えば「div:not(.foobar)」であれば.foobarクラスが設定されていないdivに対してスタイルが適用されます。:not内では擬似クラスを用いることもでき、例えば「tr:not(:last-child)」であれば兄弟要素のうち最後の要素以外のtrに対してスタイルが適用されます。

3.3.6 transition

transitionはCSSのプロパティを連続的に変更し、アニメーションさせるためのプロパティです。これを使うことでホバーするとふわっと色が変わるメニューなどを作ることができます。transitionでは変化させるプロパティ、変化にかける時間、遅延させる時間（省略可能）、変化の方法（省略可能）の順番に指定します。例えば「200ミリ秒遅らせ、1秒かけてwidthを直線的に変化させる」場合は「transition: width 1s 200ms linear;」というようになります。指定するプロパティのところにallを指定すると全てのプロパティを変化させることができます。ここでは実際

どのようになるのかを見せるのは難しいので、実際どのようになるのかは作例を確認してください。

作例：https://codepen.io/siroi_sakana/pen/MWWPXBY

より複雑なアニメーションを作りたい場合、animationや@keyframeを使ったりインラインSVGと組み合わせてtransitionなどを使ったりすることで実現することができます。ここでは取り扱いませんが、興味のある方は調べてみてください。

3.3.7　疑似要素

CSS3では**疑似要素**を使うことでHTMLには書かれていない要素をCSSで作ることができます。主なものには「::before」と「::after」があり、contentに内容を記述することで指定の要素の前と後にその内容が追加されます。contentの内容を""にすることで文字なしで装飾などを加えることもできます。

「 カッコで囲む例 」
— 直線で囲む例 —

作例：https://codepen.io/siroi_sakana/pen/vYYQZLr

なお、擬似クラスと擬似要素を併用する場合疑似要素→擬似クラスの順番で記述します。擬似クラスはある要素の特定の状態を示すものであり、要素に対して指定する必要があるからです。

3.3.8　メディアクエリ

メディアクエリを使うことで画面サイズなどによってデザインを分けることができます。以下のように記述します。

```css
@media screen and (max-width: 1000px) {
  .widthcheck {
    background: red;
  }
}

@media screen and (max-width: 700px) {
  .widthcheck {
    background: blue;
  }
}
```

この例では画面の横サイズが1000px以下であれば.widthcheckの背景色が赤に、700px以下であれば青になります。メディアクエリを使うことでPC、タブレット、スマートフォンのデザインを1つのCSSで出し分けられるようになります。

作例：https://codepen.io/siroi_sakana/pen/mddQwVG

3.3.9 CSSフレームワーク

CSSフレームワークとは利用者がデザインを組みやすいように作られたCSSのライブラリのことを指します。ブラウザ間のデザイン差異を吸収してくれたり、レイアウトを組みやすくしてくれたり、おしゃれなデザインを提供してくれたり、その機能はCSSフレームワークによって様々です。有名なCSSフレームワークには以下のようなものがあります。

Normalize.css

ブラウザによって要素のデザインが若干違うのですが、それを統一してくれるCSSフレームワークです。こういうものを**CSSリセット**と呼びます。（ただし、厳密に言えばCSSリセットの定義は少し異なります。）このブラウザでは上手く表示できるのに別のブラウザだと表示が崩れてしまっているというような事態を防ぐことができるので、CSSフレームワークを使う気がなくてもCSSリセットだけは入れておくと便利です。Normalize.cssは以下のサイトよりダウンロードすることができます。

Normalize.css: Make browsers render all elements more consistently.
https://necolas.github.io/normalize.css/

Skeleton

UIのデフォルトデザインをちょっと良くしてくれたり、グリッドシステムを使えるようにしてくれたりする超軽量CSSです。できることは少ないですが、その分汎用的に使えるので便利です。レスポンシブデザイン（さまざまな画面サイズに対応したデザイン）にも対応しています。利用する場合は先述のNormalize.cssと組み合わせて使うことが多いです。Skeletonは以下のサイトよりダウンロードすることができます。

Skeleton: Responsive CSS Boilerplate
http://getskeleton.com/

Boostrap

Twitter社が開発した超有名CSSフレームワークです。それに関連したJSフレームワークなども提供しています。レスポンシブデザインに対応しており、さまざまなUIにモダンなデザインを提供してくれます。数多くのサイトで採用されているので見たことがあると感じる方も多いのではないでしょうか。動作には別途jQueryが必要だったり、できることが多い分ファイルサイズが大きすぎたりするのが玉に瑕です。Boostrapは以下のサイトよりダウンロードすることができます。

Bootstrap · The most popular HTML, CSS, and JS library in the world.
https://getbootstrap.com/

Bulma

CSS3の機能であるflexboxをふんだんに使った新しいCSSフレームワークです。クラスの書き方が直感的で使いやすく、最近人気があります。新しい分Boostrapなどに比べると日本語資料が少ないので英語情報を読むのが苦手な場合は注意が必要かもしれません。Bulmaは以下のサイトよりダウンロードすることができます。

Bulma: Free, open source, and modern CSS framework based on Flexbox
https://bulma.io/

3.3.10 Sass(SCSS)

SassとはCSSの記法を拡張したもので、変数や入れ子表現などができるようになります。Sassには**Sass構文**と**SCSS構文**という2種類の書き方があるのですが、本書ではCSSの延長線上の感覚で扱うことのできるSCSS構文の方を使用します。利用する際はSass記法やSCSS記法でコードを書きそれをCSSに変換して使います。このように予め変換しておく処理およびそのためのソフトウェアのことを**プリプロセッサ**と呼びます。SCSSそのままでは使うことはできません。

SCSSには数多くの機能があるのですが、とりあえず使う分には以下の3つを覚えておけば大丈夫です。

変数

SCSSでは変数は$を使って宣言、利用することができ、CSSプロパティの値をいろいろ入れることができます。具体的には以下のようにします。以下の例ではメインカラーで塗りつぶしサブカラーで枠やテキストを表示している.button-filledとその逆の.button-outlinedのスタイルを定義しています。

SCSS
```scss
$maincolor: #444;
$subcolor: #DDD;
$button_width: 200px;
$button_height: 60px;
$button_margin: 10px;

.button {
  display: flex;
  justify-content: center;
  align-items: center;
  width: $button_width;
  height: $button_height;
  margin: $button_margin;
}

.button-filled {
  background-color: $maincolor;
  color: $subcolor;
```

```
  border: 2px solid $subcolor;
}

.button-outlined {
  background-color: $subcolor;
  color: $maincolor;
  border: 2px solid $maincolor;
}
```

変数を使うことで色を後から一括で変更できたりそのプロパティがどういった意味を持つのか分かりやすくしたりすることができます。

作例:https://codepen.io/siroi_sakana/pen/YzzRxQz

入れ子

SCSSでは{}を使って入れ子になっている要素を表現できます。これを使うことにより構造を分かりやすく表現することができます。

SCSS

```scss
ul {
  list-style: none;
  padding: 10px;

  li {
    margin: 5px;
  }
}
```

これはCSSでは以下のように変換されます。

SCSS
```
ul {
  list-style: none;
  padding: 10px;
}

ul li {
  margin: 5px;
}
```

&を使った表現

ただの入れ子表現では入れ子の内側は半角スペースで連結されますが、&を使うことでそのままで連結することができます。これは擬似クラスを表現するときに非常に便利です。

SCSS
```
.menubutton {
  width: 200px;
  height: 60px;
  background: red;

  &:hover {
    background: blue;
  }
}
```

これはCSSに変換すると以下のようになります。

CSS
```
.menubutton {
  width: 200px;
  height: 60px;
  background: red;
}

.menubutton:hover {
  background: blue;
}
```

また、これを利用することである要素とその派生をうまく表現することができます。変数で使った例はこれを使うことで以下のように表現できます。

SCSS
```
$maincolor: #444;
$subcolor: #DDD;
$button_width: 200px;
```

```
$button_height: 60px;
$button_margin: 10px;

.button {
  display: flex;
  justify-content: center;
  align-items: center;
  width: $button_width;
  height: $button_height;
  margin: $button_margin;

  &-filled {
    background-color: $maincolor;
    color: $subcolor;
    border: 2px solid $subcolor;
  }

  &-outlined {
    background-color: $subcolor;
    color: $maincolor;
    border: 2px solid $maincolor;
  }
}
```

3.4 Node.js

3.4.1　Node.jsとは

Node.jsは簡単に言ってしまうとサーバーで動くJavaScript及び動かすためのソフトウェアのことです。Google Chromeと同じJavaScriptエンジンであるV8 JavaScriptエンジンを採用しており、スクリプト言語としてはかなり高速に動作します。（ちなみにV8という名称はV8気筒エンジンにあやかったものです。）サーバーサイド言語として人気があり、優良なライブラリも数多く存在します。

3.4.2　シングルスレッド

Node.jsの特徴として**シングルスレッド**で動作するというものがあります。1つのプロセスでアクセスなどを処理します。これに対しPHPなどはアクセスがあるたびにプロセスを立ち上げ処理します（FastCGIなどを使っていない場合）。プロセスは例えるならWindowsのウィンドウみたいなものだと思ってください。Node.jsはウィンドウを1つだけ起動し、それを使いまわします。PHPなどはアクセスがあるたびに新しくウィンドウを立ち上げて処理し、終わったら閉じます。Node.jsのような方式のメリットは実行にかかるコストが少ないことです。逆にPHPなどの方式ではウィンドウを毎回毎回立ち上げるため、たとえそれが1+1レベルの簡単な処理であってもあまりにもアクセスが多いとパンクしてしまいます。

もちろんメリットだけではありません。PHPのような方式ではアクセスごとにウィンドウを作るのでどこかのウィンドウでエラーがあって強制終了が起きたとしても他のユーザーには影響ありません。しかし、Node.jsではウィンドウが1つしかないのでどこかでエラーが発生して強制終了が起きたら全てが停止してしまいます。一応フォローしておくと複数ウィンドウを用意できたり強制終了しても再起動してくれたりするツールがあります（とはいえそれでも堅牢性の面ではPHPに劣ります）。一長一短なのでどちらがいいのかは場合によるでしょう。

3.4.3　ノンブロッキングIO

PHPなどではあるプロセスでファイルの読み込みやデータベースのアクセスなど時間のかかるIO処理を行っていたとしても、別のプロセスではそれを気にすることなく普通に処理を進めることができます。しかし、Node.jsはプロセス1つで動作しているのでそれらを待っていると他の処理を進めることができず処理が溜まってアクセスがパンクしてしまいます。

その解決策としてNode.jsでは**ノンブロッキングIO**という方策が取られています。ファイルの読み書き、データベースのアクセスなど時間のかかる処理は非同期処理とすることでプログラム全体の流れを止めることなく処理するのです。このような特性のため、Node.jsでは非同期処理を多用することになります。

3.4.4　環境を整える

実際にNode.jsを使ってみる前に、Node.jsを使うための環境を整えましょう。Tera Termを閉じている場合は接続します。ここではrootではなくteikiで作業を行ってください。「su teiki」を実行すればrootから切り替えることができます。まずNode.jsのプロジェクトを入れるためのディレクトリを作ります。「mkdir /home/teiki/nodejs」を実行してください。これ以降、Node.jsのプロジェクトはこのディレクトリに作成します。

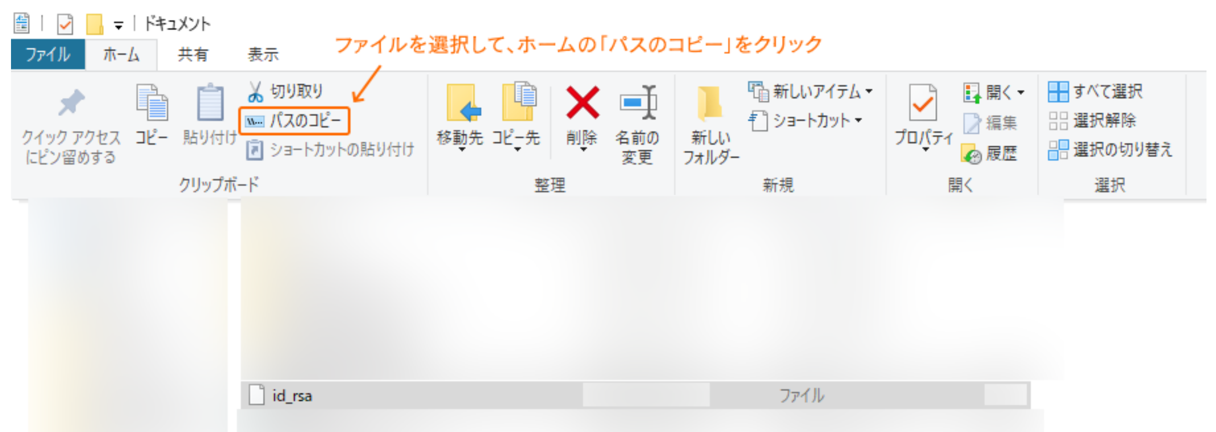
ディレクトリを作成したらVisual Studio Codeを起動します。起動したら画面左端のモニターのようなアイコンをクリックします。（以下リモートエクスプローラーメニューと呼びます。ない場合は『Visual Studio Codeの拡張機

能のインストール』の項目を飛ばしていると思われるのでそこに一旦戻ってください。）

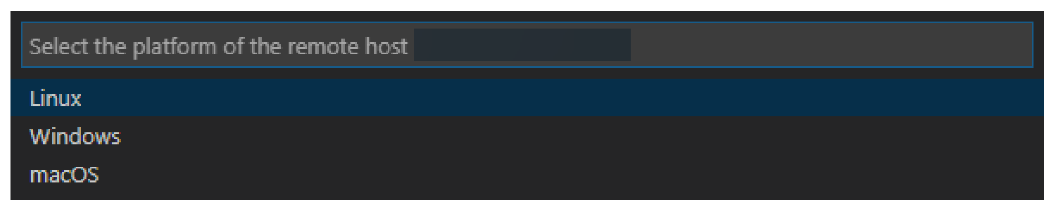
SSH TARGETSという項目があるのでマウスオーバーし、表示された＋マークをクリックします。「Enter SSH Connection Command」という項目が開くので秘密鍵ファイルのフルパスを確認しコマンドを入力します。例えば秘密鍵ファイルが「C:¥Users¥foobar¥Documents¥id_rsa」にある場合は以下のようになります。

```
ssh -i "C:\Users\foobar\Documents\id_rsa" teiki@dev.siroisakana.com -p 16815
```

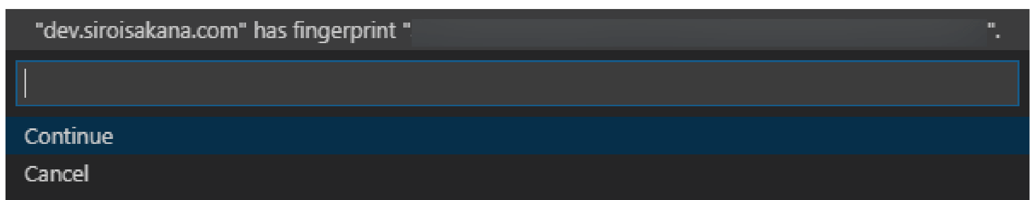
なお、Windows 10であればフルパスは以下のようにしてコピーすることができます。



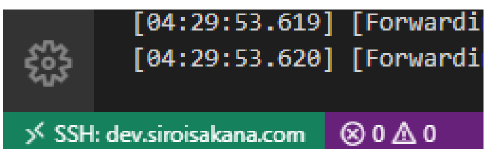
入力したら「Select SSH configuration file to update」という入力項目が開きます。これは接続設定の保存先です。上2つであればどちらでも構いませんのでどちらがいいか選択します。するとSSH TARGETSに接続先が追加されるのでそこにマウスオーバーし、+マークのついたウィンドウのようなアイコンをクリックします。クリックするとSSH接続モードで新しいウィンドウが開きます。ウィンドウ上部に下のようなウィンドウで接続先の環境を確認されるので「Linux」を選択します。



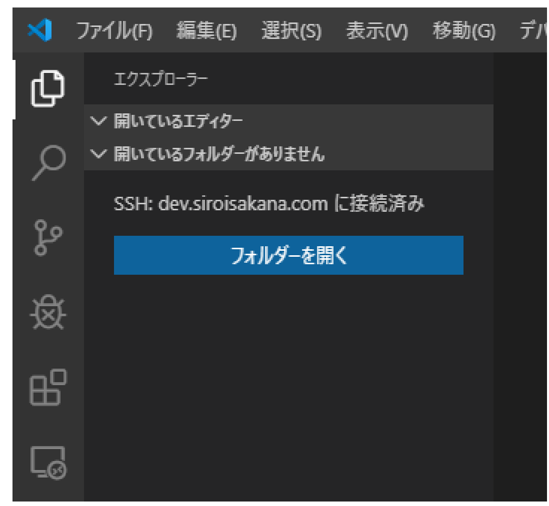
続けて接続していいか確認されるので「Continue」を選択しましょう。なお、選ぶまでに時間がかかってしまいエラーが出てしまった場合は「Retry」をクリックします。



接続できれば左下隅に緑背景に白地で「SSH: dev.siroisakana.com」というような感じで接続先が表示されているはずです。



次に左端メニューのコピー用紙のようなアイコン(以下エクスプローラーメニューと呼びます)をクリックし、「開いているフォルダーがありません」となっているところの「フォルダーを開く」という項目を選択します。

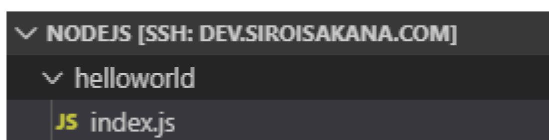


中央上部に「フォルダーを開く」という入力欄が表示されるので、「/home/teiki/nodejs/」と入力しOKを押しましょう。これで設定完了です。

なおVisual Studio Codeを閉じた後再度起動すると基本的には自動的に接続されますが、何らかの理由で自動で接続されなかった場合はリモートエクスプローラーメニューを開きSSH TARGETSから「dev.siroisakana.com」の「nodejs」を選択、そこで表示されるフォルダーのアイコンをクリックすることで再度接続が可能です。

3.4.5 Hello, World!

それではNode.jsのコードを書き実行してみましょう。エクスプローラーメニューの「NODEJS [SSH: DEV.SIROISAKANA.COM]」となっているところを開き、その下の空間内で右クリックし「新しいフォルダー」を選択します。フォルダー名の入力を求められるので「helloworld」と入力しエンターを押しましょう。作られた「helloworld」フォルダーを右クリックし、今度は「新しいファイル」を選択し「index.js」と入力します。ファイル構成は以下のようになります。

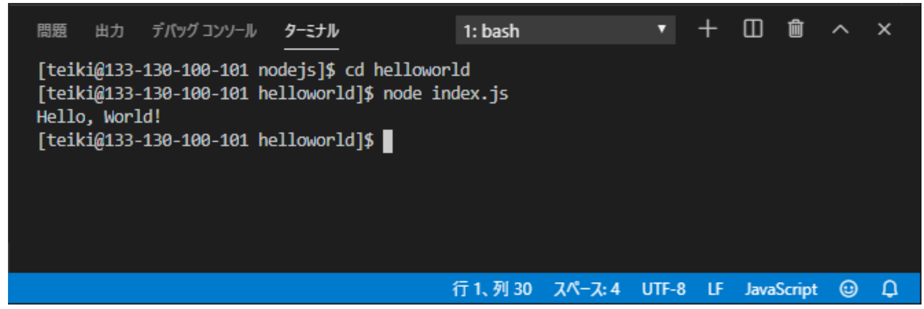


これにより「/home/teiki/nodejs/」内に「helloworld」と「helloworld/index.js」が作成されています。Visual Studio Code内ではVPS上のディレクトリやファイルをWindowsのフォルダーやファイルのように扱えるのです。さて、「index.js」を編集するタブが立ち上がっているはずなのでそこに以下の内容を入力し「Ctrl+S」を押して保存します。

index.js
```
console.log('Hello, World!');
```

次に上部メニューの「ターミナル」から「新しいターミナル」を選択します。

ソースコードを入力する所の下にターミナルが立ち上がります。なお、ここで起動するターミナルの初期ディレクトリは「/home/teiki/nodejs/」となっています。「cd helloworld」を実行してhelloworldディレクトリ内に移動し、「node index.js」を実行してみてください。「Hello, World!」と表示されていれば成功です。



3.4.6 ライブラリのインストールと使用法

Node.jsのライブラリのインストールする際はインストールしたいディレクトリ内で**npm install**コマンドを実行します。ここでは例としてvalidatorという文字列がメールアドレスかどうかなどを判定してくれるライブラリをインストールしてみましょう。ターミナルからhelloworldディレクトリ内で「npm install validator」を実行します。いくつか

WARN表示が出ますが無視して構いません。「added 1 package…」というような表示が出ていればインストール成功です。helloworldフォルダー内に「node_modules」というフォルダーが作成されており、その中にライブラリはインストールされています。

インストールと聞くとコンピューターそのものにインストールしてコンピューター全体で使えるようになるイメージがあるかもしれませんが、Node.jsでは基本的にライブラリはプロジェクトのフォルダーごとにインストールされます。そのため別のプロジェクトであるライブラリをインストールしたからといってまた別のプロジェクトで特にインストールせずにそのライブラリが使えるようになったりはしません。

NODEJS [SSH: DEV.SIROISAKANA.COM]
helloworld
node_modules
index.js
package-lock.json

さて、Node.jsでライブラリを読み込む方法には以下の2通りがあります。

```
const validator = require('validator'); // CommonJS
import validator from 'validator'; // ES6方式
```

前者の読み込み方をCommonJS方式、後者の読み込み方をES6方式などと呼んだりします。どちらを使っても構いませんが、本書では基本的にjsファイル内ではCommonJS方式、vueファイル内ではES6方式を使用します。

それではライブラリを使ってみます。「index.js」を以下のように書き換え保存してください。

index.js

```
const validator = require('validator'); // validatorの読み込み

const checkIsEmail = (str) => {
    //validatorを使ったメールアドレスかどうか判定する関数

    if (validator.isEmail(str)) {
        console.log(`${str}はメールアドレスです。`);
    } else {
        console.log(`${str}はメールアドレスではありません。`);
    }
};

checkIsEmail('foobar@hogehoge.com');
checkIsEmail('huga.foobar@mails.hogehoge.com');
checkIsEmail('foobar');
```

それでは実行してみましょう。カレントディレクトリがhelloworldではない場合は移動し、「node index.js」を実行します。コンソールに以下のような表示が出れば成功です。

```
foobar@hogehoge.comはメールアドレスです。
huga.foobar@mails.hogehoge.comはメールアドレスです。
foobarはメールアドレスではありません。
```

3.4.7　ファイルの分割

Node.jsではJavaScriptファイルを分割することができます。分割されたファイルの側ではアクセスさせたいオブジェクトを以下のような感じで指定することができます。分割したファイルのオブジェクトへアクセスするには先程学んだCommonJSの読み込み構文を使います。

```
module.exports = object; // CommonJS 方式
```

実際にファイルを分割してみましょう。「helloworld」内に「foobar.js」を作成してください。ファイル構成は以下のようになります。

```
helloworld
  foobar.js
  index.js
```

ファイルを作成したら「foobar.js」に以下の内容を入力して保存してください。

foobar.js

```
const huga = 'hugahuga';
const tasizan = (a, b) => {
  return a + b;
};

module.exports = {
  huga:    huga,
  tasizan: tasizan
};
```

「index.js」側でこれを読み込みましょう。「index.js」を以下のように書き換えて保存してください。ライブラリを読み込む際は読み込み先にはライブラリ名を指定しましたが、分割したファイルを読み込む際はそのファイルへの相対パスを指定します。

index.js

```
const foobar = require('./foobar.js');

console.log(foobar.huga);
```

```
console.log(`5+7 は${foobar.tasizan(5, 7)}`);
```

それぞれ保存したら「node index.js」で実行してみましょう。以下のような感じになれば成功です。しっかり読み込めていることが確認できます。

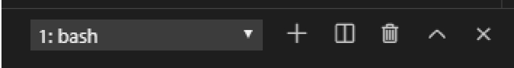
```
hugahuga
5+7は12
```

なお、ES6方式でファイルを分割することもでき、これはexport defaultで記述します。先程の例だと以下のような感じになります。

```
const huga = 'hugahuga';
const tasizan = (a, b) => {
  return a + b;
};

export default {
  huga:    huga,
  tasizan: tasizan
};
```

ただこれを読み込むにはいろいろと制約があるため本書では基本的にはCommonJS方式を使いES6方式は制約の条件を満たしている場所でのみ使います。具体的には本書では後々出てくるNuxt.js内で使うことになるのでこちらも覚えておいてください。

これで「Node.js」の基本講座は終了です。終わったらターミナルを閉じましょう。ターミナル右上のゴミ箱アイコンから閉じることができます。なお、×アイコンはターミナルを一時的に非表示にする際に使用します。



3.5 フロントエンドの基礎

3.5.1　フロントエンドとは

Webアプリ開発においてフロントエンドとはWebページの見た目を作ったり、UIを実現するためのコードを書いたり、サーバーのAPIを利用したりなど、主にクライアント側に関連する部分のことを指します。定期・APゲームにおいてはキャラクターページの表示をしたり、行動宣言の内容をサーバーに送信したり、様々な機能を持ったボタンを作ったりなどの部分が該当します。

3.5.2　3大フレームワーク

Webアプリを作るためのフレームワークには主に以下の3つがあります。

Angular

AngularはGoogleとコミュニティによって開発されているフレームワークで、記法がしっかり固まっているため大人数によるアプリケーション開発に強みをもちます。また、他2つとは違いWeb開発に必要な機能が全部備わっています。反面、利用にはAngularの仕様をしっかりと把握する必要があったり、Angular Styleと呼ばれるスタイルのコーディングをしなければならなかったり、TypeScriptという言語を習得しなければならなかったりします。そのため個人で小規模なアプリケーションを開発するといった用途にはあまり向いていません。企業開発向けのフレームワークと言えるでしょう。

React

ReactはFacebookとコミュニティによって開発されているフレームワークです。有名所だとTwitter、Slackなどは Reactによって作られています。開発にはJSXという記法を用いることになります。

Vue.js

AngularやReactとは違い、企業ではなくコミュニティ主体によるフレームワークです。元Angular開発チームだったEvan Youさんとコミュニティによって開発されています。ZOZOTOWN、noteなどで利用されているほか、AdventarもVue.jsによって作られています。Vue.jsのメリットはなんといっても記法が親しみやすく直感的で、習得が非常に容易なことです。HTML、JavaScript、CSSが書ければその延長線上の感覚で扱うことができます。反面、その柔軟な記法ゆえに設計に悩むことがあったり、後方互換性が他2つのフレームワークに比べると低かったりします。

定期・APゲームの開発をするならReactかVue.jsを使うのがいいでしょう。本書では学習コストの低さや関連ライブラリの優秀さ、日本語情報の量などを加味しVue.jsを利用します。

3.5.3　Nuxt.jsとは

Nuxt.jsはVue.jsのSSRフレームワークです。**SSR**（サーバーサイドレンダリング）とは何かというと、Vue.jsは通常ブラウザ側でHTMLを組み上げるのですが、それを予めサーバー側でHTML組み上げてからクライアント側に送信するようにするための技術です。SSRのメリットには初期表示が早くなる、SEO（Googleなどの検索エンジン

の検索順位を上げること）に有利になる、OGPなどが使えるようになるなどといったものがあります。
　ただし今回SSRは使いません。簡単に使えるようになっているとはいえやはりSSRに対応したプログラミングは少し面倒ですし、メリットである初期表示速度の改善やSEOも定期・APゲームにとってはそこまで重要ではないからです。
　SSRフレームワークなのにSSRを使用しないのは少し変に感じるかもしれませんが、Nuxt.jsにはVue.jsを使いやすくした機能が豊富にあるので今回はそちらの目的でNuxt.jsを使用します。Nuxt.jsにはSSR以外にも以下のような機能やメリットがあります。

ホットリロード

　ソースコードを編集するたびにいちいち保存し、実行を停止して再起動しブラウザをリロードするというようなことをしなくても保存するだけで自動的に変更が反映されるようになります。

Webpack

　Nuxt.jsに同梱されている、複数のファイルを1つにまとめてくれるライブラリです。Nuxt.jsを利用することで、このページにはこのライブラリが必要だからここでこれをロードしておくといった面倒な処理をしなくてもWebpackが自動でやってくれるようになります。

Babel

　ES6以降の一部ブラウザしか対応していない新しいJavaScriptの記法をほとんどの環境で動作するレガシーな記法に変換してくれるライブラリです。Nuxt.jsではBabelがデフォルトで設定されているのでブラウザ間差異をあまり気にすることなくJavaScriptを記述することができます。

Sass(SCSS)

　Nuxt.jsを使うと『Sass(SCSS)』でも紹介したSCSSを簡単に使えるようにしてくれます。

UIフレームワーク

　Vue.js向けに作られた**UIフレームワーク**を容易に扱えるようにしてくれます。UIフレームワークとはボタンやフォーム、タブなどWebアプリ制作でよく使うコンポーネントをまとめたライブラリのことです。Vue.jsのUIフレームワークにはVuetify、BoostrapVueなどがあります。ただし、Vue.jsはゲームを作るためのライブラリではないので当たり前ですがそのデザインはゲーム制作向きではないので本書では取り扱いません。

Vuex

　VuexはVue.jsのための状態管理ライブラリです。Vue.jsでは親から子、子から親へ情報を受け渡すのは簡単ですが、例えば孫要素の状態によってここを表示するかしないかなど、複数の要素をまたがなければならない情報の管理は苦手です。また、ユーザーの状態など、どこからでも参照できるようにしたいデータをどのコンポーネントにもたせるというのも非常に難しい問題です。そのような状況を解決するためのライブラリがVuexです。
　そのような情報はコンポーネントに持たせるのではなく、Vuexに管理させるようにします。Vuexはどこからでも参照することができるので、Vuexに情報をもたせることで遠く離れたコンポーネントの状態を知ることができたり、

ユーザー情報などどこからでも参照できるようにしておきたい情報を管理することができるようになったりします。

構成に関しての規約

Vue.jsは記法がゆるく、様々な構成を作ることができます。逆に言えば、初心者の場合どのような構成にすれば悩むことになってしまいます。Nuxt.jsにはある程度の構成に関する規約が存在するため、Nuxt.jsを導入することで構成に悩まなくてもよくなります。規約に従って書くことで、このページはこのURLでアクセスできるようにするなどの**ルーティング**の記述をしなくてもよくなるのもメリットでしょう。

3.5.4 拡張機能のインストール

Visual Studio Codeでは一部の拡張機能は接続先ごとに導入します。今回使用する拡張機能だと『Vetur』がそれにあたります。Vue.jsについて学ぶ前に拡張機能をインストールしておきましょう。Visual Studio Codeの左メニューから拡張機能を選択し、表示されたメニューの検索バーに『Vetur』と入力してください。

するとその拡張機能が表示されます。そこに「Install in SSH: dev.siroisakana.com」というようなボタンがあるはずなので、それをクリックしてください。インストールが完了すると「再読み込みが必要です」というような表示に変わるのでボタンをクリックしてください。Visual Studio Codeが再起動されて設定が反映されます。

3.5.5 リバースプロキシとその設定

それでは実際にNuxt.jsを利用してページを作ってみることにします。その前にNginxの設定を変更しましょう。Webアプリを公開するときは多くの場合**リバースプロキシ**という代理で応答しアクセスを転送する仕組みを利用します。

リバースプロキシを利用することで開放しなければならないポートが減りセキュリティが向上したり、データの圧縮や通信の暗号化などの処理をわざわざ行わなくてもNginxに任せることができるようになったりします。それではNginxでリバースプロキシの設定をしていきましょう。まずTera Termを起動しrootユーザーでログインします。「vi /etc/nginx/conf.d/default.conf」を実行し、「location /」となっているところの次に以下のように追記して保存します。追記した部分は斜体で表示してあります。

/etc/nginx/conf.d/default.conf

```
（省略）

  location / {
      root   /usr/share/nginx/html;
      index  index.html index.htm;
  }

  location /test/ {
      proxy_pass  http://127.0.0.1:3000;
  }

（省略）
```

ここでは「/test/」へのアクセスをこのマシン上の3000番ポートへ転送するよう設定しています。「127.0.0.1」は自分自身を表す特別なIPアドレスで**ループバックアドレス**と呼びます。他にも「localhost」というものもあるのです

が「localhost」を使うと思わぬ罠にはまることもあるのでnginxに記述するときは「127.0.0.1」を使いましょう。設定が完了したら「nginx -s reload」を実行して設定を反映させます。

なお、Nginxのproxy_passを記述する際は注意が必要です。以下の2つはURLの最後に/がついているかついていないかだけの違いに見えますが、Nginxにおいてその意味合いは全く異なります。

```
proxy_pass  http://127.0.0.1:3000;
proxy_pass  http://127.0.0.1:3000/;
```

実際にどう違うか見ていきましょう。以下のようなリバースプロキシの設定を組んだとします。……には「http://127.0.0.1:3000;」もしくは「http://127.0.0.1:3000/;」が入ります。

```
location /hoge/huga/ {
    proxy_pass  ……
}
```

これでインターネットから「/hoge/huga/」以下にアクセスがあった際に3000番ポートにアクセスが転送されるようになります。違うのはそのアクセスの転送方法です。例えば「/hoge/huga/foobar/」にアクセスがあったとしましょう。前者の/がない方であればそのまま「/hoge/huga/foobar/」にアクセスがあったという風に3000番ポートに処理が渡されます。しかし、後者の/がある方では「/foobar/」にアクセスがあったという風に処理が渡されるのです。アクセスの渡し方によって3000番ポート側での処理が変わってきます。

どちらがいいのかは場合によります。前者であればVPS内外どちらからアクセスする場合でもほとんど変わらないURLでアクセスすることができますし、後者であれば公開するURLを変更したい場合でもNginxの設定を変更するだけで実現できるようになります。本書では前者を使用しています。

どちらを利用するにせよ、Nginx側のリバースプロキシの設定はどうなっているか、処理の受け取り方は合っているかなどは意識するようにしましょう。うまくアクセスできないと思ったら設定が合致していなかったということはよくあります。

3.5.6 Nuxt.jsの初期設定

実際にNuxt.jsを使ってページを作ってみましょう。まず、Visual Studio Code上でターミナルを開きます。Nuxt.jsのプロジェクトを作るには**npx**というコマンドを使用します。今回プロジェクト名は「test」とします。カレントディレクトリが「/home/teiki/nodejs」であることを確認し、「npx create-nuxt-app test」と入力します。ダウンロードなどの処理が行われるのでしばらく待つと、プロジェクトの詳細を尋ねられます。ほとんどデフォルト設定のままエンターを押して構いませんが、以下の3項目についてはこの指定の通りに選択してください。キーボードの上下の矢印キーを使って選択できます。「Choose development tools」についてはスペースキーで指定の項目を有効化してからエンターを押します。

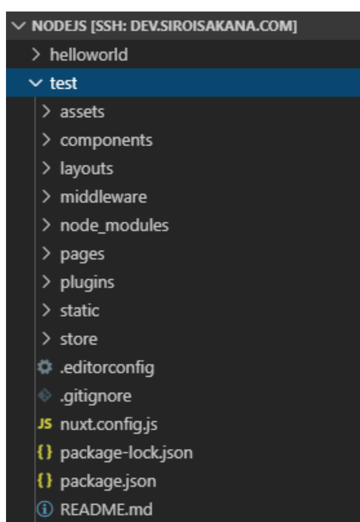
「Choose the package manager」…… Npm
「Choose rendering mode」…… Single Page App
「Choose development tools」…… jsconfig.jsonをスペースキーで有効化

```
[teiki@            nodejs]$ npx create-nuxt-app test

create-nuxt-app v2.12.0
✨ Generating Nuxt.js project in test
? Project name test
? Project description My phenomenal Nuxt.js project
? Author name
? Choose the package manager Npm
? Choose UI framework None
? Choose custom server framework None (Recommended)
? Choose Nuxt.js modules (Press <space> to select, <a> to toggle all, <i> to invert selection)
? Choose linting tools (Press <space> to select, <a> to toggle all, <i> to invert selection)
? Choose test framework None
? Choose rendering mode Single Page App
? Choose development tools jsconfig.json (Recommended for VS Code)
ℹ Installing packages with npm
```

なお、それぞれの項目で聞かれている内容の意味は以下のとおりです。

Project name	プロジェクト名
Project description	プロジェクトの概要
Author name	作者名
Choose the package manager	プロジェクトの管理に使用するパッケージマネージャ
Choose UI framework	使用するUIフレームワーク
Choose custom server framework	使用するサーバーフレームワーク
Choose Nuxt.js modules	使用するライブラリ
Choose linting tools	使用するコーディング規約ツール
Choose test framework	使用するテストツール
Choose rendering mode	レンダリングモード
Choose development tools	使用する開発ツール

入力が終わったら「Installing packages with npm」という表示になるのでしばらく待ちます。「Successfully created project test」という表示が出ればプロジェクトの作成に成功しています。Visual Studio Codeのエクスプローラーメニューに「test」というフォルダーが追加されているはずです。

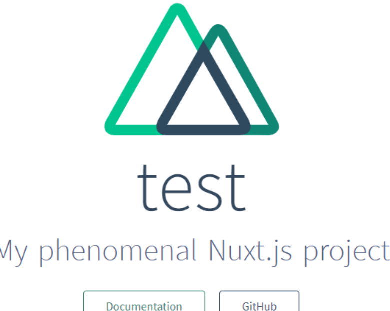


プロジェクトが作成されたら公開URLとポートの設定を行います。エクスプローラーメニューからtestフォルダーを開き、その中にある「nuxt.config.js」というファイルを開きます。色々と設定が記述されていますが、build:と書かれているところの前に以下のようにして設定を追記して保存します。なお、追記した箇所は斜体で表示してあります。

nuxt.config.js

```
（省略）

server: {
  port: 3000
},
router: {
  base: '/test/'
},
/*
** Build configuration
*/
build: {

（省略）
```

設定が完了したら実行してみましょう。「cd test」でtestディレクトリ内に移動し、「npm run dev」を実行します。なお、このコマンドは「開発モードでプロジェクトを実行する」という意味になります。実行してしばらく待ったら「waiting for file changes」というような表示になるので、「http://dev.siroisakana.com/test/」にアクセスします。以下のようなページが表示されれば実行に成功しています。



3.5.7　Vue.jsの基礎

それでは実際にコードを書いてみながらVue.js、Nuxt.jsについて学習していきましょう。Nuxt.jsにはホットリロード機能があるので、ブラウザとコンソールはそのままの状態にしておいてください。編集すると自動的に変更が反映されます。

まず「test」フォルダー内の「pages」というフォルダーを開きます。その中に「index.vue」というファイルがあるので開きましょう。まずは最小の構成で作ってみます。index.vueの内容を全て消去し、以下の内容を入力します。入力したら「Ctrl+S」を押し保存しましょう。するとコンソールに「Updated pages/index.vue」と表示され、ブラウザ側でもページの内容が自動的に切り替わるはずです。

pages/index.vue

```
<template>
  <section class="helloworld">
    Hello, {{ hensu }} World!
  </section>
</template>

<script>
export default {
  data: function() {
    return {
      hensu: 'Nuxt.js'
    };
  }
}
</script>

<style>
.helloworld {
  font-size: x-large;
}
</style>
```

Hello, Nuxt.js World!

<template>内でHTMLを、<script>内でJavaScriptを、<style>内でCSSを記入しています。この例では<script>内でhensuを宣言し、<template>内でメッセージとともにhensuの内容を表示し、<style>内で文字を大きくしています。Vue.jsでは<script>内のdataメソッドで扱いたい変数とその値をreturnし、<template>内で{{ }}で変数などの値を囲む(**マスタッシュ構文**)ことでその値を表示できます。

Vue.jsではこのようにして**コンポーネント**を記述します。コンポーネントとは何かというとWebアプリの部品のことを指します。例を上げてみましょう。定期・APゲームは「キャラクターアイコン」「メニュー」「チャットの発言」などなど、様々なUI要素が組み合わさってできています。それら一つ一つの要素のことをコンポーネントといいます。Vue.jsでは「CharacterIcon.vue」「Menu.vue」「ChatMessage.vue」というようにコンポーネントごとにファイルを作り、それを組み上げることでWebページを作っていきます。

これによりどのようなメリットがあるかというと、例えばチャットの発言とキャラクターリストでキャラクターアイコンを表示したいとします。従来であればチャットの発言とキャラクターリスト両方にキャラクターアイコンを表示するためのコーディングを行い表示させていました。Vue.jsでは「ここにキャラクターアイコンを読み込む」といったようにコーディングするだけで表示できます。また「キャラクターアイコンに枠線をつけたい」「サイズを変えたい」となった場合に従来であればチャットの発言とキャラクターリスト両方を変更しなければなりませんでしたが、Vue.jsではキャラクターアイコンファイルを変更するだけで実現できたりします。

また、Vue.jsでは変数とUIの表示を結びつけることができます。例えば<input>にfoobarという変数を結びつけたとします。すると、結びつけた<input>に「hoge」と入力するだけでいちいち取得処理を書かなくても変数foobarの中身が「'hoge'」になります。また「foobar = 'hugahuga';」とするだけで反映処理を書かなくても<input>の中身が「hugahuga」になります。このように変数とUIの表示が相互に結びついているシステムのことを**リアクティブシステム**と呼んだりします。

Vue.jsではv-modelという属性を使うことで変数と要素を結びつけることができます。それでは実際にやってみましょう。index.vueの中身を全て消去し、以下の内容を入力して「Ctrl+S」で保存します。

pages/index.vue

```
<template>
  <section>
    <input type="text" v-model="foobar"><br>
    変数 foobar の内容：{{ foobar }}
  </section>
</template>

<script>
export default {
  data: function() {
    return {
      foobar: 'hugahuga'
    };
  }
}
</script>

<style>
</style>
```

保存するとページが以下のようになります。

hugahuga
変数foobarの内容：hugahuga

テキストボックスの中身をいろいろ変えてみてください。編集すると同時に「変数foobarの内容」と書かれたところがどんどん書き換わっていきます。テキストボックスを編集すると同時に紐付けられた変数foobarが書き換わり、さらにそれによって{{ foobar }}で表示される内容も書き換わるためにこのようになります。

3.5.8 v-bind

変数をページ上の表示ではなくHTML要素の属性に反映させたい場合があります。このような場合に使うのがv-bindです。以下のように「pages/index.vue」を書き換えて保存してみましょう。

pages/index.vue

```
<template>
  <section>
    <input type="text" v-bind:placeholder="foobar">
  </section>
</template>

<script>
export default {
  data: function() {
    return {
      foobar: 'なにかここに入力'
    };
  }
}
</script>

<style>
</style>
```

ページが以下のようになります。変数foobarの内容が「v-bind:placeholder="foobar"」によってplaceholder属性に反映されています。

なにかここに入力

「v-bind:」は「:」というように略記することもできます。つまり「:placeholder="foobar"」というようにも書けるということです。本書ではこれ以降略記を利用します。

3.5.9 v-if / v-else-if / v-else

Vue.jsではJavaScriptの「if」「else if」「else」のように「v-if」「v-else-if」「v-else」を使って表示を出し分けるこ

とができます。実際にどういうことなのか見てみましょう。index.vueの内容を以下のように書き換えます。<template>内の<span>に注目してください。

pages/index.vue

```
<template>
  <section>
    パスワード：<input type="password" v-model="password"><br>
    <br>
    パスワードの長さ：{{ password.length }}文字<br>
    <span v-if="password.length == 0"> <!-- password の長さが 0 の場合 -->
      パスワードを入力してください。
    </span>
    <span v-else-if="password.length < 8"> <!-- それ以外で、password の長さが 8 文字より短い場合 -->
      パスワードが短すぎます。
    </span>
    <span v-else> <!-- それ以外の場合 -->
      パスワード OK。
    </span>
  </section>
</template>

<script>
export default {
  data: function() {
    return {
      password: ''
    };
  }
}
</script>

<style>
</style>
```

実行すると以下のようになります。

パスワード：

パスワードの長さ：0文字
パスワードを入力してください。

この例ではパスワード入力欄に表示された文字数に応じてメッセージを出し分けています。実際にパスワード入力欄にいろいろ入力してみてください。パスワードの長さが0であれば「パスワードを入力してください」、パスワードの長さが8未満であれば「パスワードが短すぎます」、そのどちらでもない場合は「パスワードOK」と表示されます。このように、「v-if=""」「v-else-if=""」の中に条件文を書くことで表示を分けることができます。

3.5.10 v-on

v-onを使うことでイベントがあった際の処理を指定できます。例えば「<div v-on:click="clicked">」というようにすることで、指定した<div>がクリックされた際にclickedというメソッドを呼び出せます。実際に書いてみましょう。index.vueの内容を以下の内容に変更します。

pages/index.vue

```
<template>
  <section>
    <button v-on:click="buttonClick">ここをクリック！</button><br>
    {{ clickCount }}回クリックしました。
  </section>
</template>

<script>
export default {
  data: function() {
    return {
      clickCount: 0
    };
  },
  methods: {
    buttonClick: function() {
      this.clickCount++;
    }
  }
}
</script>

<style>
</style>
```

保存するとページが以下のようになります。「ここをクリック！」となっている<button>をクリックするたびに「〇回クリックしました。」となっているところのカウンターが増えていきます。

ここをクリック！
0回クリックしました。

実際どういう流れになっているのかコードを見てみましょう。まず<button>に「v-on:click="buttonClick"」が指定されています。これにより、このボタンがクリックした際に「buttonClick」が呼び出されます。「buttonClick」は<script>内の「methods」内で定義されています。そしてbuttonClick関数の中では予めdataで宣言してあったclickConutが+1されています。これによってクリックの回数をカウントし表示しているというわけです。

methodsの関数を定義する際には注意が必要です。methods内にてdataで宣言した変数にアクセスするためにはthisを使う必要があるのですが、ここをアロー関数にしてしまうとthisが呼び出された先でも変更されずうまく変数にアクセスできなくなってしまいます。methodsの宣言にはfunction ()を使うようにしましょう。

なお、v-onは@というように略記することができます。「v-on:click="buttonClick"」は「@click="buttonClick"」

というように書けるということです。本書ではこれ以降後者の記法を採用します。
　v-onで使えるイベントには他にも以下のようなものがあります。

@click	クリックされたとき
@focus	フォーカスされたとき
@blur	フォーカスが外れたとき
@mouseover	マウスオーバーされたとき
@mouseout	マウスオーバーが外れたとき
@change	要素が変更されたとき （変更された瞬間ではなく、変更されてフォーカスが外れたときに発火されます）
@keydown	キーが押されたとき
@keyup	キーが離されたとき

3.5.11　v-for

　Webアプリでは同じ要素を繰り返し表示したいことがあります。定期・APゲームではキャラクターリストなどがそれにあたるでしょう。そのような際に使うのがv-forです。v-forを使うと配列を使って要素を繰り返し表示させることができます。実際に簡易なキャラクターリストを作ってみましょう。以下のようになります。

pages/index.vue

```
<template>
  <section>
    <div class="characterlist_item" v-for="character in characterList" :key="character.eno">
      <span class="name">{{ character.name }}</span>
      <span class="eno">(ENo.{{ character.eno }})</span><br>
      <span class="description">{{ character.description }}</span>
    </div>
  </section>
</template>

<script>
export default {
  data: function() {
    return {
      characterList: [
        {eno: 1, name: "太郎", description: "フルネームは山田太郎"},
        {eno: 2, name: "花子", description: "最近英語で赤点を取った"},
        {eno: 3, name: "三郎", description: "この世の全ての咎を背負う者"}
      ]
    };
  }
}
</script>

<style>
.characterlist_item {
  width: 600px;
  padding: 8px;
```

```
  border-top: 1px solid gray;
}

.characterlist_item:last-child {
  border-bottom: 1px solid gray;
}

.name {
  font-weight: bold;
}

.eno {
  color: #888;
  font-size: small;
}

.description {
  color: #222;
}
</style>
```

これを実際に表示すると以下のようになります。

太郎 (ENo.1)
フルネームは山田太郎

花子 (ENo.2)
最近英語で赤点を取った

三郎 (ENo.3)
この世の全ての咎を背負う者

このように「v-for="character in characterList"」というようにして配列から値を取り出して繰り返し処理ができます。v-forの内部ではcharacterListから取り出した要素であるcharacterが使えるということです。なお、v-forには配列の値ごとに一意な値を設定しkey属性に設定しないといけないという制約があります。ここではenoが一意な値になるのでこれをkey属性に割り当てています。

3.5.12 フィルター

フィルターを使うと指定の値をフィルタリングして簡単にその結果を表示したり使ったり出来ます。例えば時刻データなどを見やすい形式に変換して表示することができます。フィルターは<script>のexport default内にfiltersで宣言し、マスタッシュ構文内などで|(バーティカルバー)を使うことでフィルタリングすることができます。フィルタリングの際にはフィルタリングしたい値を左、フィルターを右に置きます。以下のようになります。

pages/index.vue

```
<template>
  <section>
      元の値：<br>
        {{ imanojikan }}<br>
```

```
      フィルターした値：<br>
        {{ imanojikan | dateFilter }}
  </section>
</template>

<script>
export default {
  data: function() {
    return {
      imanojikan: new Date()
    };
  },
  filters: {
    dateFilter: function(date) {
      return `${date.getFullYear()}年${date.getMonth()+1}月${date.getDate()}日`;
    }
  }
}
</script>

<style>
</style>
```

表示すると以下のようになります。Dateがフィルタリングされて使いやすい形式で表示されているはずです。

元の値：
　Fri Dec 06 2019 21:08:46 GMT+0900 (日本標準時)
フィルターした値：
　2019年12月6日

フィルターは要素のv-bindに使うことも出来ます。使う際は以下のように行います。

```
<template>
  <section>
    <img :src="rawSrc | srcFilter">
  </section>
</template>
```

3.5.13 双方向算出プロパティ

双方向算出プロパティを使うと関数をまるで変数のように扱うことができます。双方向算出プロパティではget関数とset関数を定義します。参照したときにはget関数の戻り値が値として扱われ、代入しようとしたときにはsetが呼び出されます。クラスのゲッター/セッターと同じような感じです。双方向算出プロパティは<script>のexport default内にcomputedを作って定義します。実際にやってみましょう。「pages/index.vue」を以下のように書き換え保存します。

pages/index.vue

```
<template>
  <section>
    <button @click="plus1">computedFoobar += 1;</button>
    computedFoobarの内容：{{ computedFoobar }}
  </section>
</template>

<script>
export default {
  computed: {
    computedFoobar: {
      get() {
        return 5; // 必ず5を返す
      },
      set(value) {
        // 代入しようとしたらアラートを表示
        alert("新たな値が代入されましたが反映されませんでした。");
      }
    }
  },
  methods: {
    plus1: function() {
      this.computedFoobar += 1;
    }
  }
}
</script>

<style>
</style>
```

ページを表示すると以下のようになります。ボタンをクリックすると「新たな値が代入されましたが反映されませんでした」というアラートが出ます。

computedFoobar += 1; computedFoobarの内容：5

コードの流れを追っていきましょう。まず「computedFoobarの内容:{{ computedFoobar }}」でcomputedFoobarが参照されます。computedFoobarのget関数が呼び出され5が返されるので、「computedFoobarの内容」には5が表示されます。

ボタンを押すと「this.computedFoobar += 1;」が呼び出され代入処理がされます。set関数が呼び出されますがset関数はアラートが表示されるだけで実際に代入などは行われないため、値の反映はされずcomputedFoobarの内容は5のままになります。

3.5.14 DOMの参照

例えば<textarea>の現在のカーソル位置を取得したいなど、HTMLの要素を参照したいこともあるでしょう。Vue.jsではrefを使うことでHTMLの要素を参照することができます。実際に書いてみましょう。「pages/index.vue」

を以下のように書き換え保存してください。

pages/index.vue

```
<template>
  <section>
    <textarea ref="myTextarea" v-model="textareaValue"></textarea><br>
    <button @click="cursorPositionAlert">カーソル位置を取得</button>
  </section>
</template>

<script>
export default {
  data() {
    return {
      textareaValue: 'ここに内容を入力'
    };
  },
  methods: {
    cursorPositionAlert() {
      alert(`カーソル位置は${this.$refs["myTextarea"].selectionStart}文字目です。`);
    }
  }
}
</script>

<style>
</style>
```

保存するとページの内容が以下のようになります。<textarea>の内容をいろいろ変えたりカーソル位置を変えたりしてから「カーソル位置を取得」ボタンを押してください。カーソル位置に応じた内容が表示されます。

ここに内容を入力

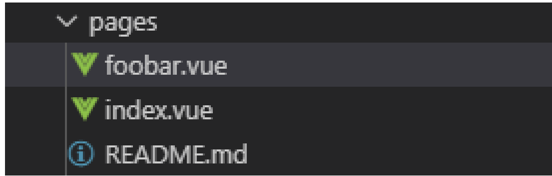
カーソル位置を取得

コードの流れを追っていきましょう。<template>内で<textarea ref="myTextarea">というようにref属性を使って参照するためのタグ付けを行っています。また、methods内でthis.$refs["myTextarea"]で<textarea>の要素を参照しカーソル位置を取得しています。

$refsの注意点としてレンダリング(実際に要素が表示されるとき)までは参照できないというものがあります。HTML要素を参照するものなので、HTML要素が作られるまでは参照できないのです。レンダリングされるまではアクセスしないようにしましょう。

3.5.15 新規ページの作成とルーティング

Nuxt.jsでは「pages」フォルダー内にvueファイルを作成することで自動的に新しいページが作成されます。実際に新しくページを作ってみましょう。「pages」フォルダーを右クリックして「新しいファイル」を選択し、「foobar.vue」を作成します。ファイル構成は以下のようになります。



作成したら「pages/foobar.vue」に以下の内容を入力して保存してください。

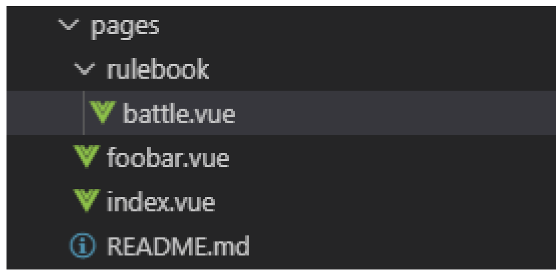
pages/foobar.vue

```
<template>
  <section>
    ここはfoobar.vue。
  </section>
</template>

<script>
export default {}
</script>

<style>
</style>
```

保存したら「http://dev.siroisakana.com/test/foobar」にアクセスしてみましょう。以下のようなページが表示されるはずです。

ここはfoobar.vue。

Nuxt.jsではこのようにしてページを追加できます。また「rulebook/battle」などトップディレクトリではなく深いディレクトリにページを作りたい場合にはフォルダーを使います。実際に作ってみましょう。まず「pages」フォルダーに「rulebook」フォルダーを作成し、さらにその中に「battle.vue」というファイルを作成します。以下のようなファイル構成になります。

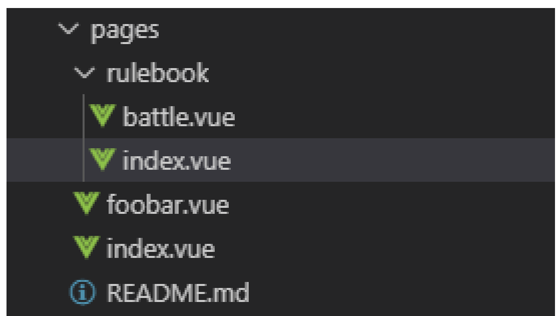


作成したら「rulebook/battle.vue」に以下の内容を入力し保存してみましょう。

pages/rulebook/battle.vue

```
<template>
  <section>
    ここはrulebook/battle.vue。
  </section>
</template>

<script>
export default {}
</script>

<style>
</style>
```

保存したら「http://dev.siroisakana.com/test/rulebook/battle」にアクセスします。「ここはrulebook/battle.vue。」とだけ表示されるページが作られています。Nuxt.jsではこのようにして深いディレクトリにページを作成します。また、この状態で「rulebook/」にページを作りたい場合は「rulebook」フォルダー内に「index.vue」を作成します。「rulebook/index.vue」を作成してみましょう。ファイル構成は以下のようになります。



「pages/rulebook/index.vue」に以下の内容を記入し保存します。

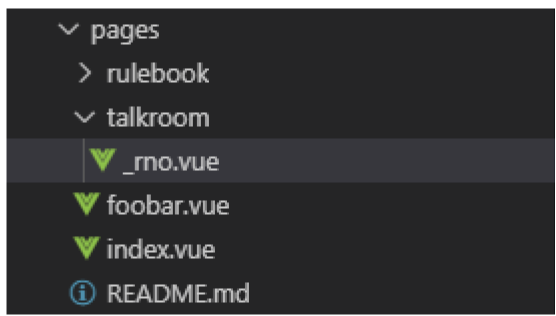
pages/rulebook/index.vue

```
<template>
  <section>
    ここはrulebook/のインデックスページ。
  </section>
</template>

<script>
export default {}
</script>

<style>
</style>
```

「https://dev.siroisakana.com/test/rulebook/」にアクセスすると「ここはrulebook/のインデックスページ。」というように表示されます。このようにしてNuxt.jsでは「index.vue」という名前でファイルを配置することで「/」で終

わるURLのルーティングを作ることができます。

　また、定期・APゲームを作る上で「talkroom/(ルーム番号)」というようなページを用意したい場合もあると思います。Nuxt.jsではファイル/ディレクトリ名の先頭に_(アンダースコア)をつけることでこのような動的なルーティングを実現することができます。実際にやってみましょう。「pages」フォルダーに「talkroom」フォルダーを作成し、その中に「_rno.vue」ファイルを作成します。ファイル構成は以下のようになります。



　「pages/talkroom/_rno.vue」に以下の内容を入力し保存します。

pages/talkroom/_rno.vue

```
<template>
  <section>
    <h2>ここは{{ $route.params.rno }}番目のトークルームです。</h2>
    <button @click="getRNo">トークルーム番号を取得</button>
  </section>
</template>

<script>
export default {
  methods: {
    getRNo: function() {
      alert(`トークルーム番号は${this.$route.params.rno}番です。`);
    }
  }
}
</script>

<style>
</style>
```

　「http://dev.siroisakana.com/test/talkroom/4」や「http://dev.siroisakana.com/test/talkroom/18」にアクセスしてみましょう。以下のようなページになるはずです。

ここは4番目のトークルームです。

[トークルーム番号を取得]

　動的なルーティングを行った場合、$route.params.(ルーティング名)で動的部分を取得することができます。

methods内などであればthis.$route.params.(ルーティング名)になります。この例であればファイル名が「_rno.vue」なので、$route.params.rnoでアクセスすることができます。

　しかし、この例では想定されていないURLからもアクセスできてしまいます。「http://dev.siroisakana.com/test/talkroom/hogehoge」にアクセスしてみましょう。以下のような表示になります。

ここはhogehoge番目のトークルームです。

トークルーム番号を取得

　一般的な定期・APゲームと同じくトークルーム番号が自然数(1, 2, 3, …)しか存在しないのであればこのようなURLにアクセスできるのは変ですし、予期せぬバグの原因にもなります。そこでNuxt.jsではvalidateを使うことで動的なルーティングにおいてアクセスできるURLを制限することができます。それではやってみます。<script>内を以下のように書き換えて保存しましょう。

pages/talkroom/_rno.vue

```
  (省略)

<script>
export default {
  validate({ params }) {
    return /^[1-9][0-9]*$/.test(params.rno);
    // params.rnoが自然数かどうか正規表現を使って判定
    // 最初の1文字目が1-9のどれかで2文字目以降～最後までが0-9なら自然数
    // 自然数であればtrueを返し、そうでなければfalseを返す
  },
  methods: {
    getRNo: function() {
      alert(`トークルーム番号は${this.$route.params.rno}番です。`);
    }
  }
}
</script>

  (省略)
```

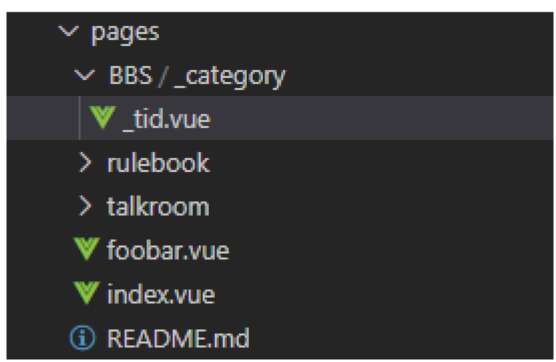
　validateではvalidate({ params })とすることで動的ルーティングのパラメーターを取得できます。今回の例では「params.rno」でアクセスできます。validate内でfalseを返せばルーティングは無効となり、反対にtrueを返せば有効になります。今回はparams.rnoが自然数の場合のみ有効にするようにしてあります。

　実際に動作するかどうかテストしてみましょう。以下のURLにそれぞれアクセスしてみてください。上2つのURLからはアクセスでき、下2つのURLだと「This page could not be found」と出るようになっていればOKです。

・http://dev.siroisakana.com/test/talkroom/4
・http://dev.siroisakana.com/test/talkroom/2000
・http://dev.siroisakana.com/test/talkroom/hogehoge
・http://dev.siroisakana.com/test/talkroom/200fuga

また、Nuxt.jsでは動的なルーティングの中にさらに動的なルーティングを作ることができます。例えばフリー掲示板・取引用掲示板など様々なカテゴリーのある掲示板を作りさらにその中にスレッドを作るといった場合に「BBS/(カテゴリー)/(スレッドID)」というようなルーティングを作れます。

実際に作ってみましょう。カテゴリーのURLは「free」「trade」、スレッドIDは自然数とします。「BBS」フォルダーを作成し、その中に「_category」フォルダー作成、さらにその中に「_tid.vue」を作成します。ファイル構成は以下のようになります。



「pages/BBS/_category/_tid.vue」の内容は以下のようになります。編集して保存してください。

pages/BBS/_category/_tid.vue

```
<template>
  <section>
    ここは{{ $route.params.category }}カテゴリーでスレッドIDは{{ $route.params.tid }}。
  </section>
</template>

<script>
export default {
  validate({ params }) {
    return (
      (params.category == 'free' || params.category == 'trade') &&
      /^[1-9][0-9]*$/.test(params.tid)
    );
    // params.categoryが"free"か"trade"で、tidが自然数ならtrue
  },
}
</script>

<style>
</style>
```

実際に確認してみましょう。以下のようなURLでアクセスしてみてください。上2つではアクセスできて下3つではアクセスできなければOKです。

http://dev.siroisakana.com/test/BBS/free/20
http://dev.siroisakana.com/test/BBS/trade/123
http://dev.siroisakana.com/test/BBS/hoge/20
http://dev.siroisakana.com/test/BBS/free/fugafuga
http://dev.siroisakana.com/test/BBS/trade/hogehoge

また、同ウィンドウ内かつNuxt.jsのルーティング内でページをリンクする際は<a>タグではなく<nuxt-link>タグを使用します。<nuxt-link>ではhref属性ではなくto属性を使ってリンク先を指定します。これにより、Nuxt.jsが自動的にダウンロードするべきファイルを判別し効率的にページを表示させられることができます。実際にやってみましょう。「pages/index.vue」を以下のように書き換え保存します。

pages/index.vue

```
<template>
  <section>
    <nuxt-link to="foobar">foobar</nuxt-link>
    <nuxt-link to="rulebook/battle">戦闘ルール</nuxt-link>
    <nuxt-link to="talkroom/20">20番トークルーム</nuxt-link>
    <nuxt-link to="BBS/free/123">フリー掲示板123番スレッド</nuxt-link>

    <nuxt-link class="decoratedlink" to="foobar">foobar</nuxt-link>
    <!-- 他のHTML要素同様にidやclassなどを付与することができる -->

    <a href="https://google.com/">Google (外部リンク)</a>
    <!-- 外部リンクはaタグを使わなければならない -->
  </section>
</template>

<script>
export default {}
</script>

<style>
.decoratedlink {
  color: red;
}
</style>
```

「http://dev.siroisakana.com/test/」を開きましょう。以下のようなページになります。いろいろリンクを開いてみましょう。しっかりとページ移動ができるはずです。

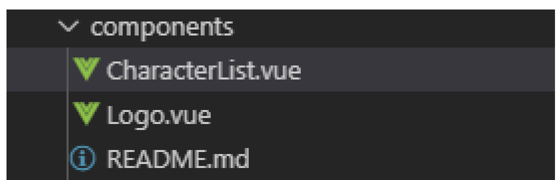
foobar 戦闘ルール 20番トークルーム フリー掲示板123番スレッド foobar Google (外部リンク)

<nuxt-link>は他のHTML要素と同様にidやclassを付与しスタイルを適用することができます。ただし<nuxt-link>は内部的にいろいろ処理された後に最終的には<a>タグとして表示されます。そのため、以下のようなスタイル指定は使えません。スタイルを適用する場合は<a>タグとして扱います。

```
<style>
/* これらのスタイル指定は無効 */
nuxt-link {
  color: red;
}

nuxt-link.decoratedlink {
  color: gray;
}
</style>
```

3.5.16 コンポーネントの作成と読み込み

Nuxt.jsでは「components」フォルダー内にvueファイルを作成することでコンポーネントを作ることができます。実際に作ってみましょう。今回は「キャラクターリスト」をコンポーネント化してみます。「components」フォルダー内に「CharacterList.vue」を作ります。以下のようなファイル構成になります。



作ったら以下の内容を記述して保存します。

components/CharacterList.vue

```
<template>
  <section>
    <div class="characterlist_item" v-for="character in characterList" :key="character.eno">
      <span class="name">{{ character.name }}</span>
      <span class="eno">(ENo.{{ character.eno }})</span><br>
      <span class="description">{{ character.description }}</span>
    </div>
  </section>
</template>

<script>
export default {
  data: function() {
    return {
      characterList: [
        {eno: 1, name: "太郎", description: "フルネームは山田太郎"},
        {eno: 2, name: "花子", description: "最近英語で赤点を取った"},
        {eno: 3, name: "三郎", description: "この世の全ての咎を背負う者"}
      ]
    };
  }
}
</script>
```

```
<style>
.characterlist_item {
  width: 600px;
  padding: 8px;
  border-top: 1px solid gray;
}

.characterlist_item:last-child {
  border-bottom: 1px solid gray;
}

.name {
  font-weight: bold;
}

.eno {
  color: #888;
  font-size: small;
}

.description {
  color: #222;
}
</style>
```

次に読み込みの処理を書いてみましょう。コンポーネントはimportで読み込み、componentsで宣言することで使用できるようになります。宣言したコンポーネントは2個目以降の大文字部分を小文字にしてその前に-(ハイフン)を置いたものをタグ名として使うことができます。とはいえ見たほうが分かりやすいと思われるので、例をいくつか用意しました。左が宣言、右が宣言したコンポーネントを使用するためのタグです。(なお、左のような書き方を**アッパーキャメルケース**、右のような書き方を**ケバブケース**と呼びます。)

宣言	タグ
CharacterList	<character-list>
CharacterIcon	<character-icon>
CharacterProfileIllust	<character-profile-illust>

今回は「pages/index.vue」でコンポーネントを読み込んでみます。「pages/index.vue」の内容を以下のように書き換え保存しましょう。

pages/index.vue

```
<template>
  <section>
    キャラクターリスト：
    <character-list></character-list>
  </section>
</template>

<script>
```

```
import CharacterList from '~/components/CharacterList.vue';

export default {
  components: {
    CharacterList
  }
}
</script>

<style>
</style>
```

　これを実行すると以下のようになります。「キャラクターリスト：」と書かれたところに「CharacterList.vue」が読み込まれています。

キャラクターリスト：

太郎 (ENo.1)
フルネームは山田太郎

花子 (ENo.2)
最近英語で赤点を取った

三郎 (ENo.3)
この世の全ての咎を背負う者

　Nuxt.jsでコンポーネントを読み込む際はファイルパスの先頭に~/(チルダ、スラッシュ)を書き込みます。忘れてしまいがちなので注意しましょう。

3.5.17　親から子への値の受け渡し

　キャラクターリストなど、表示する内容をデータによって適宜変える必要がある場合は呼び出し元のコンポーネントから値を受け渡して表示する必要があります。そのような際、Vue.jsではpropsを使うことで値を受け取ることができます。また、受け取るデータ型を指定することもできます。実際にやってみましょう。「components/CharacterList.vue」を以下のように書き換えます。

components/CharacterList.vue

```
<template>
  <section>
    <div class="characterlist_item" v-for="character in characters" :key="character.eno">
      <span class="name">{{ character.name }}</span>
      <span class="eno">(ENo.{{ character.eno }})</span><br>
      <span class="description">{{ character.description }}</span>
    </div>
  </section>
</template>

<script>
export default {
  props: {
```

```
    characters: Array
  }
}
</script>

<style>
.characterlist_item {
  width: 600px;
  padding: 8px;
  border-top: 1px solid gray;
}

.characterlist_item:last-child {
  border-bottom: 1px solid gray;
}

.name {
  font-weight: bold;
}

.eno {
  color: #888;
  font-size: small;
}

.description {
  color: #222;
}
</style>
```

次に、「pages/index.vue」を以下のように書き換え保存します。

```
pages/index.vue
<template>
  <section>
    キャラクターリスト：
    <character-list :characters="charactersParental"></character-list>
  </section>
</template>

<script>
import CharacterList from '~/components/CharacterList.vue';

export default {
  components: {
    CharacterList
  },
  data() {
    return {
      charactersParental: [
        {eno: 4, name: "スミス", description: "剣専門の鍛冶職人"},
        {eno: 5, name: "マリー", description: "柔道で黒帯を取っている"},
        {eno: 6, name: "ジョー", description: "無類のB級映画好き"}
      ]
    };
```

```
  }
}
</script>

<style>
</style>
```

するとページが以下のようになります。

キャラクターリスト：

スミス (ENo.4)
剣専門の鍛冶職人

マリー (ENo.5)
柔道で黒帯を取っている

ジョー (ENo.6)
無類のB級映画好き

どうなっているのか実際にコードの流れを追っていきましょう。まず、「pages/index.vue」のdata()内でキャラクターリストのデータcharactersParentalが定義されています。charactersParentalは<template>の<character-list>タグ内でcharacters属性として値が渡されます。「components/CharacterList.vue」ではcharacters属性で渡された値を受け取り表示しています。このようにしてVue.jsでは値を受け取って表示することができます。

また見出し用のコンポーネントを作りたいといった場合などにタグで囲む形式のコンポーネントを作りたい場合もあるでしょう。そのような場合には<slot></slot>を使用します。<slot>内にタグで囲まれた内容が挿入されます。試しに見出し用のコンポーネントを作ってみましょう。「components/SubHeading.vue」を作成し以下の内容を入力して保存します。

components/SubHeading.vue

```
<template>
  <section>
    <h2 class="subheading">
      <slot></slot>
    </h2>
  </section>
</template>

<script>
export default {}
</script>

<style>
.subheading {
  margin: 10px;
  width: 300px;
  border-bottom: dotted 2px gray;
}
</style>
```

これを呼び出してみます。「pages/index.vue」の内容を以下のように書き換え保存します。

pages/index.vue

```
<template>
  <section>
    <sub-heading>キャラクターリスト</sub-heading>
    <character-list :characters="charactersParental"></character-list>
  </section>
</template>

<script>
import CharacterList from '~/components/CharacterList.vue';
import SubHeading from '~/components/SubHeading.vue';

export default {
  components: {
    CharacterList,
    SubHeading
  },
  data() {
    return {
      charactersParental: [
        {eno: 4, name: "スミス", description: "剣専門の鍛冶職人"},
        {eno: 5, name: "マリー", description: "柔道で黒帯を取っている"},
        {eno: 6, name: "ジョー", description: "無類のB級映画好き"}
      ]
    };
  }
}
</script>

<style>
body {
  margin :10px;
}
</style>
```

ページの内容が以下のようになります。<sub-heading>タグで囲んだ内容がSubHeading.vueに渡されスタイルが反映されているのが分かります。

キャラクターリスト

スミス (ENo.4)
剣専門の鍛冶職人

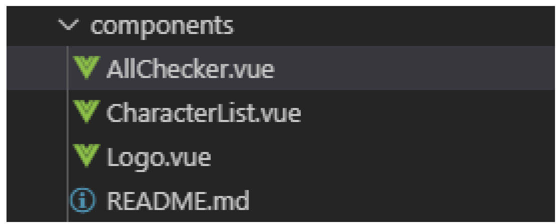
マリー (ENo.5)
柔道で黒帯を取っている

ジョー (ENo.6)
無類のB級映画好き

なお、<slot>タグを使わないコンポーネントは「<character-list></character-list>」とするのではなく、「<character-list/>」というようにすることでも呼び出すことができます。本書では<slot>を使用しない場合後者の書き方を採用します。

3.5.18 カスタムイベント（子から親への受け渡し）

Vue.jsでは$emitを使うことでオリジナルのイベントを発生させることができます。発生させたオリジナルのイベントは他のイベントと同様に@（v-on）で受け取ることができます。値を渡すことも可能で、Vue.jsでは子から親への受け渡しにはカスタムイベントを利用します。実際にやってみましょう。チェックボックスを3つ持ち、その全てがチェックされたら「allchecked」というカスタムイベントを発生させチェックされた数を渡す「components/AllChecker.vue」を作成します。以下のようなファイル構成になります。



作成したら以下の内容を記述し保存します。

components/AllChecker.vue

```
<template>
  <div>
    <input type="checkbox" v-model="check1" @change="checkAllChecked">
    <input type="checkbox" v-model="check2" @change="checkAllChecked">
    <input type="checkbox" v-model="check3" @change="checkAllChecked">
  </div>
</template>

<script>

export default {
  data() {
    return {
      check1: false,
      check2: false,
```

```
      check3: false
    };
  },
  methods: {
    checkAllChecked: function() {
      if (this.check1 == true && this.check2 == true && this.check3 == true) {
        // 全てにチェックが入っていたら
        // 補足：this.check1 && this.check2 && this.check3とも書ける

        this.$emit('all-checked', 3);
        //allcheckedイベントを発生させチェックされた数である3を返す
      }
    }
  }
}
</script>

<style>
</style>
```

「pages/index.vue」も以下のように書き換えて保存しましょう。

pages/index.vue

```
<template>
  <section>
    全部チェックしてみてね：
    <all-checker @all-checked="onAllChecked"></all-checker>
  </section>
</template>

<script>
import AllChecker from '~/components/AllChecker.vue';

export default {
  components: {
    AllChecker
  },
  methods: {
    onAllChecked: function(value) {
      alert(`${value}つ全てチェックされました！`);
      // 全てチェックされたらチェックされた数を受け取りアラート表示
    }
  }
}
</script>

<style>
</style>
```

保存するとページが以下のようになります。指示通りチェックボックスを全てチェックすると「3つ全てチェックされました！」と表示されます。

全部チェックしてみてね：

☐ ☐ ☐

プログラムの流れを追っていきましょう。まず「components/AllChecker.vue」で3つチェックボックスが定義されていてそれぞれ変更されたときにcheckAllCheckedを呼び出します。checkAllCheckedではチェックボックス全てにチェックされいるか確認し、その全てがチェックされていたなら「this.$emit('all-checked', 3);」でチェックされている数(3)とともにイベント「all-checked」を発生させます。「pages/index.vue」では「all-checked」イベントが発生したときにonAllCheckedを呼び出しています。これによってカスタムイベントと値を受け取り、「3つ全てチェックされました！」と表示しているのです。

なお、Vue.jsではカスタムイベント名はケバブケースにすることが推奨されています。HTMLは大文字と小文字を区別しないため、大文字を使うと予期せぬエラーが発生しうるためです。

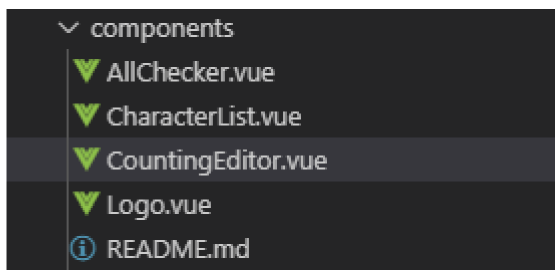
3.5.19 v-modelを使った子から親への値の受け渡し

例えば文字数をカウントしたり指定の文字列を簡単に入力したりできるオリジナルのテキストボックスのコンポーネントを作りたいとします。そのような場合<input>などと同様にv-modelを使うことができると非常に便利です。

作る前にまずはv-modelが内部的にどのようなことを行っているかを学びましょう。v-modelは以下のように書き換えることができます。

```
<input v-model="foobar">
<!-- v-modelの普通の書き方 -->

<input :value="foobar" @input="foobar = $event.target.value">
<!-- これは上と同じ意味になる -->
```

v-modelは内部的に後者の書き方に変換されています。後者を詳しく見てみると、対象のコンポーネントにvalueとして変数を渡し、inputイベントが起きたときに$event.target.valueで値を受け取って変数に値を代入しています。この仕様を満たすコンポーネントを作れば、すなわちvalueで値を受け取り、値が変更された際にinputイベントで新しい値を送信するコンポーネントを作ればv-modelを使うことができるというわけです。それでは実際に作ってみましょう。今回は文字数をカウントしたりボタンを使って簡単に「<1d100>」が入力したりできるエディターを作ってみましょう。「components/CountingEditor.vue」を作成します。ファイル構成は以下のようになります。



作成したら以下の内容を入力して保存します。

```
components/CountingEditor.vue
<template>
  <div>
    <button @click="insert1d100">1d100を挿入</button>
    <textarea ref="countingEditor" class="countingeditor" v-model="innerValue"></textarea>
    現在の文字数：{{ value.length }}文字
  </div>
</template>

<script>
export default {
  props: {
    value: String
  },
  methods: {
    insert1d100() {
      const pos = this.$refs["countingEditor"].selectionStart;
      const val = this.innerValue;
      this.innerValue = val.substr(0, pos) + '<1d100>' + val.substr(pos);
    }
  },
  computed: {
    innerValue: {
      get() {
        return this.value;
      },
      set(newValue) {
        this.$emit('input', newValue);
      }
    }
  }
}
</script>

<style>
.countingeditor {
  width: 400px;
  height: 120px;
  resize: none;
}
</style>
```

「pages/index.vue」も以下のように書き換えて保存します。

```
pages/index.vue
<template>
  <section>
    <counting-editor v-model="editorText"/>
    CountingEditorの内容：{{ editorText }}
  </section>
</template>

<script>
```

```
import CountingEditor from '~/components/CountingEditor.vue';

export default {
  components: {
    CountingEditor
  },
  data() {
    return {
      editorText: 'ここに内容を入力'
    };
  }
}
</script>

<style>
</style>
```

実行すると以下のようになります。テキストボックスの中身を編集してみたりボタンを押したりしてみてください。

今回は少し複雑ですが、コードの流れを追っていきましょう。まず「pages/index.vue」でeditorTextが定義され、v-modelでCountingEditorと紐付けられます。この部分は内部的に以下のように変換されます。

```
<counting-editor :value="editorText" @input="editorText = $event.target.value"/>
```

:valueの指定によってvalueとしてeditorTextの値が「components/CountingEditor.vue」に渡されます。CountingEditor側ではpropsの指定によりvalueを受け取ります。CountingEditorでは双方向算出プロパティによって「innerValue」が定義されています。innerValueはgetされるとvalueの値を返しますが、値をセットしようとした際は新しい値とともにinputイベントを発火します。

inputイベントは「pages/index.vue」側で内部的に変換された後の「@input」の指定により値を受け取ってeditorTextに反映させます。これで変更されると同時にeditorTextにも変更が適用されるようにもなります。editorTextに値が反映されている様子は「CountingEditorの内容」から確認することができます。

またeditorTextとCountingEditorのthis.valueは同じ変数を指しているので、this.valueが示す値も同時に更新されていますし、this.valueを参照するinnerValueのgetで参照できる値も同様に更新されています。これによりボタンをクリックしたときにthis.innerValueで今の値を参照し、新しい値を反映させることができるというわけです。

3.5.20 Vue.jsのライフサイクル

Vue.jsのコンポーネントは内部で生成され、表示され、データの変更があったら再表示され、必要なくなったら破棄されます。この一連の流れのことを**ライフサイクル**と呼びます。Vue.jsではライフサイクル中特定のタイミングで処理を実行することができます。この処理タイミングを**ライフサイクルフック**と呼び、以下のように指定します。

```
<template>
  <section></section>
</template>

<script>
export default {
  beforeCreate: function() { ... },
  created: function() { ... },
  beforeMount: function() { ... },
  mounted: function() { ... },
  beforeUpdate: function() { ... },
  updated: function() { ... },
  beforeDestroy: function() { ... },
  destroyed: function() { ... }
}
</script>

<style>
</style>
```

それぞれ以下のタイミングで処理されます。

beforeCreate	要素が作成される直前	まだ要素は作成されていないのでdata()にはアクセスできない
created	要素が作成された直後	ここからdata()にアクセスできるようになる
beforeMount	要素が表示される直前	
mounted	要素が表示された直後	
beforeUpdate	データが更新され要素が再表示される直前	
updated	データが更新され要素が再表示された直後	
beforeDestroy	要素が破棄される直前	
destroyed	要素が破棄された直後	

なお、エラーの原因となるためライフサイクルフックの処理はアロー関数ではなくfunction()で宣言する必要があります。

3.5.21 SCSSの利用 / CSSのスコープ

Nuxt.jsでは指定のライブラリをインストールすることで簡単にSCSSを利用することができるようになります。インストールするためにまずはNuxt.jsの実行を停止しましょう。コンソール上でCtrl+Cを押すことで実行を停止することができます。コマンドを入力できるようになるので、「npm install node-sass sass-loader」を実行しましょう。イ

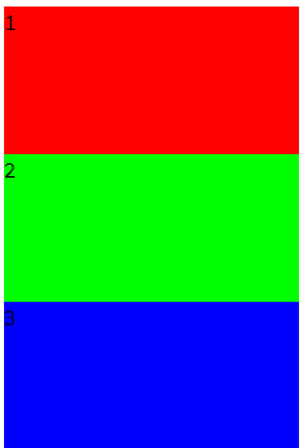
ンストールが完了したら再び「npm run dev」でNuxt.jsを実行します。(もしこのときにENOSPCエラーが出る場合は「4.1.8 補足:ENOSPCエラーが出る場合」を参照してください。)

実行したら「pages/index.vue」を以下のように書き換え保存しましょう。

pages/index.vue

```
<template>
  <section>
    <div id="foobar">
      <div>1</div>
      <div>2</div>
      <div>3</div>
    </div>
  </section>
</template>

<script>
export default {}
</script>

<style lang="scss" scoped>
$innerDivWidth: 200px;
$innerDivHeight: 100px;
$firstDivColor: #FF0000;
$secondDivColor: #00FF00;
$thirdDivColor: #0000FF;

#foobar {
  div {
    width: $innerDivWidth;
    height: $innerDivHeight;

    &:nth-child(1) {
      background: $firstDivColor;
    }

    &:nth-child(2) {
      background: $secondDivColor;
    }

    &:nth-child(3) {
      background: $thirdDivColor;
    }
  }
}
</style>
```

一度接続が切れたので手動で「https://dev.siroisakana.com/test/」を表示しているタブをリロードして再度接続しましょう。表示されるページの内容は以下のようになります。しっかりとSCSSが反映されています。



Nuxt.jsではこのようにライブラリをインストールした上で<style>に「lang="scss"」を指定するだけでSCSSが利用できるようになります。また、Nuxt.jsにはスコープという機能があります。例示のように「scoped」を指定することによって指定のコンポーネント以下の要素にのみCSSが反映されるようになります。クラス名の衝突をあまり気にしなくてよくなるのでつけておくといいでしょう。

3.5.22 assets

Nuxt.jsではassetsフォルダー内に画像やCSSファイルを入れておくことで<img>で読み込んだりスタイルを全体に適用したりすることができます。例えば「assets/img/foobar.png」というように画像を入れておいたとしましょう。その場合は以下のようにして画像を読み込むことができます。

```
<template>
  <section>
    <img src="~/assets/img/foobar.png">
  </section>
</template>

<script>
export default {}
</script>

<style>
</style>
```

CSSを適用する場合はアップロードして「nuxt.config.js」に設定を追記することで全体にスタイルを適用できます。また、SCSSを扱うためのライブラリをインストールしておくことでSCSSも利用できます。例えば「assets/css/theme.scss」というようにSCSSファイルを入れていたとすると、「nuxt.config.js」に以下のように記述することでスタイルを全体に適用することができます。

nuxt.config.js

```
（省略）
```

```
  css: [
    '~/assets/css/theme.scss'
  ],

 （省略）
```

3.5.23 layouts

Nuxt.jsでは全体に適用したいレイアウトがある場合layoutsを利用します。デフォルトでは「layouts/default.vue」のレイアウトが適用されています。layouts内ではページの内容は<nuxt />というコンポーネントとして扱われます。レイアウトを変更してみましょう。「layouts/default.vue」の内容を以下のように変更し保存します。

layouts/default.vue

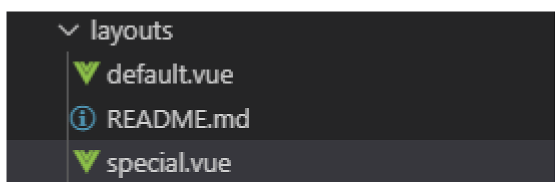
```
<template>
  <div>
    全体ヘッダー
    <hr>
    <nuxt />
  </div>
</template>

<style>
</style>
```

この状態で「http://dev.siroisakana.com/test/」や「http://dev.siroisakana.com/test/foobar」にアクセスしてみましょう。以下のようなページになっており、全体にレイアウトが適用されていることが確認できます。

全体ヘッダー

ここはfoobar.vue。

あるページだけ別のレイアウトを適用したい場合はレイアウト用のvueファイルを新しく記述し、ページに適用したいレイアウトを設定します。「layouts/special.vue」を作りましょう。ファイル構成は以下のようになります。



作ったら以下の内容を記述して保存します。

layouts/special.vue

```
<template>
  <div>
    スペシャルレイアウト
```

```
    <hr>
    <nuxt />
  </div>
</template>

<style>
</style>
```

作ったレイアウトを「pages/foobar.vue」に適用してみます。「pages/foobar.vue」を以下のように書き換えます。

pages/foobar.vue

```
<template>
  <section>
    ここはfoobar.vue。
  </section>
</template>

<script>
export default {
  layout: 'special'
}
</script>

<style>
</style>
```

「http://dev.siroisakana.com/test/foobar」にアクセスするとページが以下のようになっており、レイアウトが反映されていることがわかります。

スペシャルレイアウト

ここはfoobar.vue。

3.5.24 ページタイトルの設定

Nuxt.jsではheadを使うことでページごとの<head>を設定することができます。これによりページタイトルをページごとに変えることができます。「pages/foobar.vue」を以下のように書き換えましょう。dataのように関数で指定します。

pages/foobar.vue

```
<template>
  <section>
    ここはfoobar.vue。
  </section>
</template>

<script>
export default {
  head() {
```

```
    return {
      title: 'ふーばー'
    }
  }
}
</script>

<style>
</style>
```

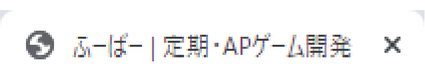
この例のようにするとページのタイトルが「ふーばー」に変わります。

また、実際に定期・APゲームを作るときは「ルールブック | Teiki Adventure」というようにページタイトルに共通タイトルを入れたいこともあるでしょう。Nuxt.jsでは「nuxt.config.js」にtitleTemplateを指定することで全ページに共通のタイトルを入れることもできます。titleTemplateではページごとに変えたい部分を「%s」で指定します。「nuxt.config.js」のhead部分を以下のように変更してください。変更した部分は斜体で表示してあります。

nuxt.config.js

```
（省略）

head: {
  title: '無題',
  titleTemplate: '%s | 定期・APゲーム開発',
  meta: [

（省略）
```

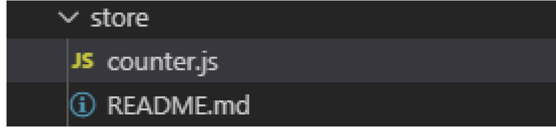
するとページタイトルが以下のようになります。このように設定するとページ名を設定していないページでは「無題 | 定期・APゲーム開発」、設定してあるページでは以下のように表示されます。



3.5.25 Vuex

Vuexは先述の通り状態管理ライブラリで、これを使うことでコンポーネントに依存しないデータや使おうとするとコンポーネントの親子関係を多く辿らなければならないデータなどを上手く管理することができます。Vuexではデータやデータに関する処理を「State」「Mutations」「Actions」の3つに分けて考えます。「State」ではデータの実体を管理し、「Mutations」でデータへの操作方法を定義し、「Actions」でMutationsを通してデータを操作するための関数を定義します。また、データを取得したりするための「Getter」もあります。

Nuxt.jsではVuexを「state」フォルダー内で記述します。カウンターを作ってみましょう。「state/counter.js」を作成します。ファイル構成は以下のようになります。



作成したら以下の内容を記述して保存します。

store/counter.js

```
const state = () => ({
  // 初期値0でカウンターの値、countを宣言
  count: 0
});

const getters = {
  value(state) {
    // カウンターの値を取得する
    return state.count;
  }
};

const mutations = {
  countup(state) {
    // カウントを+1する
    state.count++;
  },
  reset(state) {
    // カウントをリセットし0にする
    state.count = 0;
  },
  set(state, value) {
    // カウンターに任意の値をセットする
    state.count = value;
  }
};


const actions = {
  countup({ commit }) {
    // MutationsのCountupを呼び出す
    commit('countup');
  },
  reset({ commit }) {
    // Mutationsのresetを呼び出す
    commit('reset');
  },
  set({ commit }, value) {
    //Mutationsのsetを呼び出し値を渡す
    commit('set', value);
  }
};
```

```
export default {
  state,
  getters,
  mutations,
  actions
}
```

カウンターを呼び出してみましょう。Nuxt.jsでは$storeでVuexにアクセスできます。$store.getters['ファイル名/gettersの宣言名']で値を取得し、$store.dispatch('ファイル名/actionsの宣言名')でactionsを呼び出すことができます。(methods内ではthis.$storeというようにします。)実際にやったほうが分かりやすいと思うのでやってみましょう。「pages/index.vue」を以下のように書き換えます。

pages/index.vue

```
<template>
  <section>
    カウンターの値：{{ $store.getters['counter/value'] }}
    <button @click="alert">カウンターの値を表示</button>
    <button @click="countup">カウントアップ</button>
    <button @click="reset">リセット</button>
    <button @click="set">カウンターの値を5に</button>
  </section>
</template>

<script>
export default {
  methods: {
    alert: function() {
      alert(`カウンターの値は${this.$store.getters['counter/value']}です。`);
      // counter.jsのgetterからvalueを参照
    },
    countup: function() {
      this.$store.dispatch('counter/countup');
      // counter.jsのactionsからcountupを呼び出す
    },
    reset: function() {
      this.$store.dispatch('counter/reset');
      // counter.jsのactionsからresetを呼び出す
    },
    set: function() {
      this.$store.dispatch('counter/set', 5);
      // counter.jsのactionsからsetを呼び出し値を渡す
    }
  }
}
</script>

<style>
</style>
```

するとページの内容が以下のようになります。それぞれのボタンを押すとVuexのgettersやactionsが呼び出されカウンターの値を表示したり変えたりできます。

カウンターの値：0 [カウンターの値を表示] [カウントアップ] [リセット] [カウンターの値を5に]

重要なのはこれが「pages/index.vue」でもそれ以外のページでも、他のコンポーネントからでも呼び出すことができるということです。これによってコンポーネントに依存しないデータを管理することができます。例えばログイン情報などを管理するときに非常に役立つでしょう。

3.5.26 middleware

middlewareを使うことで特定のページを開く際に前処理を行うことができます。例えばログインが必要なページにアクセスしようとした際にログインページにリダイレクトしたりできます。この処理を書くなら以下のようになるでしょう。ログインの仕組みを作っていないのでこの項目の例については実際に書かなくても構いません。こういうことができるという風に把握しておいてください。

middleware/authenticated.js

```
export default function ({ store, redirect }) {
  if (!store.getters['auth/isAuthenticated']) {
    return redirect('/login');
  }
}
```

以上の内容が「middleware/authenticated.js」に保存されているとします。これを適用したいページ、つまりログインを必要とするページには以下のように設定します。これでこのページにはログインしないとアクセスできなくなります。（正確には表示自体はされませんがページのvueファイルはクライアントに送信されます。セキュリティ目的でこれを使うことはできないでしょう。）

```
<template>
  <section>
    ここはログインが必要なページ。
  </section>
</template>

<script>
export default {
  middleware: 'authenticated'
}
</script>

<style>
</style>
```

3.5.27 plugins

pluginsを使うことでその名の通りプラグインを読み込んだり、その他にもアクセスしてページが表示される前の

前処理を実装したりすることができます。これを使うことで前ログインしていたときの接続情報を持っておき、再度アクセスした際に自動でログインするといった仕組みを作ることができます。この処理を書くなら以下のようになるでしょう。Vuexのactionsでログインの仕組みを組んでいることを想定しています。なお、ログインの仕組みを作っていないのでこの項目の例については実際に書かなくても構いません。こういうことができるという風に把握しておいてください。

plugins/autoLogin.js
```
export default async (context) => {
  await context.store.dispatch('auth/login');
}
```

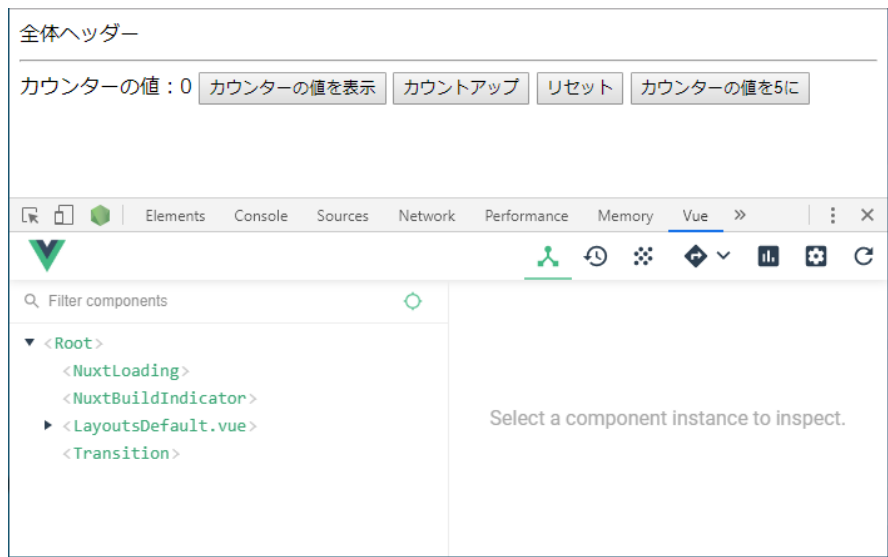
記述したプラグインは「nuxt.config.js」で読み込むことで有効化できます。pluginsという場所に読み込むファイルを記述します。上の例が「plugins/autoLogin.js」に記述されていたとすると以下のようになります。

nuxt.config.js
```
  (省略)

  plugins: [
    '~/plugins/autoLogin.js'
  ],

  (省略)
```

3.5.28 Vue.js devtoolsの使い方

ここでは『Google Chromeの拡張機能のインストール』でインストールしたChrome拡張機能である「Vue.js devtools」の使い方について学びます。「https://dev.siroisakana.com/test/」にアクセスしてみましょう。これより前の手順で気づいていた方もいるかもしれませんが、Nuxt.jsを利用したページを開くとVue.js devtoolsのアイコンがこのように変化します。

この状態でGoogle Chromeの開発者ツールを開きましょう。開発者ツールはF12キーを押すと開くことができます。開発者ツールメニュー上にVueという項目があるので選択します。(ない場合、»の内側に隠れこんでいる場合があります。)以下のような内容が開きます。



Vue.js devtoolsを使うことでコンポーネントやページのdataなどの内容、Vuexの内容や発生したactions、発生したカスタムイベントを確認したりすることができ、デバッグに非常に役立ちます。

3.5.29 公式ガイド

Vue.jsは詳細な日本語ガイドが公式で用意されています。読むには本書の想定読者レベルより少し高いレベルの知識が要求されますが、あまり分からないとしてもどのような機能があるのか把握しておく意味で一度目を通しておくといいでしょう。Vue.jsの日本語ガイドは以下のURLよりアクセスできます。

はじめに — Vue.js
https://jp.vuejs.org/v2/guide/

Nuxt.jsにも公式日本語ガイドが用意されています。Vue.jsの公式ガイドを読んだ後、こちらにも一度目を通しておいた方がいいでしょう。Nuxt.jsの日本語ガイドは以下のURLよりアクセスできます。

はじめに - Nuxt.js
https://ja.nuxtjs.org/guide

3.6 バックエンドの基礎

3.6.1　バックエンドとは

Webアプリ開発においてバックエンドとはフロントエンド側で利用するAPIを提供したり、データベースにアクセスしてデータの書き込みや読み込みなどを行ったりなど、主にサーバー側に関連する部分のことを指します。定期・APゲームにおいてはログイン/ログアウト機構の提供や、チャットや戦闘システムの実装部分などがバックエンド部分に該当します。

3.6.2　リバースプロキシの追加

バックエンドAPIを公開するためのリバースプロキシを追加しましょう。Tera Termを起動しrootでログインします。「vi /etc/nginx/conf.d/default.conf」を開き、「location /test/」となっている所の下に以下のように内容を追記しましょう。追記した内容は分かりやすいよう斜体で表示しています。なお、上側に追加してしまうと実行順の関連でうまくAPIにアクセスすることができません。必ず下に追記しましましょう。

/etc/nginx/conf.d/default.conf

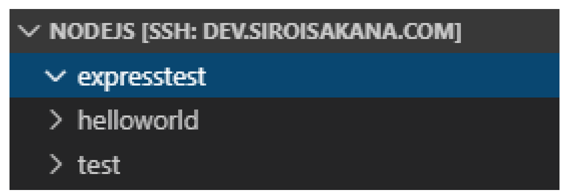
```
(省略)

 location / {
     root   /usr/share/nginx/html;
     index  index.html index.htm;
 }

 location /test/ {
     proxy_pass  http://127.0.0.1:3000;
 }

 location /test/api/ {
     proxy_pass  http://127.0.0.1:4000;
 }

(省略)
```

追記したらコマンドモードで「:wq」を入力し上書きして保存してください。「nginx -s reload」を実行すると設定を反映することができます。

3.6.3　Expressの基礎

ExpressはWebアプリケーションのフレームワークです。これを使うことで簡単にWebサーバーアプリケーションを作ってアクセスを待ち受けることができます。Expressを使うためにプロジェクトを作成しましょう。まだNuxt.jsを実行中の場合はコンソールにCtrl+Cを押してNuxt.jsを終了させてください。

まずはプロジェクトを入れるフォルダーを作成します。Visual Studio Codeのエクスプローラーメニューの「NODEJS [SSH: DEV.SIROISAKANA.COM]」となっているところのなにもない空間を右クリックし「新しいフォルダー」を選択します。フォルダー名は「expresstest」とします。ファイル構成は以下のようになります。



フォルダーを作成したらVisual Studio Codeのコンソールで「cd /home/teiki/nodejs/expresstest」を実行し作成したディレクトリに移動します。（Nuxt.jsを実行し続けている場合は終了してください。）次に「npm init」を実行してください。いろいろと項目が聞かれますが、ほとんどそのままエンターで進んで構いません。licenseと書かれているところではソースコードのライセンスを指定できるのですが、プライベートなプロジェクトでオープンソースにする気がない場合は「UNLICENSED」と入力しておきましょう。なお、聞かれている内容は以下のとおりです。

package name	パッケージ名
version	バージョン
description	プロジェクトの説明
entry point	最初に実行されるファイル
test command	npm testとしたときに実行されるコマンド
git repository	プロジェクトを管理するGitリポジトリ
keywords	プロジェクトを表すキーワード郡
author	作者名
license	プロジェクトに適用するライセンス

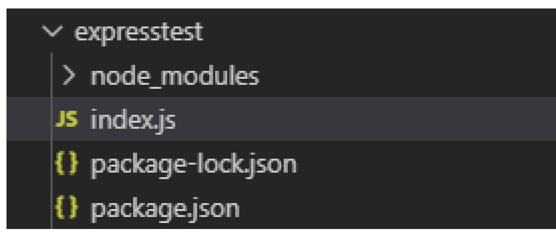
全て入力したら「Is this ok?」と聞かれるので「yes」と入力し処理を終えましょう。「packages.json」が自動的に作成されます。次に必要なライブラリをインストールします。「npm install express body-parser」を実行します。インストールが終わったら設定変更しましょう。「expresstest/packages.json」を開き、"scripts"の部分を以下のように書き換え保存します。分かりやすいよう変更した部分は斜体で表示しています。これでこのディレクトリで「npm run start」をすることで「index.js」が実行されるようになります。

packages.json

```
（省略）

"main": "index.js",
"scripts": {
  "start": "node index.js"
},
"author": "",

（省略）
```

3.6.4　ルーティングとレスポンス

それでは実際にアクセスを受け取ってみましょう。まずは「index.js」を作ります。以下のようなファイル構成になります。



そして以下の内容を記述して保存します。

index.js

```
const express = require('express');
const app = express();
const port = 4000;

app.get('/test/api/', (req, res) => {
  console.log('/test/api/へのアクセスがありました。');
  res.send('Hello, Express World!');
});

app.listen(port);
```

ここまで書いたらコンソールから「npm run start」を実行しましょう。コードが実行されアクセスを待ち受けるようになるので「http://dev.siroisakana.com/test/api/」にアクセスしてみます。以下のようなページが表示され、コンソールに「/test/api/へのアクセスがありました。」と表示されます。この例ではapp.getでアクセスを受け取る宣言をし、console.logでアクセスがあった際にコンソールにログを表示、res.sendでアクセスへ応答（レスポンス）しています。

Hello, Express World!

また、宣言を増やすことでルーティングを増やすことができます。「/test/api/foobar」でもアクセスを受け取るようにしてみましょう。（以下、このような受け取り先のことをAPIと呼称します。）「index.js」に以下のようにして内容を追記して保存します。分かりやすいよう追記した内容を斜体で表示しています。

index.js

```
（省略）

app.get('/test/api/', (req, res) => {
  console.log('/test/api/へのアクセスがありました。');
  res.send('Hello, Express World!');
});
```

```
app.get('/test/api/foobar', (req, res) => {
  console.log('/test/api/foobarへのアクセスがありました。');
  res.send('ここは/test/api/foobar。');
});

app.listen(port);
```

保存したらプログラムを再起動しましょう。コンソールにCtrl+Cを入力して実行を中断し、「npm run start」を入力して再度実行します。「http://dev.siroisakana.com/test/api/foobar」にアクセスしてみましょう。以下のようなページが表示され、コンソールに「/test/api/foobarへのアクセスがありました。」と表示されます。

ここは/test/api/foobar。

post、put、deleteなどget以外のリクエストも以下のようにして受け取ることができます。

```
app.post('/test/api/foobar', (req, res) => {});
app.put('/test/api/foobar', (req, res) => {});
app.delete('/test/api/foobar', (req, res) => {});
```

また、Expressではrouterを使うことであるURL以下のAPIへのアクセスをまとめて受け取ることができます。「index.js」を以下のように書き換えて保存しましょう。

index.js

```
const express = require('express');
const app = express();
const port = 4000;

const router = express.Router();

router.get('/', (req, res) => {
  res.send('Hello, Express World!');
});

router.get('/foobar', (req, res) => {
  res.send('ここは/test/api/foobar。');
});

app.use('/test/api', router);

app.listen(port);
```

プログラムを再起動しましょう。前の例と同様に「http://dev.siroisakana.com/test/api/」、「http://dev.siroisakana.com/test/api/foobar」にアクセスすることができるようになっています。

routerはexpress.Router();で宣言し、利用したいURL以下のアドレスのみを指定してAPIを宣言、app.use('r

outerを利用したいURL', router);というようにすることで実現することができます。この例ではAPIが2つだけなのであまりメリットを実感できないかもしれませんが、数が増えてくるとAPIのURLの変更が大変になってしまいます。routerを使うとそのような状況を解決することができたり、ファイルを分割してわかりやすく管理することもできたりします。またrouter.useでまた別のrouterをつけることもでき、これによって多重ルーターを構成することも可能です。

3.6.5　ミドルウェア関数

Expressではミドルウェア関数を使うことで複数のAPIへのアクセスへの前処理を実現することができます。実際にやってみましょう。「index.js」を以下のように書き換え保存します。

index.js

```
const express = require('express');
const app = express();
const port = 4000;

const logging = (req, res, next) => {
  console.log(`${req.originalUrl}へのアクセスがありました。`);
  next();
};

const router = express.Router();

router.get('/', logging, (req, res) => {
  res.send('Hello, Express World!');
});

router.get('/foobar', logging, (req, res) => {
  res.send('ここは/test/api/foobar。');
});

app.use('/test/api', router);

app.listen(port);
```

保存したらプログラムを再起動しましょう。APIにアクセスすると「/test/api/へのアクセスがありました。」「/test/api/foobarへのアクセスがありました。」というようにログがコンソールに表示されるようになります。

処理の流れを追ってみましょう。ミドルウェア関数としてlogging関数が宣言されています。logging関数ではリクエストされたURLへのアクセスがあったことをconsole.logで表示していて、next()で処理を続ける宣言をしています。logging関数はrouter.get('/', logging, ...)というようにAPI側で利用宣言され、これにより利用宣言したAPIにアクセスがあったときにミドルウェア関数が実行されるようになっています。

ミドルウェア関数ではこのようにnext()で前処理が終わったときに処理を次へと渡すことができます。逆に言うと、next()を実行せずにres.sendでレスポンスを返してしまうことで処理を次に移さずに応答してしまうことができます。これを利用することでログインしていないユーザーがトークルームなどログインの必要なページにアクセスできないようにしたりすることができます。

ミドルウェアはAPIごとに設定するだけではなくルーターごとやアプリ全体に適用することも可能です。また複数

のミドルウェアを設定することもでき、複数設定した場合は先に設定した方から処理されます。それぞれ以下のように行います。

```
// loggingとprecheckという2つのミドルウェア関数があるとします
// APIに複数のミドルウェアを適用する例
router.get('/', logging, precheck, (req, res) => {});

// ルーターに適用する例
router.use(logging);
router.use(precheck);

// アプリ全体に適用する例
app.use(logging);
app.use(precheck);
```

3.6.6 エラーコード

何かエラーが発生したときや非ログインユーザーがログインユーザー限定のAPIへアクセスしようとした場合などにエラーコードを返したい場合があるでしょう。そのような際Expreesではres.status(404).send()というようにすることで**HTTPステータスコード**を返すことができます。HTTPステータスコードとはレスポンスの意味を表現するための3桁の数字でできたコードです。「404 Not Found」などは非常に有名なので知っている方が多いのではないでしょうか。この例では「404」がHTTPステータスコードです。HTTPステータスコードでは200番台が成功、300番台がリダイレクト、400番台がクライアント側に問題があるエラー、500番台がサーバー側に問題があるエラーを意味します。主要なHTTPステータスコードには以下のようなものがあります。

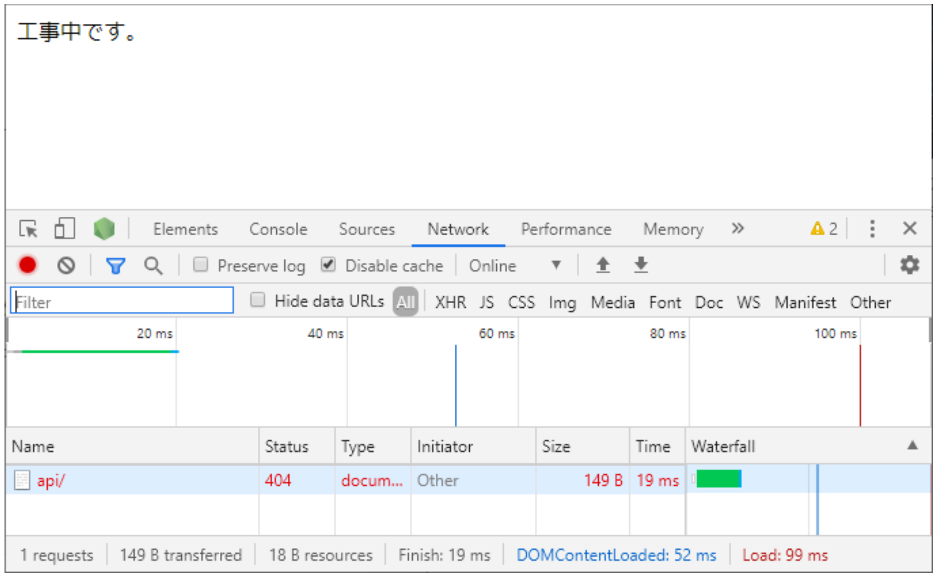
コード	内容	意味
200	OK	成功。Expressではデフォルトで200が送信されている。
204	No Content	内容なし。成功だが返す内容が存在しない。
301	Moved Permanently	恒久的に移動したという意味だが、今日ではリダイレクトに使われる。
304	Not Modified	未更新。前のアクセスから内容が更新されていない。
400	Bad Request	リクエストが不正。リクエストの内容がおかしい。
401	Unauthorized	認証が必要。ログインが必要なのにログインしていないなど。
403	Forbidden	禁止されている。アクセス権限がないなど。
404	Not Found	リソースが見つからない。アクセスしようとしたファイルが存在しない。
500	Internal Server Error	サーバー内部エラー。サーバー側で何かしらエラーが発生した。
502	Bad Gateway	不正なゲートウェイ。リバースプロキシ先が動作していないなど。
503	Service Unavailable	サービス利用不可。メンテナンスや過負荷で一時的に利用不可。
504	Gateway Timeout	ゲートウェイタイムアウト。リバースプロキシ先からの応答がないなど。

Expressでは以下のようにしてエラーコードを送信することができます。「index.js」を以下のように書き換え保存

してください。以下の例では404と同時に「工事中です。」という内容を送信しています。

```
index.js
const express = require('express');
const app = express();
const port = 4000;

app.get('/test/api/', (req, res) => {
  res.status(404).send('工事中です。');

  // 404 Not Found と一緒に「工事中です。」という内容を送信。
  // res.status(404).send();とすることでステータスコードだけ送信することもできる。
});

app.listen(port);
```

プログラムを再起動し「http://dev.siroisakana.com/test/api/」にアクセスします。「工事中です。」というページが表示されます。これでは404が送信されているかどうか分からないので、F12で開発者ツールを開いてNetworkタブを開きF5でリロードしてみましょう。すると以下のように404で送信されているのが確認できます。



3.7 データベースの基礎

3.7.1　データベースとは

データベースとは扱いやすいように情報をまとめたもの、またそれを扱うためのソフトウェアのことを指します。データベースではいろいろな処理が最適化されており、適切にデータベースを組むことができれば検索などの処理を自分でアルゴリズムを組むよりはるかに高速に行うことができます。また堅牢性、整合性にも優れています。そのため、データを保存したり検索したりする場合はデータベースにデータ管理を任せるのが一般的です。

3.7.2　SQLとNoSQL

データベースには主に**SQL**と**NoSQL**の2種類があります。SQLはSQL文と呼ばれるコマンドによってデータベースを管理するデータベースのことです。有名なものにはMySQLやMariaDBなどがあります。安定性、堅牢性などに非常に優れ企業のプロダクトなどにもよく使われるデータベースですが、習得に高めの学習コストがかかったり速度の面で難があったりする場合があります。

NoSQLはそれに当てはまらないデータベースのことを指します。なお、「No+SQL」ではなく「Not only SQL」の略となっています。NoSQLには様々なものがあります。JavaScriptのオブジェクトをそのまま放り込むことができるデータベースや、超高速に動作する代わりに電源を切るとデータが全て消えるデータベースなどがあります。

3.7.3　MongoDBとは

今回使用するMongoDBはNoSQLの一種で、基本的なデータ型のみで構成されたJavaScriptのオブジェクトをそのままデータベースに入れて管理することができます。そのため習得が非常に容易です。ある程度高速に動作しますが、その代わりにデータの堅牢性などはSQLに劣ります。定期・APゲームには商業ソフトウェアのような厳密性は求められないため、今回はMongoDBを使用します。（なお、MongoDBでもやろうと思えば厳密性のある処理（ACID）を実現することはできるのですが定期・APゲームにそのレベルの厳密性は必要ないと思われるため本書では取り扱いません。興味のある方は各自で調べてみてください。）

Node.jsからMongoDBを操作する際に本書ではmongooseというライブラリを使用します。mongooseを使うことでより簡単にMongoDBを操作できるようになります。

3.7.4　データベースの作成

MongoDBを学習するにあたってまずは学習用のデータベースとそのデータベースの管理ユーザーを作成しましょう。Expressを実行している場合はCtrl+Cで終了しコンソールに「mongo --port 34189」を入力します。するとMongoDBのコンソールに移行します。

まずは「use admin」と入力して管理者用データベースに接続します。次に「db.auth("admin", "q%sv|N#IE99i")」を実行しMongoDBに管理者でログインします。次に作りたいデータベースに移行します。今回はtestというデータベースを作りましょう。「use test」を実行しtestデータベースに接続します。次にtestデータベース用のユーザーを作成しましょう。ここではユーザー名は「test」、パスワードは「Uj$bIiB6Dy-F」にします。ただし、MongoDBはインターネットからアクセスできるようになっているので、パスワードは必ず変更してください。読み替えやすいよう背景色をこのように変えてあります。以下のコマンドを実行します。

```
db.createUser({user: "test", pwd:"Uj$bIiB6Dy-F", roles:[{role:"readWrite",  db:"test"}]})
```

これでtestデータベースとtestデータベースの管理ユーザーが作成されます。作成したら「exit」を入力するかCtrl+Cを押してMongoDBのコンソールから離脱しましょう。

```
> use admin
switched to db admin
> db.auth("admin", "q%sv|N#IE99i")
1
> use test
switched to db test
> db.createUser({user: "test", pwd:"Uj$bIiB6Dy-F", roles:[{role:"readWrite",  db:"test"}]})
Successfully added user: {
        "user" : "test",
        "roles" : [
                {
                        "role" : "readWrite",
                        "db" : "test"
                }
        ]
}
> exit
bye
```

3.7.5　スキーマ

MongoDBでは登録するデータのことを**ドキュメント**、ドキュメントに入っている値のことを**フィールド**と呼び、ドキュメントをまとめたものを**コレクション**と呼びます。ドキュメントにはJavaScriptのオブジェクトをそのまま入れられるのですが、同じコレクションにも関わらずデータの構造が異なるのは好ましくありません。

そこでmongooseではスキーマと呼ばれるものを定義することでMongoDBに入れるオブジェクトの構造を指定します。以下のようにしてスキーマを定義します。

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const CharacterSchema = new Schema({
  name: String, // キャラクター名
  eno: Number, // ENo
  registrationDate: Date, // 登録日時
  tags: [String], // キャラクタータグ
  fav: [Schema.Types.ObjectId] // お気に入りしているキャラ
});
```

スキーマでは**インデックス**を貼ることができます。インデックスとはデータベースの検索の際に使われる索引のようなもので、適切にインデックスを貼ることで検索を非常に高速化することができます。上の例ではENoはよく検索されると思われるのでインデックスを貼るべきでしょう。また、デフォルト値も指定することができます。このようにオプションを指定する際は以下のようにします。

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const CharacterSchema = new Schema({
  name: String, // キャラクター名
  eno: {type: Number, index: true}, // ENo
  registrationDate: {type: Date, default: Date.now}, // 登録日時
  tags: [String], // キャラクタータグ
  fav: [Schema.Types.ObjectId] // お気に入りしているキャラ
});
```

このようにオプションを指定する際にはオブジェクト形式で指定し、typeに設定したい型を設定します。指定できる型には主に以下のようなものがあります。

String	文字列
Number	数値
Boolean	真偽値
Date	時刻
Array	配列 [String]というようにすることでどの型の配列なのか指定可能
ObjectId	他のドキュメントのオブジェクトID Schema.Types.ObjectIdというようにアクセスする

また、指定できるオプションには主に以下のようなものがあります。

type	型
index	インデックスを貼るかどうか
require	データ作成時必須にするかどうか 使用する場合明示的にindex: falseとしない限りインデックスが貼られる
ref	ObjectIdで参照するスキーマ
match	正規表現を使い設定できる文字列の内容を指定
minlength/maxlength	設定できる文字列の長さの最小値/最大値を指定
unique	ユニークインデックスを貼るかどうか (値の重複を許さないインデックス)
sparse	スパースインデックスを貼るかどうか (空の値を除き、値の重複を許さないインデックス)

なお、1ドキュメントあたりのデータの長さは最大16MBまでという制限があります。そのため、定期・APゲームでトークルームを用意して発言内容をトークルームの配列に保存するといったことは難しいでしょう(ではどうやって保存するのかはゲーム制作編で解説します。)

3.7.6　mongooseを使ったMongoDBへの接続

mongooseでデータ操作を行う場合、まずスキーマからモデルを作成しモデルから操作するようにします。またデータベースへの接続も必要です。以下のように行います。

```
const Character = mongoose.model('Character', CharacterSchema);
mongoose.connect(`mongodb://test:${encodeURIComponent('Uj$bIiB6Dy-F')}@127.0.0.1:34189/test`);
```

このように「mongodb://(利用するデータベースのユーザー名):(パスワード)@(アドレス):(MongoDBのポート)/(利用するデータベース)」というようにURL形式でデータベースにアクセスします。パスワードに記号が含まれている場合はencodeURIComponentを使って記号をURLでも使える形式に変換しておきます。

3.7.7　データ保存

さて、モデルの作成方法とデータベースへの接続方法を学んだ所で実際にデータベースを使ってみましょう。まずはデータ保存の方法について学びます。プロジェクトは「expresstest」をそのまま利用してしまいましょう。まずはmongooseをインストールします。「npm install mongoose」を実行しましょう。mongooseがインストールされたら「index.js」の内容を以下のように変更します。

index.js

```
const express = require('express');
const mongoose = require('mongoose');
const app = express();
const port = 4000;

const Schema = mongoose.Schema;

const CharacterSchema = new Schema({
  name: String, // キャラクター名
  eno: {type: Number, unique: true}, // ENo
  registrationDate: {type: Date, default: Date.now}, // 登録日時
  tags: [String], // キャラクタータグ
  fav: [Schema.Types.ObjectId] // お気に入りしているキャラ
});

const Character = mongoose.model('Character', CharacterSchema);
mongoose.connect(`mongodb://test:${encodeURIComponent('Uj$bIiB6Dy-F')}@127.0.0.1:34189/test`);

//---------- ここからAPIの宣言、これ以前は以降の解説では省略 ----------//

app.get('/test/api/create', async (req, res) => {
  const obj = { // 保存するオブジェクトを作成
    name: '太郎',
    eno: 1,
    tags: ['超能力者', 'リンクフリー']
  };

  const doc = new Character(obj); // オブジェクトからドキュメントを作成
  try {
    await doc.save(); // ドキュメントを保存
```

```
    res.send('ENo.1 太郎が登録されました。'); // 保存されたらメッセージを返す
  } catch (e) {
    res.status(400).send('すでにENo.1は登録されています。');
    // すでに登録している場合400とともにメッセージを返す
  }
});

//---------- ここまでAPIの宣言、これ以降は以降の解説では省略 ----------//

app.listen(port);
```

「npm run start」でプログラムを実行し、「http://dev.siroisakana.com/test/api/create」にアクセスしてみましょう。「ENo.1 太郎が登録されました。」というメッセージが帰ってくるはずです。これでデータベースにENo.1 太郎というキャラクターが登録されます。なお、enoはunique属性をつけているのでもう一度アクセスしたら「すでにENo.1は登録されています。」と表示されるようになります。同じenoでは重複登録できないためです。

さて、では処理の流れを追っていきましょう。まずスキーマが宣言され、モデルが作成され、MongoDBへ接続されています。「/test/api/create」でアクセスを受け取るようになっており、その中で保存するオブジェクトが作成されています。new Character(obj);でオブジェクトからドキュメントが作成され、await doc.save();でドキュメントが保存されています。データベースへのアクセスは非同期処理なのでawaitで処理を待つようになっています。(そのためAPIの関数宣言もasyncになっています。またこれ以降紹介するデータベースへのアクセスメソッドも全て非同期処理なのでawaitを使用します。)そして保存が終わったら「ENo.1 太郎が登録されました。」というメッセージを返します。

また、Characterのスキーマではenoがunique: trueに指定されています。これによりenoの値が重複できなくなるので、すでにENo.1の登録が終わっている場合await doc.save();でエラーが発生するようになります。そのためtry-catchでエラーを受け取り、エラーが発生した場合は「すでにENo.1は登録されています。」というメッセージを返すようになっています。

なおこれ以降も「index.js」を使い回すのですが特に断りのない限りこの『データベースの基礎』の章ではコメントにも書いてあるとおりAPIの宣言以外の部分を省略します。あらかじめご了承ください。

さて、これ以降の項目で使うためのデータも登録してしまいましょう。「index.js」のAPI宣言部分を以下のように書き換え保存します。

index.js

```
//---------- ここから API の宣言、これ以前は省略されています ----------//

app.get('/test/api/create', async (req, res) => {
  const characterObjs = [
    {
      name: '花子',
      eno: 2,
      tags: ['リンクフリー']
    },
    {
      name: '三郎',
      eno: 3,
      tags: ['超能力者']
    },
```

```
    {
      name: 'スミス',
      eno: 4,
      tags: ['職人', 'リンクフリー']
    },
    {
      name: 'マリー',
      eno: 5,
      tags: ['柔道']
    },
    {
      name: 'ジョー',
      eno: 6,
      tags: ['サメ映画', 'リンクフリー']
    },
  ]

  try {
    for (let i = 0; i < characterObjs.length; i++) {
      await (new Character(characterObjs[i])).save();
    }
    res.send('テスト用データが登録されました。');
  } catch(e) {
    res.send('すでにテスト用データは登録済みです。');
  }
});

//---------- ここまで API の宣言、これ以降は省略されています ----------//
```

プログラムを再起動して「http://dev.siroisakana.com/test/api/create」にアクセスします。「テスト用データが登録されました。」と表示されればテスト用データの登録が完了しています。これでデータベースに以下のようなデータが登録されます。（登録日時はデータ登録処理の実行日時によって変わります。）

name	eno	tags	registrationDate	fav
太郎	1	['超能力者', 'リンクフリー']	2019-11-16T23:20:44.637Z	[]
花子	2	['リンクフリー']	2019-11-17T09:22:46.760Z	[]
三郎	3	['超能力者']	2019-11-17T09:22:46.760Z	[]
スミス	4	['職人', 'リンクフリー']	2019-11-17T09:22:46.760Z	[]
マリー	5	['柔道']	2019-11-17T09:22:46.760Z	[]
ジョー	6	['サメ映画', 'リンクフリー']	2019-11-17T09:22:46.760Z	[]

指定したデータが入っている他、スキーマの「default: Date.now」の指定によってregistrationDateも自動的に作成されています。なお、登録される日付は**日本標準時**（JST）ではなく**協定世界時**（UTC）基準なので時刻を扱う場合は気をつけましょう。JSTはUTC+09:00なので、日本で時刻を扱う場合は登録されている時間+9時間が日本での時間になります。

また、MongoDBでは上の表の内容に加えて管理のために「_id」というフィールドが自動的に作成されていま

す。「_id」フィールドは後々使うことになるのでそういうものが存在するということを覚えておきましょう。（他にも「__v」というフィールドも作られているのですが、こちらについてはあまり気にしなくても構いません。）

3.7.8　データ検索

　データの検索には様々な方法があります。基本的なものはfindで、モデルからfindを呼び出すことで条件に当てはまるものを全て検索することができます。以下のように検索します。「index.js」のAPI宣言部分を以下のように書き換えて保存しましょう。

index.js

```
//---------- ここからAPIの宣言、これ以前は省略されています ----------//

app.get('/test/api/read', async (req, res) => {
  const characters = await Character.find({eno: 2});
  res.send(characters);
});

//---------- ここまでAPIの宣言、これ以降は省略されています ----------//
```

　プログラムを再起動して「http://dev.siroisakana.com/test/api/read」にアクセスしてみましょう。以下のようにデータが取得できます。なお、当てはまるものをすべて検索するという都合上、キャラクターデータそのままではなくキャラクターデータが入った配列が取得されます。findを使うとデータが入った「配列」が帰ってくるということは忘れがちなので覚えておいてください。

[{"tags":["リンクフリー"],"fav":[],"_id":"5dd11166b242451195bbd697","name":"花子","eno":2,"registrationDate":"2019-11-17T09:22:46.760Z","__v":0}]

　また、配列に指定の要素を含んでいるドキュメントを検索する場合もこれと同様に行えます。以下の例ではtagsに「リンクフリー」を含むドキュメントを検索しています。

index.js

```
app.get('/test/api/read', async (req, res) => {
  const characters = await Character.find({tags: 'リンクフリー'});
  res.send(characters);
});
```

　これを実行すると以下のようになります。見づらいですが、たしかにtagsに「リンクフリー」が含まれているドキュメントが検索できていることがわかります。

[{"tags":["超能力者","リンクフリー"],"fav":[],"_id":"5dd0844cf65e130ca76f3d6b","name":"太郎","eno":1,"registrationDate":"2019-11-16T23:20:44.637Z","__v":0},{"tags":["リンクフリー"],"fav":[],"_id":"5dd11166b242451195bbd697","name":"花子","eno":2,"registrationDate":"2019-11-17T09:22:46.760Z","__v":0},{"tags":["職人","リンクフリー"],"fav":[],"_id":"5dd11166b242451195bbd699","name":"スミス","eno":4,"registrationDate":"2019-11-17T09:22:46.773Z","__v":0},{"tags":["サメ映画","リンクフリー"],"fav":[],"_id":"5dd11166b242451195bbd69b","name":"ジョー","eno":6,"registrationDate":"2019-11-17T09:22:46.777Z","__v":0}]

　条件は複数指定することもできます。複数指定した場合AND検索（全ての条件を満たすものを検索）になります。以下の例では名前が「太郎」かつENoが「1」のドキュメントを検索しています。

```
index.js
app.get('/test/api/read', async (req, res) => {
  const characters = await Character.find({name: '太郎', eno: '1'});
  res.send(characters);
});
```

他にも、条件を指定しないことで全てのドキュメントデータを取り出すことも可能です。以下の例ではCharacterに保存されている全てのデータを検索しています。

```
index.js
app.get('/test/api/read', async (req, res) => {
  const characters = await Character.find({});
  res.send(characters);
});
```

また、特定の要素を含まないドキュメントを検索したい場合もあるでしょう。今度は逆に「リンクフリー」が含まれていないドキュメントを検索してみます。

```
index.js
app.get('/test/api/read', async (req, res) => {
  const characters = await Character.find({tags: {$nin: 'リンクフリー'}});
  res.send(characters);
});
```

これを実行すると以下のようになります。今度は「リンクフリー」がtagsに含まれていないドキュメントのみを検索できています。

[{"tags":["超能力者"],"fav":[],"_id":"5dd11166b242451195bbd698","name":"三郎","eno":3,"registrationDate":"2019-11-17T09:22:46.771Z","__v":0},{"tags":["柔道"],"fav":[],"_id":"5dd11166b242451195bbd69a","name":"マリー","eno":5,"registrationDate":"2019-11-17T09:22:46.775Z","__v":0}]

この例では「{tags: {$nin: 'リンクフリー'}}」というようになっており、これは「tagsに'リンクフリー'を含まない」という意味になります。mongooseでは$ninのような演算子を使うことでさまざまな検索を実現することができます。主な演算子には以下のようなものがあります。

演算子	意味	使用例
$and	全ての条件を満たす	`{$and: [{tags:'サメ映画'}, {tags:'リンクフリー'}]}`
$or	どれかの条件を満たす	`{$or: [{eno: 2}, {tags:'リンクフリー'}]}`
$gt	～より大きい	`{eno: {$gt: 3}}`
$gte	～以上	`{eno: {$gte: 3}}`
$lt	～より小さい	`{eno: {$lt: 3}}`
$lte	～以下	`{eno: {$lte: 3}}`

$ne	～と等価でない	{eno: {$ne: 3}}
$not	条件を満たさない	{eno: {$not: {$gte: 3}}}
$in	～を含む	{tags: {$in: 'リンクフリー'}}
$nin	～を含まない	{tags: {$nin: 'リンクフリー'}}
$exists	フィールドが存在する	{eno: {$exists: true}}

また、ドキュメントの内容ではなく条件に合うドキュメントの件数が知りたい場合はfindの代わりにcountを使用します。

index.js

```
app.get('/test/api/read', async (req, res) => {
  const characterCount = await Character.count({tags: 'リンクフリー'});
  res.send(`リンクフリーがタグ付けされているキャラクターは${characterCount}人います。`);
});
```

プログラムを再起動してアクセスすると「リンクフリーがタグ付けされているキャラクターは4人います。」と表示され、たしかに条件に合うドキュメントの件数が取得できていることが確認できます。

その他にもfindの代わりにfindOneを使うことで検索結果のうち1つだけを取得することができます。例えば特定のENoの検索など、検索結果が1つであることが保証されているような検索ではこちらを使うといいでしょう。

index.js

```
app.get('/test/api/read', async (req, res) => {
  const character = await Character.findOne({eno: 4});
  res.send(character);
});
```

この例では以下のような結果になります。findOneでは結果が1つだけになるのでドキュメントの配列ではなくドキュメントそのものが帰ってきます。

{"tags":["職人","リンクフリー"],"fav":[],"_id":"5dd11166b242451195bbd699","name":"スミス","eno":4,"registrationDate":"2019-11-17T09:22:46.773Z","__v":0}

3.7.9 プロジェクション

多くの場合ドキュメントの全てのフィールドを使うことはなく、使用するのは一部のフィールドのみです。そのような場合、使用するフィールドを絞り込むことでデータが扱いやすくなったり検索が高速化したりしますこれをMongoDBではプロジェクションと呼びます。プロジェクションはfindやfindOneなどの2つ目の引数に指定し、利用するフィールドに「1」を指定します。それではやってみましょう。例えば名前とENoのみを使うのであれば以下のようにします。以下の例ではtagsに超能力者を含むドキュメントを検索し、名前とENoのみを取り出しています。

index.js

```
app.get('/test/api/read', async (req, res) => {
  const characters = await Character.find({tags: '超能力者'}, {name: 1, eno: 1});
  res.send(characters);
});
```

プログラムを再起動してURLにアクセスすると以下のようになります。

```
[{"_id":"5dd0844cf65e130ca76f3d6b","name":"太郎","eno":1},{"_id":"5dd11166b242451195bbd698","name":"三郎","eno":3}]
```

名前とENo、そして「_id」のみが表示されていることが分かります。特別に指定されない限り「_id」は自動的にプロジェクションに含まれます。「_id」を含めたくない場合「_id: 0」というように明示的に「_id」をプロジェクションに含めないことを指定します。

index.js

```
app.get('/test/api/read', async (req, res) => {
  const characters = await Character.find({tags: '超能力者'}, {_id: 0, name: 1, eno: 1});
  res.send(characters);
});
```

このようにすると以下のように「_id」が検索結果に含まれなくなります。

```
[{"name":"太郎","eno":1},{"name":"三郎","eno":3}]
```

3.7.10 複雑な検索

mongooseではfindにsort、skipなどの関数をくっつけることでより複雑な条件で検索できたり、並べ替えたりすることができます。実際に書いてみましょう。「index.js」のAPI宣言部分を以下のように書き換えます。

index.js

```
app.get('/test/api/read', async (req, res) => {
  const characters =
    await Character.find({}).sort({eno: -1}).skip(2).limit(2).select({_id: 0, eno: 1, name: 1}).exec();
  res.send(characters);
});
```

プログラムを再起動してアクセスしてみましょう。以下のようになるはずです。

```
[{"name":"スミス","eno":4},{"name":"三郎","eno":3}]
```

プログラムの流れを追っていきましょう。まずCharacter.find({})でCharacterの全てのデータが検索されます。そしてsort({eno: -1})でENo逆順ソートされ、skipで検索結果のうち最初の2つがスキップされ、limitで検索結果

が残ったうち最初の2つだけに絞り込まれ、selectでプロジェクションが指定され、exec()で実行されます。データはENo.が1，2，3，4，5，6という順番で登録されているので、これらの操作により最終的にENo.4とENo.3のドキュメントのENoと名前がこの順番で出力されています。このように関数をつなげて処理するような書き方のことを**メソッドチェーン**と呼びます。findに繋げることができるメソッドには主に以下のようなものがあります。

メソッド	意味	使用例
skip	検索結果を〜件飛ばす	.skip(10)
limit	〜件だけ検索する	.limit(5)
select	プロジェクションを指定する	.select({eno: 1})
lean	検索結果を純粋なJavaScriptオブジェクトに変換する	.lean()
exec	メソッドチェーンを実行する	.exec()

3.7.11 データ更新

mongooseではfindOneAndUpdateを使うことで変更したいデータを検索して更新することができます。試しにENo.1のキャラクターの名前を「太郎 Mk-2」に変えてみましょう。以下のように行います。以下の例では「http://dev.siroisakana.com/test/api/update」にアクセスすると実行されます。

index.js

```
app.get('/test/api/update', async (req, res) => {
  const character = await Character.findOneAndUpdate({eno: 1}, {name: '太郎 Mk-2'}, {new: true});
  res.send(character);
});
```

実行すると以下のようになります。ENo.1のキャラクターの名前が「太郎 Mk-2」に変わっています。

{"tags":["超能力者","リンクフリー"],"fav":[],"_id":"5dd0844cf65e130ca76f3d6b","name":"太郎 Mk-2","eno":1,"registrationDate":"2019-11-16T23:20:44.637Z","__v":0}

findOneAndUpdateでは1つ目の引数に検索条件、2つ目の引数に更新したい内容を指定します。また、3つ目の引数に{new: true}を指定することで更新された後のドキュメントを受け取ることができます。({new: true}を指定しない場合は更新前のドキュメントが戻り値になります。)

単純に代入するのではなく入っていた値に+1したり配列に要素を追加したりしたい場合もあるでしょう。mongooseでは検索演算子などと同じように更新用の演算子を使うことで特別な更新処理を行うことができます。例えばtagsに「改造済み」という要素を追加する処理は以下のように書くことができます。

index.js

```
app.get('/test/api/update', async (req, res) => {
  const character =
    await Character.findOneAndUpdate({eno: 1}, {$addToSet: {tags: '改造済み'}}, {new: true});
  res.send(character);
});
```

実行すると以下のようになり、tagsに「改造済み」が追加されていることが確認できます。

{"tags":["超能力者","リンクフリー","改造済み"],"fav":[],"_id":"5dd0844cf65e130ca76f3d6b","name":"太郎 Mk-2","eno":1,"registrationDate":"2019-11-16T23:20:44.637Z","__v":0}

データ更新時に使える演算子には以下のようなものがあります。

演算子	意味	使用例
$inc	加算	`{$inc: {ap: 1}}`
$addToSet	重複していなければ配列に追加	`{$addToSet: {tags: '改造済み'}}`
$push	配列に追加	`{$push: {tags: '改造済み'}}`
$each	〜に対して処理をそれぞれ繰り返す	`{$push: {tags: {$each: {['宇宙人', '火星人']}}}}`
$pop	-1で配列の先頭、1で末尾を削除	`{$pop: {tags: -1}}`
$pull	配列から条件を満たす値を削除	`{$pull: {tags: {$in: ['柔道', '職人']}}}`

また更新処理はモデルから呼び出すだけでなく検索結果のドキュメントから呼び出すことも可能です。受け取れる値は変わってしまいますが、最初の例は以下のようにも書くことができます。

index.js

```
app.get('/test/api/update', async (req, res) => {
  const character = await Character.findOne({eno: 1});
  const result = await character.update({name: '太郎 Mk-2'});
  res.send(result);
});
```

実行すると以下のような結果になります。ドキュメントからupdateを呼び出した場合、ドキュメントの内容ではなく処理が正常に実行されたかどうかなど更新処理の実行結果が帰ってきます。なお「n」が更新の対象になった件数、「nModified」が実際に更新された件数、「ok」が正常に実行されたかを表します。この例では更新対象件数が1件であり、実際に更新された件数は0件で(すでに名前が「太郎 Mk-2」になっているので更新されなかった)、正常終了したということを表します。

{"n":1,"nModified":0,"ok":1}

ただしドキュメントからのupdateには注意点があり、プロジェクションで「_id」を明示的に取得しないようにして

いる場合updateが使えなくなってしまいます。これはドキュメントからの操作は内部的に「_id」を使用しているためです。ドキュメントからの操作を行いたい場合には「_id」の取得を無効化しないようにしましょう。

　他にもmongooseではfindOneAndUpdateの代わりにupdateManyを実行することで複数のドキュメントに一度に更新処理を行うことができます。以下の例ではENoが3以下のキャラクターのタグに「日本人」を追加しています。

index.js

```
app.get('/test/api/update', async (req, res) => {
  const result = await Character.updateMany({eno: {$lte: 3}}, {$addToSet: {tags: '日本人'}});
  res.send(result);
});
```

　実行すると「{"n":3,"nModified":3,"ok":1}」というように帰ってきます。updateManyではドキュメントからのupdate同様に実行結果のみが帰ってくるのです。ちなみにそれぞれのデータを取得してみると以下のようになり、それぞれのタグに「日本人」が追加されていることがわかります。

[{"tags":["超能力者","リンクフリー","改造済み","日本人"],"eno":1},{"tags":["リンクフリー","日本人"],"eno":2},{"tags":["超能力者","日本人"],"eno":3}]

3.7.12　データ削除

　mongooseではfindOneAndDeleteなどを使うことでドキュメントを削除することができます。以下の例では「http://dev.siroisakana.com/test/api/delete」にアクセスすることでENo.6のキャラクターデータが削除されます。

index.js

```
app.get('/test/api/delete', async (req, res) => {
  const character = await Character.findOneAndDelete({eno: 6});
  res.send(character);
});
```

　しかし、定期・APゲームで利用するにあたってはデータを削除してしまうのは好ましくありません。例えばキャラクターを削除してもトークルームなどでの発言内容は保持しておきたいといった場合にキャラクターデータが完全に削除されてしまっていると、トークルームでキャラクター名を参照しようとしたときにデータが存在しなくてエラーの原因になってしまったりするからです。本書で作る定期・APゲームではデータ削除処理は一切行いません。ではどうやってキャラクター削除などを実現するかというと「deleted」などのフィールドを用意しておき、trueなら削除扱いにしてキャラクターリストなどで表示しない、falseなら削除されていない扱いにする、といったように実現します。

3.7.13　ObjectIdとpopulate

　例えばキャラクターにどのキャラクターをお気に入りしているかという情報を持たせたいなど、他のドキュメントへの参照情報を持たせたい場合があります。そのような場合に使うのがドキュメントに自動的に作られる「_id」というフィールドです。「_id」はObjectId型で表現されます。（スキーマで扱える型の説明のときに少しだけ出てきたSchema.Types.ObjectIdのことです。）MongoDBでは参照する側のドキュメントフィールドに参照される側の「_id」を

設定しておくことで参照情報をもたせることができます。

　ObjectIdが設定されたフィールドはpopulateを使って**正規化**を行うことができます。正規化とは簡単に言ってしまうと参照先の情報を取り出すことです。実際に見たほうが分かりやすいと思われるのでやってみましょう。「index.js」のAPI宣言部分を以下のように書き換えます。以下の例ではENo.4のキャラクターデータのfavにENo.3のキャラクターへの参照を追加し、populateでENo.4のキャラクターデータから情報を取り出しています。

index.js

```
app.get('/test/api/populate', async (req, res) => {
  const characterEno3 = await Character.findOne({eno: 3}, {_id: 1});
  await Character.findOneAndUpdate({eno: 4}, {$addToSet: {fav: characterEno3._id}});
  // ENo4のキャラクターデータのfavにENo3への参照を追加

  const characterEno4 = await Character.findOne({eno: 4}).populate('fav').exec();
  // favを正規化して取り出す

  res.send(characterEno4);
});
```

　プログラムを再起動し「http://dev.siroisakana.com/test/api/populate」にアクセスすると以下のようになり、favの中でENo.3のキャラクター情報が取得できていることがわかります。


```
{"_id":"5dd11166b242451195bbd699","tags":["職人","リンクフリー"],"fav":[{"_id":"5dd11166b242451195bbd698","tags":["超能力者","日本人"],"fav":[],"name":"三郎","eno":3,"registrationDate":"2019-11-17T09:22:46.771Z","__v":0}],"name":"スミス","eno":4,"registrationDate":"2019-11-17T09:22:46.773Z","__v":0}
```


　また、正規化した対象も含めてプロジェクションを行いたい場合には以下のようにします。注意点として正規化される側のドキュメント（今回はENo.4のキャラクター）のプロジェクションを指定する際に「_id」を明示的に無効化していると正規化できなくなってしまいます。

```
index.js
app.get('/test/api/populate', async (req, res) => {
  const characterEno3 = await Character.findOne({eno: 3}, {_id: 1});
  await Character.findOneAndUpdate({eno: 4}, {$addToSet: {fav: characterEno3._id}});
  // ENo4のキャラクターデータのfavにENo3への参照を追加

  const characterEno4 =
    await Character.findOne({eno: 4}, {
      eno: 1,
      name: 1,
      fav: 1,
    }).populate({path: 'fav', select: {
      _id: 0,
      eno: 1,
      name: 1
    }}).exec();
  // favを正規化してプロジェクションし取り出す

  res.send(characterEno4);
});
```

なお正規化を行わない場合ObjectIdのみがそのまま取得されます。これで取得できるObjectIdを使った命令にはfindByIdやfindByIdAndUpdateなどがあり、指定する条件の部分に指定するものがObjectIdになっているだけでfindOneやfindOneAndUpdateと同じです。

```
index.js
app.get('/test/api/populate', async (req, res) => {
  const characterEno4 = await Character.findOne({eno: 4}, {_id: 0, fav: 1});
  // characterEno4: {"fav":["5dd11166b242451195bbd698"]}
  // characterEno4.fav[0]: "5dd11166b242451195bbd698"

  await Character.findById(characterEno4.fav[0]);
  // Character.findOne({id: characterEno4.fav[0]})と同じ

  await Character.findByIdAndUpdate(characterEno4.fav[0], {name: '三郎 Mk-2'});
  // Character.findOneAndUpdate({id: characterEno4.fav[0]}, {name: '三郎 Mk-2'})と同じ

  return res.send(characterEno4);
});
```

MongoDBにとって正規化は比較的重い処理であり、多用してしまうと処理速度低下の原因になってしまいます。軽快に動作する定期・APゲームを作りたい場合、正規化を最小限にしたデータ構造をうまく作り上げることが一番大切です。

3.7.14 取得したドキュメントの編集

mongooseで取得したデータは特殊な型になっており、そのままでは通常のオブジェクトと同じようには扱うことはできません。以下は取得したドキュメントから「_id」プロパティを削除しようとした例ですが、このようにしても削除することはできません。

```js
index.js
app.get('/test/api/populate', async (req, res) => {
  const character = await Character.findOne({eno: 4}, {eno: 1, name: 1});
  //{"_id":"5dd11166b242451195bbd699","name":"スミス","eno":4}

  delete character._id;
  //_idプロパティを削除しようとする

  res.send(character);
  //{"_id":"5dd11166b242451195bbd699","name":"スミス","eno":4}
  //というようになり、_idプロパティが消えていない
});
```

編集したい場合、ドキュメントを通常のオブジェクトに変換します。変換した場合ドキュメントからのupdateなどは使えなくなりますが通常のオブジェクトと同様に扱うことができるようになります。変換は取得したドキュメントから.toObject()を呼び出すことで行えます。この処理は非同期処理ではないのでawaitは必要ありません。

```js
index.js
app.get('/test/api/populate', async (req, res) => {
  const character = await Character.findOne({eno: 4}, {eno: 1, name: 1});
  const characterObj = character.toObject();
  //{"_id":"5dd11166b242451195bbd699","name":"スミス","eno":4}

  delete characterObj._id;
  //_idプロパティを削除しようとする

  res.send(characterObj);
  //{"name":"スミス","eno":4}
  //というようになり、_idプロパティが消えている
});
```

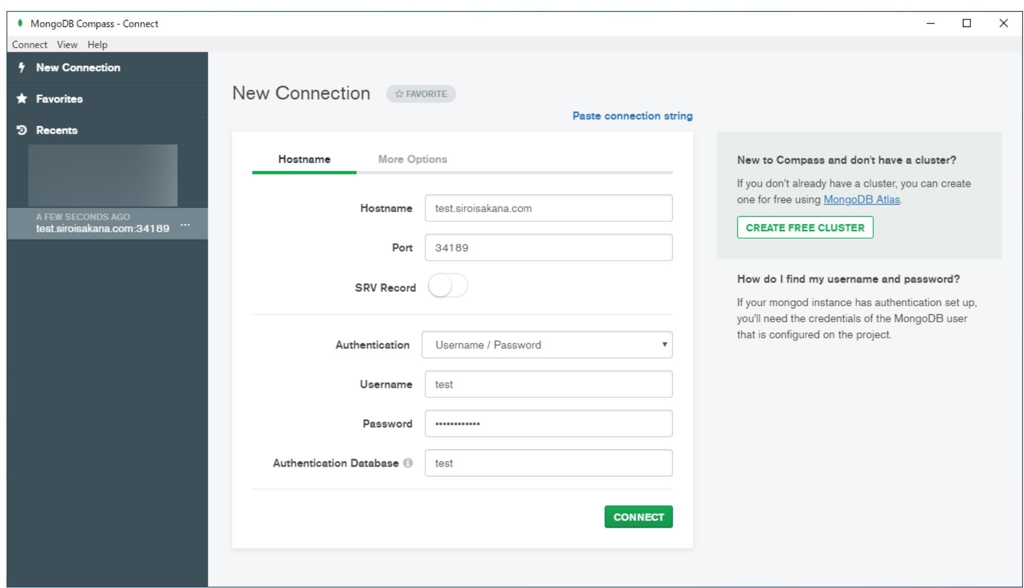
他にもfindなどのメソッドチェーンから.lean()を呼び出すことで最初から普通のオブジェクトとして受け取ることもできます。

さて、これでmongooseの使い方については終了です。ちなみにデータの基本操作であるCreate（新規作成）、Read（データ取得）、Update（データ更新）、Delete（データ削除）のことを総称してCRUDと呼びます。時々見る言い回しなので覚えておくと役に立つこともあるでしょう。

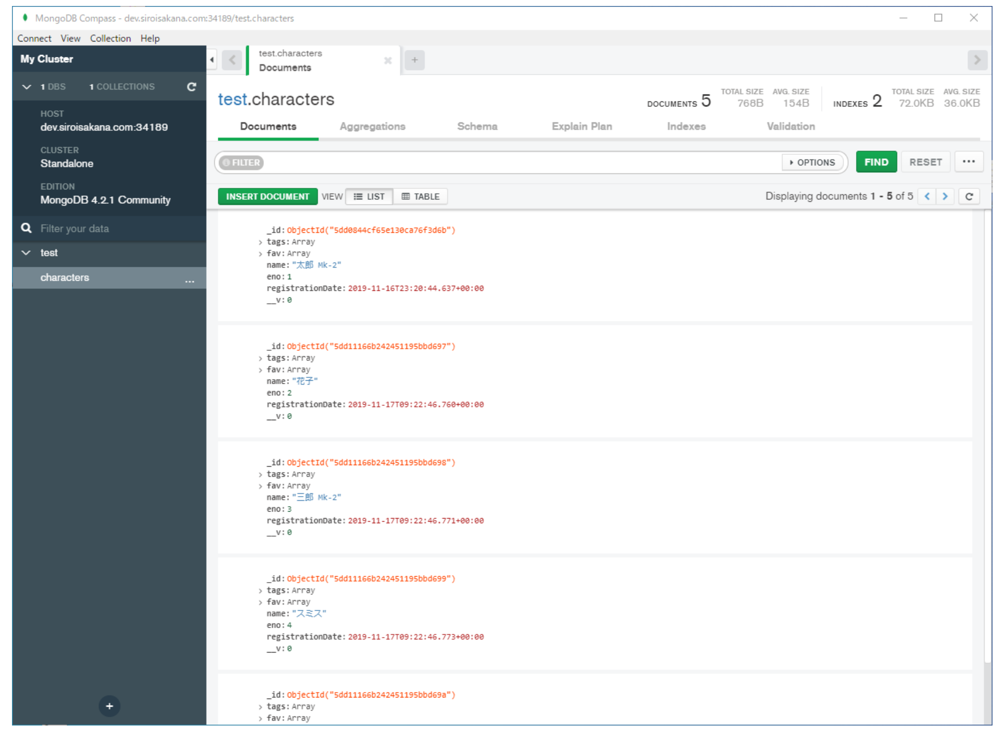
3.7.15 MongoDB Compassの使い方

MongoDB CompassではMongoDBを遠隔操作したり、リセットしたり、データを閲覧したりなどさまざまな操作を行うことができます。まずはデータベースにアクセスしてみましょう。MongoDB Compassを起動します。起動したら「Fill in connection fields individually」と書かれているところをクリックしましょう。接続情報を入力する画面になるので、以下のように入力します。特に記述のない項目はそのままで構いません。

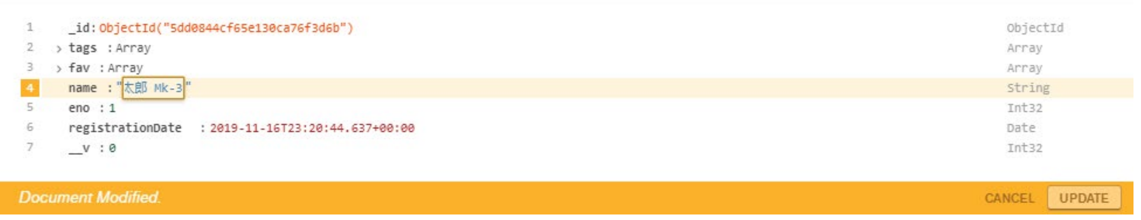
Hostname	dev.siroisakana.com
Port	34189
Authentication	Username / Password
Username	test
Password	Uj$bIiB6Dy-F
Authentication Database	test



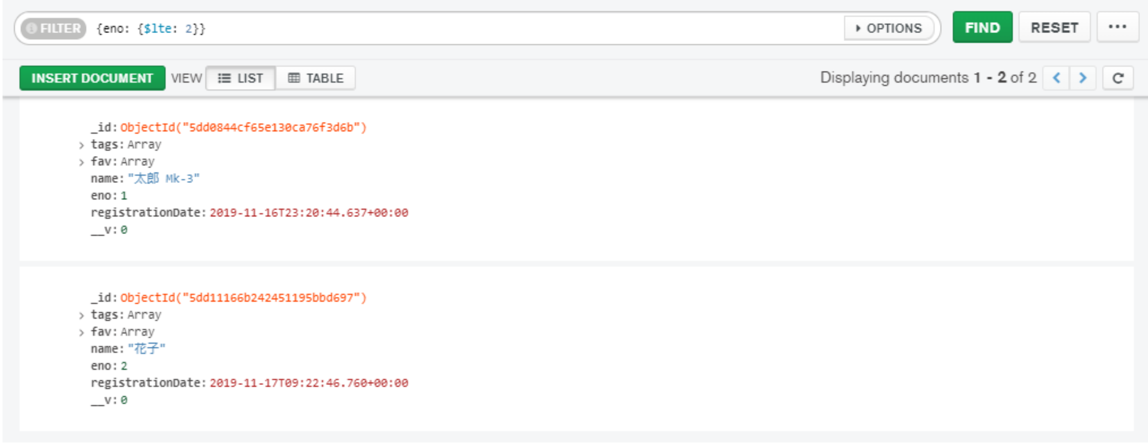
入力したら画面下の「CONNECT」という緑色のボタンをクリックしましょう。データベースに接続されます。接続されたら画面左メニューに「test」というデータベースが表示されるので開き、「characters」を選択してみましょう。テストに使ったキャラクターデータが確認できます。



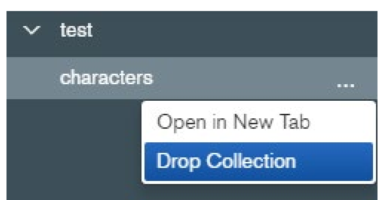
また、ここからデータの内容を編集することも可能であり、編集したいフィールドをダブルクリックすると編集できるようになります。編集したらドキュメント右下の「UPDATE」を選択することで編集内容を適用できます。



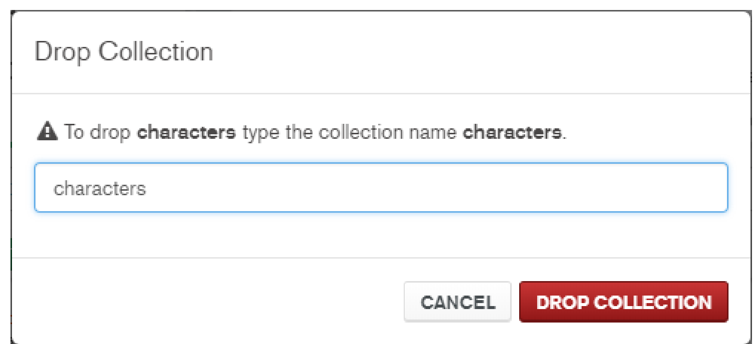
ドキュメントを検索することも可能で、画面上部の「FILTER」とかかれている検索バーからfindに設定する検索条件と同様に検索条件を指定し「FIND」をクリックすることで条件に合うドキュメントを確認できます。



他にもコレクションを削除することもできます。これ以降テスト用データを使うことはないので削除してしまいましょう。コレクションを削除したい場合左メニューから削除したいコレクションを選択し右側の「...」から「Drop Collection」を選択します。



本当に削除していいか確認され、コレクション名の入力を求められます。今回は「characters」を削除するのでそのように入力し右下の「DROP COLLECTION」をクリックします。これでコレクションが削除されます。なお、MongoDBはコレクションなどを利用しようとしたときに自動的にそれらが作成されるので削除はリセットと同義と言えます。内容をリセットしたいときもこれを利用しましょう。



これで基礎学習編は終了です。お疲れさまでした。MongoDB Compassを閉じ、実行している「expresstest」を停止してください。次の章からは学習した内容を使って実際に定期・APゲームを作成していきます。なお、この章で使用した「helloworld」「test」「expresstest」フォルダーはこれ以降使用しないので削除しても構いません。Visual Studio Codeのエクスプローラーメニュー上でフォルダー名を右クリックして「完全に削除」を選択することで削除することができます。

Chapter 4
ゲーム制作編

ここまで学習してきた知識を用いて、
実際に定期・APゲームを制作していきます。

4.1 仕様策定 / プロジェクトの作成

4.1.1 仕様の決定

まず、定期・APゲームを作成するにあたって仕様を決定します。仕事で制作しており納期があるといったわけではないと思われるので行き当たりばったりで制作していくのも悪くはないのですが、どういったゲームにするのか予めある程度決めておいた方が効率的に開発を進めることができます。今回はサンプルということで比較的シンプルな定期更新要素、AP要素が両方あるゲームを作ります。

仕様を決める際には他の定期・APゲームのルールブックを参考にしながらルールブックを実際に書いてみるのがおすすめです。ただし本書ではスペースの関係上文章形式で簡易的に仕様を書いています。どのようなスタイルで書くにせよ、どんな定期・APゲームを作るのかイメージを固めておくことが大切です。

『Teiki Adventure』

ユーザーはオリジナルキャラクターを1人1キャラクター登録し、1日1回配布されるAPで「探索戦」を行い、1週間に1度更新される「物語戦」を進めることでゲームは進行していく。APは最大15まで貯めることができる。「探索戦」はコンプリートには何回かクリアする必要がある。「物語戦」は1回クリアするごとに次の「物語戦」に挑むことができるようになる。それぞれ自キャラクター+お気に入りの中から連れ出した4人の5人PTで攻略する。

「探索戦」や「物語戦」をコンプリートした際には能力値ポイント(NP)が手に入り、それを割り振っていくことでキャラクターを成長させることができる。能力値は「ATK」「DEX」「MND」「AGI」「DEF」の5つに割り振ることができ、割り振りによってキャラクターのHPや攻撃力などが決まる。割り振った能力値が規定の値以上になるとスキルを習得することができるようになり、習得したスキルは5枠のスキルセット枠にセットすることによって戦闘で使えるようになる。

他のキャラクターをお気に入りすることもできる。お気に入りすることでそのキャラクターを連れ出せるようになる。また、交流要素として「トークルーム」「日記」があり、「トークルーム」ではリアルタイムに交流が可能。全員が自由に書き込めるトークルームと個々人が立てることのできる個室のトークルームがある。

「トークルーム」内では「全体」「お気に入り」「関連」「自分」でログをフィルタリングすることができる。それぞれ「全体のログ」「お気に入りキャタクターと自キャラクターのログ」「自キャラクターへの返信を含むログと自キャラクターのログ」「自キャラクターのログ」というフィルター。

「日記」は「物語戦」の更新と同時に公開される。指定の書式を使うことで装飾したりすることが可能。

4.1.2 プロジェクトフォルダーの作成

仕様がある程度固まったら実際にプロジェクトを作っていきましょう。フロントエンドとバックエンドで2つのプロジェクトを作成します。今回は「teikiadv」フォルダー内に「frontend」「backend」というプロジェクトをそれぞれ作成します。Visual Studio Codeのエクスプローラーメニューの「NODEJS [SSH: DEV.SIROISAKANA.COM]」となっている所の下のなにもない空間を右クリックし「新しいフォルダー」を選択、「teikiadv」フォルダーを作成します。また、その中に「backend」フォルダーを作成します。(「frontend」フォルダーはコマンドにより自動的に作られるので作成しなくても構いません。)ファイル構成は以下のようになります。

NODEJS [SSH: TEST.SIRO...
teikiadv / backend

4.1.3　フロントエンドの初期設定

　Visual Studio Codeのターミナルから「cd /home/teiki/nodejs/teikiadv/」を実行しプロジェクトフォルダー内に移動します。まずはフロントエンドのプロジェクトを作成しましょう。「npx create-nuxt-app frontend」を実行します。しばらく待つと設定内容を聞かれるので以下のように入力します。選択が必要な際はキーボードの上下キーで選択できます。なお、設定入力を間違えたときはCtrl+Cで中断して最初からやり直すことができます。

Project name	何も入力せず次へ
Project description	何も入力せず次へ
Author name	何も入力せず次へ
Choose programming language	JavaScriptを選択
Choose the package manager	Npmを選択
Choose UI framework	Noneを選択
Choose custom server framework	None (Recommended)を選択
Choose Nuxt.js modules	Axiosを有効化して次へ (スペースキーで有効化できます)
Choose linting tools	何も入力せず次へ
Choose test framework	Noneを選択
Choose rendering mode	Single Page Appを選択
Choose development tools	jsconfig.json (Recommended for VS Code) を有効化して次へ

```
[teiki@             teikiadv]$ npx create-nuxt-app frontend

create-nuxt-app v2.15.0
✨  Generating Nuxt.js project in frontend
? Project name frontend
? Project description My dandy Nuxt.js project
? Author name
? Choose programming language JavaScript
? Choose the package manager Npm
? Choose UI framework None
? Choose custom server framework None (Recommended)
? Choose Nuxt.js modules Axios
? Choose linting tools (Press <space> to select, <a> to toggle all, <i> to invert selection)
? Choose test framework None
? Choose rendering mode Single Page App
? Choose development tools jsconfig.json (Recommended for VS Code)
```

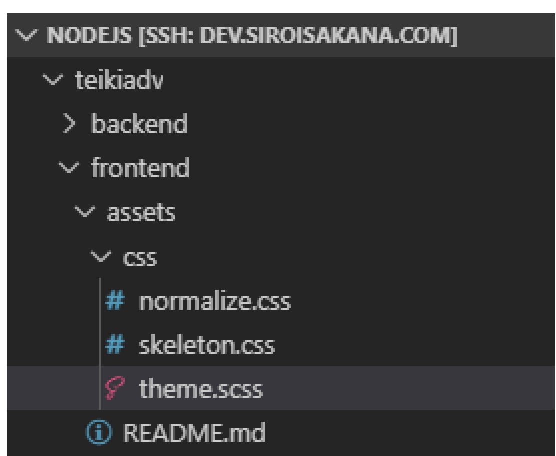
　しばらく待つと「frontend」フォルダー内にプロジェクトが作成されます。まずは追加のライブラリをインストールしましょう。「cd frontend」でディレクトリ移動し「npm install node-sass sass-loader jssha」を実行します。「jssh

a」については後々解説します。次にCSSフレームワークを設定します。ここではNormalize.cssとSkeletonを利用します。以下のURLからそれぞれファイルをダウンロードしてください。

Normalize.css: Make browsers render all elements more consistently.
https://necolas.github.io/normalize.css/
Downloadボタンを右クリックして「名前を付けてリンク先を保存」を選択し「normalize.css」として保存します。

Skeleton: Responsive CSS Boilerplate
http://getskeleton.com/
ダウンロードしたファイルを解凍(展開)して「css」フォルダー内にある「skeleton.css」を取り出します。

ファイルが入手できたら「frontend/assets」フォルダー内に「css」フォルダーを作成し、「normalize.css」と「skeleton.css」をそれぞれその中にドラッグアンドドロップで保存します。また、プロジェクト全体に適用するSCSSとして「theme.scss」も作成しておきましょう。コンポーネントによらないスタイルはこのファイルに記述するようにします。ファイル構成は以下のようになります。



最後に設定変更を行いましょう。今回は「https://dev.siroisakana.com/ta/」でフロントエンドにアクセスできるようにします。https化していない場合は「http://dev.siroisakana.com/ta/」にしてください。「frontend/nuxt.config.js」を開き、書いてある内容を全て削除して以下のように設定を書き換えます。

frontend/nuxt.config.js

```
export default {
  mode: 'spa',
  server: {
    port: 3000
  },
  head: {
    title: 'Teiki Adventure',
    titleTemplate: '%s | Teiki Adventure',
    meta: [
      { charset: 'utf-8' },
```

```
      { name: 'viewport', content: 'width=device-width, initial-scale=1' },
      { hid: 'description', name: 'description', content: '定期・APゲームのサンプル' },
      { hid: 'robots', name: 'robots', content: 'noindex' }
    ],
    link: [
      { rel: 'icon', type: 'image/x-icon', href: '/ta/favicon.ico' }
    ]
  },
  loading: false,
  css: [
    '~/assets/css/normalize.css',
    '~/assets/css/skeleton.css',
    '~/assets/css/theme.scss'
  ],
  plugins: [
  ],
  buildModules: [
  ],
  modules: [
    '@nuxtjs/axios'
  ],
  axios: {
    browserBaseURL: 'https://dev.siroisakana.com/ta'
  },
  router: {
    base: '/ta/'
  },
  build: {
    extend (config, ctx) {
    }
  }
}
```

　これで設定完了です。https化していない場合「axios」の「browserBaseURL」の先頭部分を「http」に変えるのを忘れないでください。なお、設定の内容は以下のようになっています。

・SPAモードでアプリケーションを実行する
・3000番ポートを使用する
・デフォルトのページタイトルを「Teiki Adventure」にする
・ページタイトルの末尾に「 | Teiki Adventure」が付与されるようにする
・<meta>タグのdescriptionを「定期・APゲームのサンプル」にする
・<meta>タグのrobotsを「noindex」に設定し、Googleなどの検索に載らないようにする
・「static/favicon.ico」をアイコンに設定する
・ローディングバーを表示しない
・「assets/css/normalize.css」「assets/css/skeleton.css」「assets/css/theme.scss」をこの順番で読み込む
・@nuxtjs/axiosモジュールを読み込む
・このプログラムから@nuxtjs/axiosモジュールを利用してAPIなどへのアクセスを行う際、「https://dev.siroisakana.com/ta」をベースURLにする

・「/ta/」ディレクトリでこのプログラムへアクセスするようにする

4.1.4　バックエンドの初期設定

次にバックエンドを作成しましょう。コンソールから「cd /home/teiki/nodejs/teikiadv/backend/」を実行し、さらに「npm init」を実行します。設定内容を聞かれるので以下のように入力し、「Is this OK?」と確認されたら「yes」と入力します。

package name	何も入力せず次へ
version	何も入力せず次へ
description	何も入力せず次へ
entry point	server/index.js
test command	何も入力せず次へ
git repository	何も入力せず次へ
keywords	何も入力せず次へ
author	何も入力せず次へ
license	UNLICENSED

次に必要なライブラリをインストールしましょう。以下のコマンドを実行します。

```
npm install express express-session redis connect-redis jssha mongoose passport passport-local config ejs
```

ここでインストールされるライブラリは以下の通りです。

express	Webアプリケーションライブラリ
express-session	Expressでセッション情報を扱いやすくするためのミドルウェア関数
redis	Redisというデータベースに接続するためのライブラリ
connect-redis	express-sessionのセッション情報をRedisに保存するためのライブリラリ
jssha	パスワードなどをハッシュ化するためのライブラリ
mongoose	MongoDBに接続して利用するためのライブラリ
passport	ログイン機構を作るためのライブラリ
passport-local	passportでユーザーIDとパスワードによるログイン機構を作るためのライブラリ
config	プロジェクトで利用するための設定を一元管理するためのライブラリ
ejs	簡単に使えるテンプレートエンジン

次に開発中に利用するライブラリをインストールします。「npm install nodemon -D」を実行します。nodemonはバックエンド側でホットリロードのような処理を実現するためのライブラリです。

さて、それではプロジェクトで利用するための設定ファイルを作りましょう。「backend」内に「config」フォルダーを

作成しその中に「default.json」を作成しましょう。ファイル構成は以下のようになります。

```
backend
  config
    {} default.json
```

公開するURLやポートなどの設定はこのファイルに記述しconfigライブラリを使ってアクセスします。今回バックエンド側のポートは4000番にし、URLは「/ta/api/」としましょう。以下の内容を入力して保存します。

backend/config/default.json

```
{
  "port": 4000,
  "directoryUrl": "/ta/api/"
}
```

次に実際のコードを書くためのファイルを用意しましょう。今回は「server/index.js」を実行することにします。また、ファイルの分割も行いましょう。バックエンドAPIは「server/api.js」に記述することにします。まずは「server」フォルダーを用意し、それぞれファイルを作成しましょう。ファイル構成は以下のようになります。

```
backend
  > config
  > node_modules
  server
    JS api.js
    JS index.js
  {} package-lock.json
  {} package.json
```

作成したらそれぞれ以下の内容を記述して保存します。「config.directoryUrl」で設定した公開URLを使用しています。

backend/server/index.js

```
const express = require('express');
const config  = require('config');
const api     = require('./api.js');
const app     = express();
const port    = config.port;

app.use(config.directoryUrl, api);
app.listen(port);
```

```
backend/server/api.js
const express = require('express');
const router  = express.Router();

router.get('/', (req, res) => {
  res.send('Backend API');
});

module.exports = router;
```

最後に起動設定を記述しましょう。「backend」内の「package.json」を開き、"scripts"の内容を以下のように書き換えて保存します。これで「npm run dev」とすることで自動リロード付きでプログラムが起動するように、「npm run start」とすることで普通にプログラムが起動するようになります。

```
backend/package.json
（省略）

"main": "server/index.js",
"scripts": {
  "dev": "nodemon server/index.js --watch config --watch server",
  "start": "node server/index.js"
},
"author": "",

（省略）
```

4.1.5 データベースの作成

次に使用するデータベースを作成しましょう。データベース名は「teikiadv」、パスワードは「6+lQ(N&si-&z」とします。パスワードは必ずこれではなく独自のものを使用してください。コンソールから「mongo --port 34189」を入力します。MongoDBのコンソールに移行するので、「db.auth("admin", "q%sv|N#IE99i")」を実行しMongoDBに管理者でログインします。

次に「use teikiadv」を実行しteikiadvデータベースに接続します。次にteikiadv用のユーザーを作成しましょう。ユーザー名はデータベース名と同じく「teikiadv」にします。以下のコマンドを実行します。

```
db.createUser({user: "teikiadv", pwd:"6+1Q(N&si-&z", roles:[{role:"readWrite",  db:"teikiadv"}]})
```

これでデータベースが作成されます。作成したら「exit」を入力するかCtrl+Cを押してMongoDBのコンソールから離脱しましょう。

```
[teiki@                          ]$ mongo --port 34189
MongoDB shell version v4.2.1
connecting to: mongodb://127.0.0.1:34189/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("115e82ae-93e2-448d-b072-3e9de460e683") }
MongoDB server version: 4.2.1
> use admin
switched to db admin
> db.auth("admin", "q%sv|N#IE99i")
1
> use teikiadv
switched to db teikiadv
> db.createUser({user: "teikiadv", pwd:"6+lQ(N&si-&z", roles:[{role:"readWrite",  db:"teikiadv"}]})
Successfully added user: {
        "user" : "teikiadv",
        "roles" : [
                {
                        "role" : "readWrite",
                        "db" : "teikiadv"
                }
        ]
}
> exit
bye
```

4.1.6　Nginxの設定変更

最後にnginxの設定を変更しましょう。Tera Termを起動しrootでログインします。「vi /etc/nginx/conf.d/default.conf」を実行し基礎学習編で使用した「location /test/」や「location /test/api/」の記述は削除し以下のように書き換えます。分かりやすいよう追記された部分は斜体になっています。

/etc/nginx/conf.d/default.conf

```
(省略)

 location / {
     root   /usr/share/nginx/html;
     index  index.html index.htm;
 }

 location /ta/ {
     proxy_pass   http://127.0.0.1:3000;
 }

 location /ta/api/ {
     proxy_pass   http://127.0.0.1:4000;
 }

(省略)
```

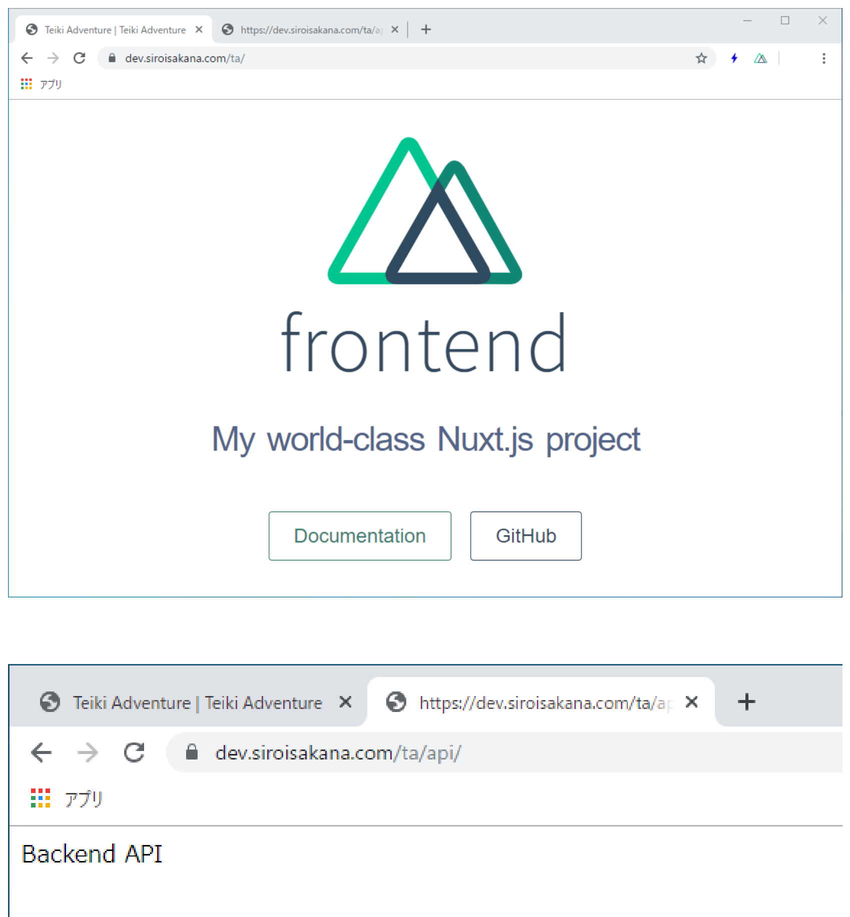
書き換えたら「nginx -s reload」を実行して設定を反映します。

4.1.7　プログラムの実行

それでは実際にプログラムを実行してみましょう。今回はフロントエンドとバックエンドのプログラムを同時に実行します。Visual Studio Codeではコンソールを2つ以上同時に起動しコンソール画面を分割することができます。コンソール右上にある分割アイコンをクリックしましょう。

するとコンソールが左右に分割されるはずです。まず左側のコンソールで「cd /home/teiki/nodejs/teikiadv/frontend」を実行し「npm run dev」を実行、次に右側のコンソールで「cd /home/teiki/nodejs/teikiadv/backend」を実行し「npm run dev」を実行します。これで左側のコンソールでフロントエンドが、右側のコンソールでバックエンドが起動します。なお、ENOSPCやEADDRINUSEといったようなエラーが出る場合は『補足:ENOSPCエラーが出る場合』や『補足:EADDRINUSEエラーが出る場合』を参照してください。

それでは「http://dev.siroisakana.com/ta/」「http://dev.siroisakana.com/ta/api/」にアクセスしてみましょう。それぞれ以下のように表示されれば正常にプロジェクトが作成できています。ホットリロード機能を使用するため、以降の解説では実行とタブでの表示が続行されているものとして解説しています。



4.1.8 補足:ENOSPCエラーが出る場合

プログラムを実行しようとしたときに「ENOSPC: System limit for number of file watchers reached」というようなエラーが出ることがあります。Nuxt.jsではホットリロード用にファイルの変更を監視しておりそのためにファイルウォッチャーを使用しているのですが、これがデフォルトではOS全体で8192個しか用意されておらず枯渇するとこのエラーが発生します。

ファイルウォッチャー数を上げることでこのエラーを解決することができます。まずはTera Termを起動しrootで

ログインしてください。ログインしたら以下の2つのコマンドを実行します。ここではファイルウォッチャー数を65536に上げています。

「sysctl fs.inotify.max_user_watches=65536」
「echo fs.inotify.max_user_watches=65536 | tee -a /etc/sysctl.conf」

これでENOSPCエラーは出なくなるはずです。これでもまだ出てしまう場合、数字の部分をさらに増やしてコマンドを再度実行しファイルウォッチャー数を増やしてください。

4.1.9　補足：EADDRINUSEエラーが出る場合

EADDRINUSEは指定のポートがすでに使われていて使えないときに表示されるエラーです。バックエンド側で保存してnodemonによりリロードされた際によく起こります。何回かリトライすればうまくいくのでうまくいくまでCtrl+Sで保存を繰り返してみましょう。

それでも上手く行かない場合一度Ctrl+Cで実行を停止し「lsof -i:(エラーの起こるポート)」を実行します。今回の例においてバックエンド側で起きている場合は「lsof -i:4000」になります。-i:の後にはスペースをあけないことに注意してください。すると以下のように指定のポート番号を専有しているプロセスIDが表示されます。（PIDと書かれている部分がそれです。）

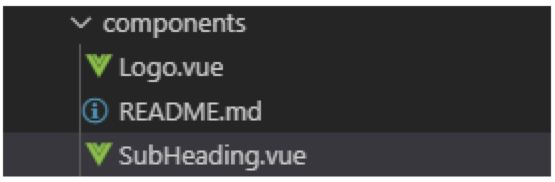
```
[teiki@                    ]$ lsof -i:4000
COMMAND   PID  USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
node    17971 teiki  19u  IPv6 532849      0t0  TCP *:terabase (LISTEN)
```

プロセスIDを確認したら「kill (プロセスID)」でプロセスを強制終了します。この画像の例の場合「kill 17971」になります。こうすることでEADDRINUSEが解消されるはずです。

4.2 トップページ / メニュー

4.2.1　トップページとレイアウトの変更

　まずは全体に適用するSCSS、全体のレイアウト、見出し用コンポーネント、トップページの内容を設定しましょう。このうち見出し用コンポーネントはファイルがないので作成します。「frontend/components/SubHeading.vue」を作成しましょう。ファイル構成は以下のようになります。



　作成したファイルに以下の内容を記述して保存します。

frontend/components/SubHeading.vue

```
<template>
  <h2 class="subheading"><slot></slot></h2>
</template>

<style lang="scss" scoped>
.subheading {
  font-size: 15px;
  margin: 20px 0px ;
  padding: 8px 0 8px 20px;
  border-left: 1px solid lightgray;
}
</style>
```

　見出し用コンポーネントが作成できたら残りを編集していきましょう。「frontend」の「assets/css/theme.scss」「layouts/default.vue」「pages/index.vue」をそれぞれ編集して保存します。

frontend/assets/css/theme.scss

```
html {
  overflow-y: scroll;
}

body {
  background: #F8F8F8;
  color: #333;
}

input::placeholder, textarea::placeholder {
  color: #CCC;
}

hr {
```

```
  margin: 20px 0;
}
```

frontend/layouts/default.vue

```
<template>
  <div>
    <div id="title">
      <nuxt-link id="title-logo" to="/">Teiki Adventure</nuxt-link>
    </div>
    <div id="columns">
      <main id="main-column">
    <nuxt />
    </main>
    <nav id="sub-column">
    </nav>
    </div>
  </div>
</template>

<script>
export default {}
</script>

<style lang="scss">
$columnsWidth: 1000px;
$mainColumnWidth: 800px;
$mainColumnPaddingX: 20px;
$mainColumnPaddingBottom: 20px;
$subColumnWidth: 200px;
$subColumnPaddingX: 0;

#title {
  margin: 0 auto;
  width: $columnsWidth;
  border-bottom: 1px solid lightgray;

  #title-logo {
    display: inline-block;
    font-size: 36px;
    padding: 10px 20px;
    color: #666;
    text-decoration: none;
  }
}

#columns {
  margin: 0 auto;
  width: $columnsWidth;
  display: flex;
  justify-content: space-between;

  #main-column {
    box-sizing: border-box;
    width: $mainColumnWidth;
    padding: 0 $mainColumnPaddingX $mainColumnPaddingBottom $mainColumnPaddingX;
  }
```

```
  #sub-column {
    box-sizing: border-box;
    width: $subColumnWidth;
    padding: 0 $subColumnPaddingX;
  }
}
</style>
```

```
frontend/pages/index.vue
<template>
  <section>
    <section>
      <sub-heading>イントロダクション</sub-heading>
      <p>
        Teiki Adventureへようこそ。<br>
        このゲームはあなただけのオリジナルキャラクターを登録し、<br>
        世界を冒険したり他のキャラクターと交流したりしながら遊ぶゲームです。<br>
        定期・APゲーム制作教本のサンプルとして作られました。そのため実際に遊ぶことはできません。<br>
        <br>
        新規登録は<nuxt-link to="/register">こちら</nuxt-link>、ログインは<nuxt-link to="/login">こち
ら</nuxt-link>から行えます。
      </p>
    </section>
  </section>
</template>

<script>
import SubHeading from '~/components/SubHeading.vue'

export default {
  components: {
    SubHeading
  },
  head() {
    return {
      title: 'Teiki Adventure',
      titleTemplate: '%s'
    };
  }
}
</script>
```

ここまで設定するとトップページが以下のようになるはずです。今回は2カラム形式でページを作っていきます。なおリンクが張られていますがこれは後から作るページへのリンクを先に記述しておいたものでありクリックしてもリンク先はありません。

4.2.2　メニューの作成と読み込み

サブカラムにメニューを作っていきましょう。今回は以下の2つのコンポーネントで表現することにします。

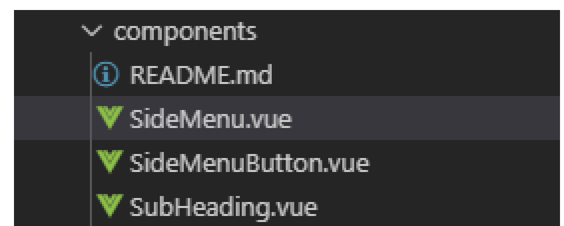
SideMenuButton.vue

サブカラムのメニューのボタン1つ1つのコンポーネントです。

SideMenu.vue

サブカラムのメニュー全体のコンポーネントです。「SideMenuButton.vue」を複数並べてメニューを作ります。

それでは作っていきましょう。「components」フォルダー内にそれぞれファイルを作成します。なお、この先「Logo.vue」は使用しないので削除しても構いません。ファイル構成は以下のようになります。



作成したらそれぞれ以下の内容を記述して保存します。

frontend/components/SideMenuButton.vue

```
<template>
  <nuxt-link class="sidemenu-button-link" :to="to">
    <div class="sidemenu-button">
      <slot></slot>
    </div>
  </nuxt-link>
</template>
```

```
<script>
export default {
  props: {
    to: String
  }
}
</script>

<style lang="scss" scoped>
$height: 40px;
$marginLength: 10px;
$textColor: #333;
$borderColor: lightgray;

.sidemenu-button-link {
  text-decoration: none;
}

.sidemenu-button {
  margin: $marginLength $marginLength 0 $marginLength;
  height: $height;
  display: flex;
  align-items: center;
  justify-content: center;
  color: $textColor;
  border: 1px solid $borderColor;
  border-radius: 4px;
}
</style>
```

frontend/components/SideMenu.vue

```
<template>
  <div>
    <side-menu-button to="/notice">お知らせ</side-menu-button>
    <side-menu-button to="/declare">宣言</side-menu-button>
    <side-menu-button to="/explore">探索</side-menu-button>
    <side-menu-button to="/log">ログ</side-menu-button>
    <side-menu-button to="/skill">戦闘設定</side-menu-button>
    <side-menu-button to="/talk/public">交流</side-menu-button>
    <side-menu-button to="/profile/main/edit">キャラクター設定</side-menu-button>
    <side-menu-button to="/list">キャラクター一覧</side-menu-button>
    <side-menu-button to="/rulebook">ルールブック</side-menu-button>
    <div class="mini-link-wrapper">
      <nuxt-link class="mini-link" to="/inquiry">&gt;&gt; 問い合わせ</nuxt-link>
    </div>
  </div>
</template>

<script>
import SideMenuButton from '~/components/SideMenuButton.vue'

export default {
  components: {
    SideMenuButton
```

```
  }
}
</script>

<style lang="scss" scoped>
.mini-link-wrapper {
  display: flex;
  flex-direction: column;
  align-items: flex-end;
  padding: 10px;

  .mini-link {
    text-decoration: none;
    color: #666;
    font-size: 12px;
  }
}
</style>
```

「SideMenuButton.vue」はto属性でリンク先を受け取り内部的に<nuxt-link>でリンクしているボタンになっていて、「SideMenu.vue」はそれを並べています。サイドメニューを1つのコンポーネントで表現するのではなくこのように分けることでボタンのデザインを変えやすくなってメンテナンス性が向上したりコードが分かりやすくなったりします。

それではレイアウトにこれを読み込みましょう。「frontend/layuouts/default.vue」の<template>と<script>を以下のように書き換え保存します。(<style>については変更がないので省略しています。)

frontend/layuouts/default.vue

```
<template>
  <div>
    <div id="title">
      <nuxt-link id="title-logo" to="/">Teiki Adventure</nuxt-link>
    </div>
    <div id="columns">
      <main id="main-column">
        <nuxt />
      </main>
      <nav id="sub-column">
        <side-menu></side-menu>
      </nav>
    </div>
  </div>
</template>

<script>
import SideMenu from '~/components/SideMenu.vue'

export default {
  components: {
    SideMenu
  }
}
</script>
```

```
（省略）
```

トップページが以下のようになれば成功です。

4.3 キャラクター登録

4.3.1　キャラクタースキーマの作成

　キャラクターを登録するにあたってデータを管理するためにデータベースのスキーマを組んでいきましょう。スキーマを組む場合後から変更すると整合性を取るのが難しくなるため、必要だと思われるフィールドをあらかじめ用意しておくことが大切です。それではキャラクタードキュメントに必要になるフィールドについて考えましょう。仕様から考えると以下のようなフィールドが必要になるでしょう。

ENo	エントリーナンバー
パスワード	ハッシュ化されたパスワード
キャラクター名	キャラクターのフルネーム
短縮名	キャラクターの短縮名
登録日時	登録された日時
タグ	キャラクターに付与されたタグ
メインアイコン	キャラクターリストなどで表示されるアイコン
アイコン	トークルームなどで使用するアイコン
サマリー	キャラクターリストで表示される短いキャラクターの説明文
プロフィール文	キャラクターページで表示されるプロフィール文
プロフィール画像	キャラクターページでランダムに表示されるプロフィール画像
削除フラグ	キャラクターが削除されているかどうかのフラグ
AP	現在AP
NP	割り振れる能力値ポイント
ATK, DEX, MND, AGI, DEF	割り振ってある能力値ポイント
スキル	習得しているスキル
物語クリア状況	「物語戦」のクリア状況
探索クリア状況	「探索戦」のクリア状況
宣言内容・行き先	定期更新の宣言のうち、どの物語戦に挑戦するか
宣言内容・連れ出しキャラクター	定期更新の宣言のうち、物語戦で連れ出すキャラクター
宣言内容・日記	定期更新の宣言のうち、日記の内容
お気に入り	お気に入りしているキャラクター
ブロック	ブロックしているキャラクター
ミュート	ミュートしているキャラクター

　ただしこれだけでは足りません。「どのキャラクターにブロックされているか」というデータも保持しておきます。これはデータベースを検索すれば分かる情報ではありますが、必要になる回数が多く持たせておくことで検索の手間を省ける（＝データベースの検索回数が減り処理が軽量化できる）他に、「ブロックしているリスト」と「ブロックさ

れているリスト」を単純につなげるだけで「非表示にするべきキャラクターのリスト」が作れて便利だからです。

以上を踏まえてスキーマを作ります。その前にまずは「backend/config/default.json」に使う設定を記述しておきましょう。ここではデータベースの接続情報とCSRFトークンの長さ、キャラクター名・短縮名の最大の長さ・初期NPを指定します。なお、突然CSRFという謎の単語が出現しましたがこれは後の方で解説します。CSRFについてはこの先も時々出てくるのですが今は気にしなくても構いません。とりあえずおまじないみたいなものだと思っておいてください。さて、それでは内容を以下のように書き換えます。追記された部分は斜体で表示してあります。

backend/config/default.json
```
{
  "port": 4000,
  "directoryUrl": "/ta/api/",
  "dbName": "teikiadv",
  "dbUser": "teikiadv",
  "dbPassword": "6+LQ(N&si-&z",
  "dbPort": 34189,
  "csrfTokenLength": 32,
  "nameMaxLength": 16,
  "nicknameMaxLength": 8,
  "initialNp": 20
}
```

次にスキーマを記述しましょう。モデルの宣言については管理しやすくするためファイルを分割します。「backend/server」内に「model.js」を作ります。ファイル構成は以下のようになります。

```
backend
  config
  node_modules
  server
    api.js
    index.js
    model.js
```

作成したら以下の内容を記述して保存します。

backend/server/model.js
```
const mongoose = require('mongoose');
const config = require('config');
const crypto = require('crypto');
const Schema = mongoose.Schema;

const CharacterSchema = new Schema({
  eno:      {type: Number, sparse: true},    // ENo
  password: {type: String, required: true}, // パスワード
  name:     {type: String, required: true, index: false, maxlength: config.nameMaxLength},      // キャラクター名
  nickname: {type: String, required: true, index: false, maxlength: config.nicknameMaxLength}, // 短縮名
```

```
  csrf:     {type: String, default: () => crypto.randomBytes(config.csrfTokenLength).toString('hex')}, // CSRF
トークン
  deleted:  {type: Boolean, default: false, index: true}, // 削除フラグ
  tags:     [String], // タグ
  mainicon: String,   // キャラクターリストなどに表示されるメインアイコン
  summary:  String,   // キャラクターリストに表示される短いキャラ説明文
  profile:  String,   // プロフィール文
  profileImages:    [String], // プロフィール画像
  registrationTime: {type: Date, default: Date.now}, // 登録日時
  icons: [{  // アイコン
    name: String, // アイコン名
    url:  String  // アイコンURL
  }],

  ap: {type: Number, default: 0}, // 現在AP
  np: {type: Number, default: config.initialNp}, // 割り振れる能力値ポイント
  status: {  // 割り振ってある能力値
    atk: {type: Number, default: 0},
    dex: {type: Number, default: 0},
    mnd: {type: Number, default: 0},
    agi: {type: Number, default: 0},
    def: {type: Number, default: 0}
  },
  skill:   [Number], // セットしているスキル
  story:   [Number], // 物語戦クリア状況
  explore: [Number], // 探索戦クリア状況
  declare: {         // 宣言内容
    diary:   {type: String, default: null}, // 日記         nullで未設定
    storyId: {type: Number, default: null}, // 物語戦行き先 nullで未設定
    party:   {type: [Schema.Types.ObjectId], ref: 'Character'} // 物語戦連れ出しメンバー
  },
  fav:     {type: [Schema.Types.ObjectId], ref: 'Character'}, // お気に入り
  block:   {type: [Schema.Types.ObjectId], ref: 'Character'}, // ブロック
  blocked: {type: [Schema.Types.ObjectId], ref: 'Character'}, // 被ブロック
  mute:    {type: [Schema.Types.ObjectId], ref: 'Character'}  // ミュート
});

const Character = mongoose.model('Character', CharacterSchema);

mongoose.connect(`mongodb://${config.dbUser}:${encodeURIComponent(config.dbPassword)}@127.0.0.1:${config.dbPor
t}/${config.dbName}`);

module.exports = {
  Character: Character
};
```

よく検索される＆重複しないENoにはスパースインデックスを張っている他、パスワード・フルネーム・短縮名を必須にしています。またフルネームと短縮名には最大の長さを指定してあります。その他、AP・NP・各能力値は初期値を0としています。使用する際は外部ファイルから以下のようにしてモデルを受け取ってそのまま使用します。

```
const Character = require('./model.js').Character;
await Character.find({});
```

なお、CSRFというよくわからない項目もありますがこれは後々解説します。今はまだ気にしなくても構いません。

4.3.2　キャラクター登録APIの作成

それではキャラクター登録APIを作りましょう。POSTでオブジェクトを受け取ってドキュメントを作成し保存、という手順になります。「express.json()」というミドルウェア関数を使うとPOSTやPUTで送られた値をreq.bodyからオブジェクト形式で受け取れるようになります。まずはその利用宣言を行います。「frontend/server/index.js」を以下のように書き換えて保存しましょう。

frontend/server/index.js

```
const express = require('express');
const config  = require('config');
const api     = require('./api.js');
const app     = express();
const port    = config.port;

app.use(express.json());

app.use(config.directoryUrl, api);
app.listen(port);
```

次にデータベースを読み込んでAPIを作成します。正常に処理が完了すればステータスコード200を、それ以外であればステータスコード500を返すようにしましょう。以下のようになるでしょうか。

```
router.post('/characters', async (req, res) => {
  /* req.body = {
      name: String,
      nickname: String,
      password: String
    } */

  const character = new Character(req.body);

  try {
    await character.save();
    res.status(200).send();
  } catch (e) {
    res.status(500).send();
  }
});
```

しかし、これはセキュリティ的に好ましくないAPIです。ここで注意するべきなのは「クライアントから送信されたオブジェクトをそのまま使用しない」ということです。例えば悪意を持ったユーザーが最初からAP10にしたいと思ってなんらかの手段を用いてURLに以下のようなオブジェクトを送ってきたとしましょう。

```
{
  name: 'ああああ',
  nickname: 'ああああ',
  password: 'PaSsWoRd',
  ap: 10
}
```

これは以下のように解釈され、最初からAP10を持ったキャラクターが作成されてしまいます。

```
const character = new Character({
  name: 'ああああ',
  nickname: 'ああああ',
  password: 'PaSsWoRd',
  ap: 10
});
```

このような事態を防ぐため、クライアント側から送られてきたデータは信用しないようにしましょう。送るデータを改竄するのは比較的簡単にできてしまうからです。それを踏まえてAPIを作ります。ここでは「/characters」にPOSTリクエストを送ることでキャラクターを作成できるようにしましょう。「backend/server/api.js」を以下のように書き換え保存します。必要なフィールド以外が登録されないようになっています。

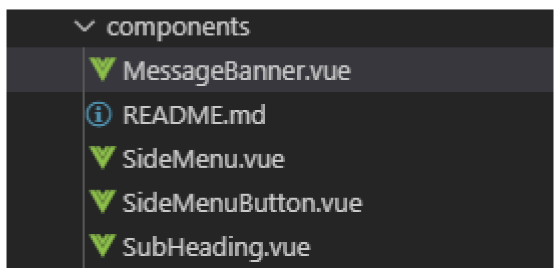
backend/server/api.js

```
const express   = require('express');
const Character = require('./model.js').Character;
const router    = express.Router();

router.post('/characters', async (req, res) => {
  const character = new Character({
    name: req.body.name,
    nickname: req.body.nickname,
    password: req.body.password
  });

  try {
    await character.save();
    res.status(200).send();
  } catch (e) {
    res.status(500).send();
  }
});

module.exports = router;
```

4.3.3　ページの作成

それではフロントエンド側でページを作成しましょう。その前に登録時などは入力内容がおかしいなどでエラー

が発生する可能性があるためエラーメッセージなどを表示するためのコンポーネントを作成します。「frontend/components」内に「MessageBanner.vue」を作成します。ファイル構成は以下のようになります。



作成したら以下の内容を記述して保存します。

frontend/components/MessageBanner.vue

```
<template>
  <div :class="['message-banner', type]">
    <slot></slot>
  </div>
</template>

<script>
export default {
  props: {
    type: String
  }
}
</script>

<style lang="scss" scoped>
.message-banner {
  width: 100%;
  margin: 20px 0;
  padding: 10px 20px;
  box-sizing: border-box;
  border: 1px solid gray;
  display: flex;
  flex-direction: column;
  justify-content: center;
  background-color: lightgray;
  border-radius: 4px;
}

.success {
  border: 1px solid #007243;
  background-color: #9cd9ac;
}

.info {
  border: 1px solid #005476;
  background-color: #79baca;
}
```

```
.warning {
  border: 1px solid #b39300;
  background-color: #f2e6b8;
}

.error {
  border: 1px solid #a61d39;
  background-color: #e8c2bf;
}
</style>
```

「MessageBanner.vue」はメッセージを受け取って表示するためのコンポーネントです。typeという属性を受け取りそれによってメッセージボックスの色を変えるようにしてあります。実装的にはtypeで受け取った値をそのままクラスに適用してそのクラスによって色を出し分けています。

メッセージ表示システムを作ったら次は「nuxt.config.js」に設定を記述しましょう。名前と短縮名の最大の長さとパスワードの最小の長さを設定しておきます。バックエンド側では「backend/config/default.json」に設定内容を記述していましたが、Nuxt.js側では「nuxt.config.js」内の「env」に設定を記述していきます。「frontend/nuxt.config.js」を開き、「env」を作って以下のように設定を記述します。分かりやすいよう追記された箇所は斜体で表示してあります。「env」の位置は適切な場所ならどこでもいいのですが、ここでは「build」の前に置いてあります。

frontend/nuxt.config.js

```
  （省略）

  router: {
    base: '/ta/'
  },
  env: {
    nameMaxLength: 16,
    nicknameMaxLength: 8,
    passwordMinLength: 8
  },
  build: {

  （省略）
```

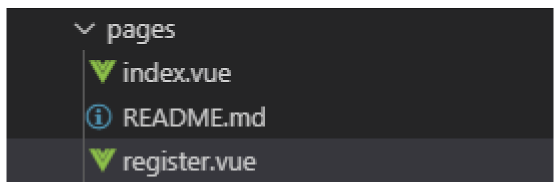
また、ページに適用するスタイルも作成します。フォームに関係するスタイルを作るのですが、これはこのページ以外にも多用することが予想されるので「frontend/assets/css/theme.scss」に記述します。ファイルを開き、末尾に以下のようにスタイルを追記して保存します。

frontend/assets/css/theme.scss

```
  （省略）

.form {
  margin: 10px 0 0 20px;

  &-title {
    font-weight: bold;
```

```
    color: #555;
  }

  &-description {
    color: #666;
  }

  &-input {
    margin: 8px 0;
    width: 300px;
  }

  &-input-long {
    margin: 8px 0;
    width: 600px;
  }

  &-textarea {
    margin: 8px 0;
    width: 600px;
    height: 200px;
    resize: none;
  }
}

.button-wrapper {
  width: 100%;
  display: flex;
  justify-content: flex-end;

  .button {
    margin: 10px;
  }
}
```

ここまで組んだら実際にページを作ります。「frontend/pages」内に「register.vue」を作成します。ファイル構成は以下のようになります。



作成したら以下の内容を記入して保存してください。

frontend/pages/register.vue

```
<template>
  <section>
    <section>
      <sub-heading>利用規約</sub-heading>
      <ol id="terms-list">
```

```
        <li>登録は1人1キャラクターのみ</li>
        <li>登録はオリジナルキャラクターのみ</li>
        <li>不適切な画像の設定や投稿などを行わないこと</li>
        <li>教本のサンプル用ゲームのため更新等は行われません</li>
      </ol>
    </section>
    <section>
      <sub-heading>新規登録</sub-heading>
      <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
      <section class="form">
        <div class="form-title">キャラクターのフルネーム（{{ name.length }} / {{ nameMaxLength }}文字）</div>
        <div class="form-description">
          キャラクターのフルネームを入力します。後からでも変更可能です。
        </div>
        <input class="form-input" type="text" v-model="name" placeholder="フルネーム">
      </section>
      <section class="form">
        <div class="form-title">キャラクターの短縮名（{{ nickname.length }} / {{ nicknameMaxLength }}文字）</div>
        <div class="form-description">
          キャラクターの短縮名を入力します。後からでも変更可能です。
        </div>
        <input class="form-input" type="text" v-model="nickname" placeholder="短縮名">
      </section>
      <section class="form">
        <div class="form-title">パスワード（{{ passwordMinLength }}文字以上）</div>
        <div class="form-description">
          ログインに使用するパスワードを入力します。
        </div>
        <input class="form-input" type="password" v-model="password" placeholder="パスワード">
      </section>
      <section class="form">
        <div class="form-title">パスワード再入力</div>
        <div class="form-description">
          再度同じパスワードを入力します。
        </div>
        <input class="form-input" type="password" v-model="confirm" placeholder="再入力">
      </section>
      <div class="button-wrapper">
        <button class="button" @click="register">登録</button>
      </div>
    </section>
  </section>
</template>

<script>
import jsSHA         from 'jssha'
import SubHeading    from '~/components/SubHeading.vue'
import MessageBanner from '~/components/MessageBanner.vue'

export default {
  components: {
    SubHeading,
    MessageBanner
  },
  head() {
    return {
      title: '新規登録'
```

```
    };
  },
  data() {
    return {
      // nuxt.config.jsのenvの内容を受け取る
      nameMaxLength:      process.env.nameMaxLength,
      nicknameMaxLength: process.env.nicknameMaxLength,
      passwordMinLength: process.env.passwordMinLength,

      name:             '', // キャラクターのフルネーム
      nickname:         '', // キャラクターの短縮名
      password:         '', // パスワード
      confirm:          '', // パスワード再入力\
      errorMessage:     '', // エラーメッセージ
      waitingResponse: false // 通信待ち中かどうか
    }
  },
  methods: {
    register: async function() {
      if (this.waitingResponse) {
        // 接続待ち状態であればアラートを表示しreturn、以降の処理を行わない
        return alert('しばらくお待ち下さい');
      }

      // 入力内容の検証を行う、問題があればエラーメッセージを表示、returnして処理を中断
      if (!this.name) { // フルネームが入力されていない
        return this.errorMessage = 'フルネームが入力されていません';
      }
      if (!this.nickname) { // 短縮名が入力されていない
        return this.errorMessage = '短縮名が入力されていません'
      }
      if (!this.password) { // パスワードが入力されていない
        return this.errorMessage = 'パスワードが入力されていません'
      }

      if (this.nameMaxLength < this.name.length) {
        return this.errorMessage = 'フルネームが長すぎます';
      }
      if (this.nicknameMaxLength < this.nickname.length) {
        return this.errorMessage = '短縮名が長すぎます';
      }
      if (this.password.length < this.passwordMinLength) {
        return this.errorMessage = 'パスワードが短すぎます';
      }
      if (this.password != this.confirm) {
        return this.errorMessage = 'パスワードと再入力の内容が一致しません';
      }

      // 接続に入る、接続待ち状態をON(true)に
      this.waitingResponse = true;

      try {
        await this.$axios.post('/api/characters', {
          name:     this.name,
          nickname: this.nickname,
          password: this.password,
        });
```

```
        console.log('登録が完了しました');
      } catch (e) {
        // 登録に失敗した場合接続待ち状態をOFF(false)に
        this.waitingResponse = false;
        console.log('登録に失敗しました');
      }
    }
  }
}
</script>

<style lang="scss" scoped>
#terms-list {
  margin: 20px;
}
</style>
```

保存して「http://dev.siroisakana.com/ta/register」にアクセスすると以下のようなページになるはずです。

Teiki Adventure

利用規約

1. 登録は1人1キャラクターのみ
2. 登録はオリジナルキャラクターのみ
3. 不適切な画像の設定や投稿などを行わないこと
4. 教本のサンプル用ゲームのため更新等は行われません

新規登録

キャラクターのフルネーム（16文字以内）
キャラクターのフルネームを入力します。後からでも変更可能です。
フルネーム

キャラクターの短縮名（8文字以内）
キャラクターの短縮名を入力します。後からでも変更可能です。
短縮名

パスワード（文字以上）
ログインに使用するパスワードを入力します。
パスワード

パスワード再入力
再度同じパスワードを入力します。
再入力

登録

お知らせ
宣言
探索
ログ
戦闘設定
交流
キャラクター設定
キャラクター一覧
ルールブック
>> 問い合わせ

少し長いですが順を追って処理を追いかけていきましょう。まずは<script>のdata()から見ていきます。「nuxt.config.js」の「env」に設定した値はdata()内などでprocess.envから取得することができます。ここで設定内容をこのページで使えるようにしています。また、v-modelに設定するためにそれぞれの入力項目の値を作成している他、エラーメッセージ表示用の値と接続待ち用の値も用意してあります。

さて、ユーザーが必要な値の入力を行った後に「登録」ボタンを押すとmethodsの「register」が呼び出されま

す。registerではまず入力内容の検証を行い、なにか問題があればerrorMessageにエラーメッセージの内容を代入して処理を中断します。<template>内の「v-if="errorMessage"」の指定によりエラーメッセージになにか値が入っている場合MessageBannerが表示されるようになっているので、これによりエラーメッセージがなにか問題がある際にエラーメッセージを表示することができます。

特に問題がなければ接続に入っていきます。ここではAxiosというモジュールを使用しています。AxiosはAPIへのデータ送信などのHTTP通信を扱いやすくしてくれるライブラリで、プロジェクト作成時にAxiosを有効化しているのでmethods内などでthis.$axiosから呼び出すことができます。

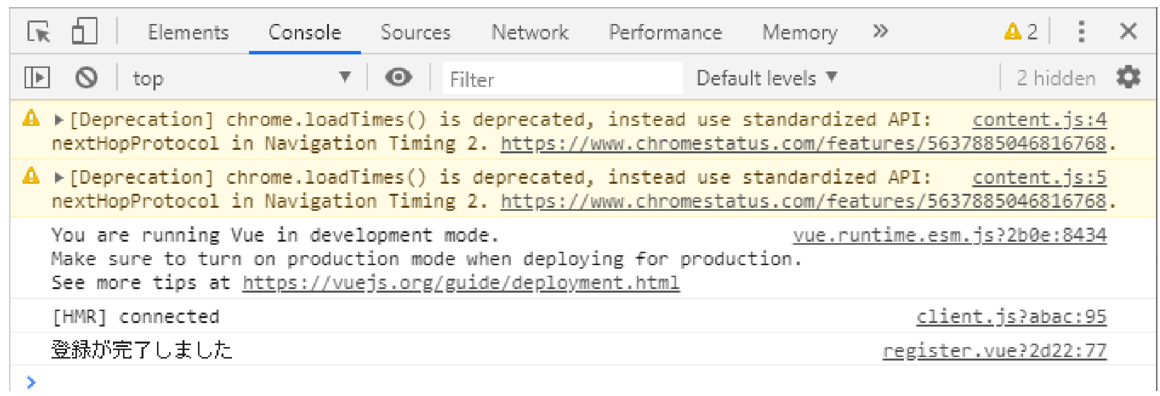
AxiosからPOSTを行う際はthis.$axios.post('送信先', 送信するオブジェクトの内容)で行います。この例では名前と短縮名とパスワードを送信しています。この操作はネットワーク接続、つまり非同期処理になるのでasync/awaitを使用しています。

なお、ボタンを何回もクリックしてしまったときに何キャラも登録されてしまわないように連続クリック対策を行っています。waitingResponseという値がそれで、接続待ち中はtrueになるようにしています。これにより、waitingResponseがまだtrueのときに再度登録処理を行おうとしたときに警告メッセージを出して処理を中断できるようにしてあります。

試しに登録を行ってみましょう。その際、テストのためにあえて名前が長すぎるなど不正な値の入力を試してみてください。次のようにエラーメッセージが表示されればOKです。

フルネームが長すぎます

確認したら正常な値を入力してキャラクターを登録してみてください。登録できていればChromeの開発者ツール(F12キーを押すことで開けます)のConsoleに「登録が完了しました」と表示されています。ちなみに「chrome.loadTime() is deprecated,」「Failed to load resource: net::ERR_HTTP2_PROTOCOL_ERROR」といった警告やエラーが出ている場合もありますが、出ていても支障はなく公開時には出なくなるので気にしなくても構いません。



登録したらMongoDB Compassから登録されたデータを確認してみます。MongoDB Compassを起動し、Connect to Hostの入力欄にそれぞれ以下の内容を入力して画面下の「CONNECT」というボタンを押します。特に記述のない項目はそのままで構いません。

Hostname	dev.siroisakana.com
Port	34189
Authentication	Username / Password
Username	teikiadv
Password	6+lQ(N&si-&z
Authentication Database	teikiadv

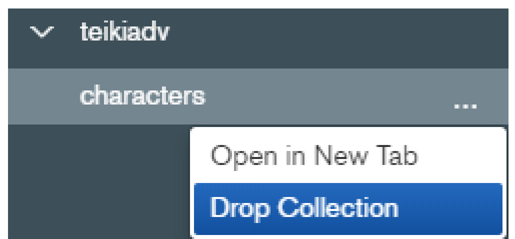
接続して「characters」コレクションを確認してみましょう。以下のように登録されているはずです。

```
  _id: ObjectId("5de3ca59b0666950424cc8e1")
> status: Object
> declare: Object
  deleted: false
> tags: Array
> profileImages: Array
  ap: 0
  np: 20
> skill: Array
> story: Array
> explore: Array
> fav: Array
> block: Array
> blocked: Array
> mute: Array
  name: "テスト"
  nickname: "テスト"
  password: "password"
  csrf: "5d90fa40dfa11c8dc3e26964f89216f9d78a18b0da48663224fb76f6d300b162"
  registrationTime: 2019-12-01T14:12:41.050+00:00
> icons: Array
  __v: 0
```

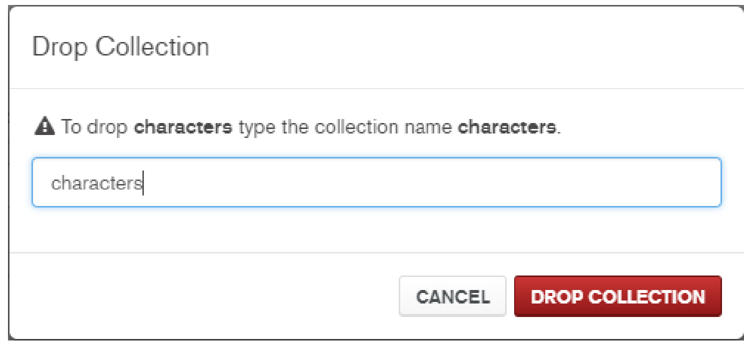
入力した内容通りに登録できていることが確認できます。

4.3.4 ENoの設定

登録されている内容ですが、1つ問題があります。それはENoが登録されていないということです。MongoDBを使う場合ENoは自分で登録処理を書かなければなりません。それではENoの登録処理を書いていきましょう。その前に、charactersコレクションは削除(リセット)しておきます。左メニューのcharactersから「Drop Collection」を選択します。



削除確認としてコレクション名の入力を求められるので「characters」と入力し「DROP COLLECTION」ボタンをクリックします。



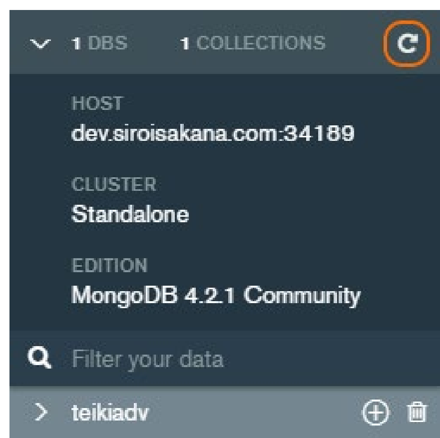
さて、MongoDBではデフォルトの検索結果は登録順に並んでいます。これを利用し全キャラクタードキュメントのうち登録したドキュメントが何番目にあたるかを調べることでENoを設定します。「backend/server/api.js」のAPI宣言部分のtry〜catchを以下のように書き換え保存します。

backend/server/api.js

```
（省略）

try {
  await character.save();
  const allCharacters = await Character.find({}, {_id: 1});

  for (let i = allCharacters.length - 1; 0 <= i; i--) {
    if (allCharacters[i]._id.toString() == character._id.toString()) {
      await character.update({eno: i + 1});
      return res.status(200).send({eno: i + 1});
    }
  }

  return res.status(500).send();
} catch (e) {
  return res.status(500).send();
}

（省略）
```

全てのキャラクタードキュメントの「_id」を検索し、作成したキャラクタードキュメントの「_id」と一致すればインデックス+1をENoとして設定しています。（配列のインデックスは0, 1, 2 …ですがENoは一般に1, 2, 3 …のため+1して数を合わせています。）なおそれぞれのキャラクタードキュメントは特殊な型になっており単純な==やequals()では比較することができないため文字列化してから一致するか比較しています。なお新規登録されたキャラクタードキュメントは配列の後ろ側にあるため後ろから検索するようにしています。また、ENo何番に登録されたのかという情報は後々使うことになるのでその情報も返送するようにしています。

さて、変更内容を保存したらもう一度キャラクターを登録してみましょう。登録したらMongoDB Compassで登録できているかどうか確認します。リロードは左メニュー上のリロードボタンから行います。



リロードしたら「teikiadv」内に「characters」コレクションが作成されるはずなので確認してみましょう。以下のようにenoが正常に登録されていれば成功です。

```
  _id: ObjectId("5de3cb392b0a7c506f6998a0")
> status: Object
> declare: Object
  deleted: false
> tags: Array
> profileImages: Array
  ap: 0
  np: 20
> skill: Array
> story: Array
> explore: Array
> fav: Array
> block: Array
> blocked: Array
> mute: Array
  name: "テスト"
  nickname: "テスト"
  password: "password"
  csrf: "06c99e8bc1130a04ccf9ec78b29f66bc9e81985450190cc6b98fb5640304dc74"
  registrationTime: 2019-12-01T14:16:25.855+00:00
> icons: Array
  __v: 0
  eno: 1
```

登録の成功を確認したらもう一度「characters」コレクションを削除(リセット)しておきましょう。先程と同様の手順で削除することができます。

4.3.5　パスワードのハッシュ化

登録処理はこれでうまくいっているように思えますが、まだこのままではセキュリティ上の懸念があります。それはパスワードがそのままの形(**平文**)で保存されているということです。パスワードが平文で保存されたままだと万が一データベースに不正アクセスされた際にパスワードが盗み出されてしまい、ユーザーが他サイトと共通のパスワードを使っていた場合他サイトへも被害が行ってしまう可能性があります。(本当はユーザー側もサイトごとに個別のパスワードを使うべきではありますが、実際にそれをユーザー全員に強制させられるかというと非常に難しいです。)

そこで利用するのが**ハッシュ化**です。ハッシュ化とは**ハッシュ関数**を使い、もとの値を一見ランダムな値に不可逆的に変換する操作のことを指します。ハッシュ関数は以下のような特性を持ちます。

1. 同じ値を入れると同じ値が出力される
2. 元の値が少しでも変わると全く異なる値が出力される
3. ハッシュ化された値から元の値を推測や特定することは非常に難しい(ほぼ不可能)

これはパスワードの管理という点で非常に有用です。1.の性質により同じパスワードからは同じハッシュ値が生成されます。つまり、ハッシュ化されたパスワードさえ管理しておけばパスワードがどんな内容か分からなくてもパスワードが合っているかどうかは判断することができます。また、万が一ハッシュ化されたパスワードが漏洩したときにも3.の性質により元のパスワードがどんなものだったか特定できなくなります。

もしハッシュ化されたパスワードから元のパスワードを特定しようとした場合、とにかく手当り次第にいろいろなパスワードをハッシュ化してみて偶然結果が一致すれば特定完了、という方法を取らざるを得なくなります。このような攻撃のことを**総当たり攻撃**(ブルートフォースアタック)と呼びます。

実際に使う際にはさらにセキュリティ性を高めるために**ソルティング**と**ストレッチング**と呼ばれる操作を行います。

ソルティングとは元のパスワードに**ソルト**と呼ばれる適当な値を加えてハッシュ化される前の値を複雑にする操作のことを指します。例えば元のパスワードが「pass1234」だったときにソルトが「H5*eYMDJ76y(」だったとしましょう。ハッシュ化される前の値は「pass1234H5*eYMDJ76y(」になります。これによりどのようなメリットがあるかというと、総当たり攻撃の結果を予め計算しておいたファイル(**レインボーテーブル**)が存在したときに「pass1234」などであればもしかしたらレインボーテーブルに結果が存在していて容易にパスワードが破られる可能性がありますが、「pass1234H5*eYMDJ76y(」であればレインボーテーブルにデータが存在する確率はほぼ0になりパスワードが破られる可能性が低くなります。

また、ストレッチングとはハッシュ化を何回も繰り返すことを指します。例えばハッシュ化1回につき0.001秒かかり100億回総当たり攻撃をすればパスワードを特定できるとしましょう。ハッシュ化1回だけだと1つパスワードを特定するのにかかる時間は約115日です。長いとはいえこれぐらいであれば現実的に可能な気がしますね。しかしハッシュ化を300回繰り返せば1回の操作にかかる時間は0.3秒になります。ユーザー側は0.001秒が0.3秒になってもほぼ感覚としては大差ありませんが、攻撃者側が1つパスワードを特定しようとした場合にかかる時間は約115日から約95年になり実質不可能といってもいいレベルになります。このためセキュリティの強化につながります。

実はこの操作をすべて行ったとしても例えば「password」「pass1234」「qwertyui」のようなよく使われるパスワードを利用していた場合破られる可能性が高くなります。人間がよく使うパスワードというものは決まりきっていて流石に例のようなものは使わないとしても特定の単語だとか、それに数字をつけただけだとか、単語を2つ組み合わせただけ、といったようなパスワードを利用している可能性はそれなりに高いです。

そのようなパスワードだけに絞って総当たりを行えばセキュリティ意識の甘いユーザーのパスワードは特定することができてしまいます。このような攻撃を**辞書攻撃**といいます。この攻撃に対する対策はプログラム側では難しく、そもそもデータベースから情報が漏洩しないようにするというぐらいしかありません。そのためユーザー側もしっかりとしたパスワードを使うことが重要になります。

さて、ハッシュ関数にはいろいろなものがあるのですが今回は**SHA-256**というハッシュ関数を利用し、その利用には**jsSHA**というライブラリを利用します。

4.3.6　クライアントの改良

それではそれらを踏まえてクライアント側を改良していきましょう。まずは「nuxt.config.js」の「env」内にハッシュ化に使うソルトとストレッチングの回数を記述しておきます。以下の内容を追記して保存してください。追記されている箇所は斜体で表示してあります。「salt」の内容については同じものを使うのはセキュリティ的に好ましくないので適宜書き換えてください。

frontend/nuxt.config.js

```
（省略）

env: {
  nameMaxLength: 16,
  nicknameMaxLength: 8,
  passwordMinLength: 8,
  salt: 'Xln&F0DZpqo1',
  stretch: 1000
},

（省略）
```

それでは「frontend/pages/register.vue」を書き換えましょう。data()内とmethods内を以下の内容で書き換えてください。data()内で新たに設定した内容を受け取るようにしている他、パスワードを事前にハッシュ化するようにしています。

frontend/pages/register.vue

```
（省略）

data() {
  return {
    // nuxt.config.jsのenvの内容を受け取る
    nameMaxLength:     process.env.nameMaxLength,
    nicknameMaxLength: process.env.nicknameMaxLength,
    passwordMinLength: process.env.passwordMinLength,
    salt:              process.env.salt,
    stretch:           process.env.stretch,

    name:             '', // キャラクターのフルネーム
    nickname:         '', // キャラクターの短縮名
    password:         '', // パスワード
    confirm:          '', // パスワード再入力\
    errorMessage:     '', // エラーメッセージ
    waitingResponse: false // 通信待ち中かどうか
  }
},
methods: {
  register: async function() {
    if (this.waitingResponse) {
      // 接続待ち状態であればアラートを表示しreturn、以降の処理を行わない
      return alert('しばらくお待ち下さい');
    }
```

```
    // 入力内容の検証を行う、問題があればエラーメッセージを表示、returnして処理を中断
    if (!this.name) { // フルネームが入力されていない
      return this.errorMessage = 'フルネームが入力されていません';
    }
    if (!this.nickname) { // 短縮名が入力されていない
      return this.errorMessage = '短縮名が入力されていません'
    }
    if (!this.password) { // パスワードが入力されていない
      return this.errorMessage = 'パスワードが入力されていません'
    }

    if (this.nameMaxLength < this.name.length) {
      return this.errorMessage = 'フルネームが長すぎます';
    }
    if (this.nicknameMaxLength < this.nickname.length) {
      return this.errorMessage = '短縮名が長すぎます';
    }
    if (this.password.length < this.passwordMinLength) {
      return this.errorMessage = 'パスワードが短すぎます';
    }
    if (this.password != this.confirm) {
      return this.errorMessage = 'パスワードと再入力の内容が一致しません';
    }

    // 問題がなければハッシュ化と接続に入る、接続待ち状態をON(true)に
    this.waitingResponse = true;

    // パスワードのハッシュ化、ソルティング、ストレッチングを行う
    let s = this.password + this.salt;
    for (let i = 0; i < this.stretch; i++) {
        const shaObj = new jsSHA("SHA-256", "TEXT");
        shaObj.update(s);
        s = shaObj.getHash("HEX");
    }

    try {
      await this.$axios.post('/api/characters', {
        name:     this.name,
        nickname: this.nickname,
        password: s
      });
      console.log('登録が完了しました');
    } catch (e) {
      // 登録に失敗した場合接続待ち状態をOFF(false)に
      this.waitingResponse = false;
      console.log('登録に失敗しました');
    }
  }
}

（省略）
```

キャラクター登録ページはこれでひとまず完成です。実際に登録してMongoDB Compassで確認してみましょう。例の画像のようにパスワードがハッシュ化されていれば成功になります。

```
_id: ObjectId("5de3ce872b0a7c506f6998a2")
> status: Object
> declare: Object
deleted: false
> tags: Array
> profileImages: Array
ap: 0
np: 20
> skill: Array
> story: Array
> explore: Array
> fav: Array
> block: Array
> blocked: Array
> mute: Array
name: "テスト"
nickname: "テスト"
password: "5a804330c4faa495a587d6d6f2c1b1959ecc5167a363b701a03853025be71ec6"
csrf: "b6bf62c350de207a068d444f895e6265cc94f7f9947d7054d4008b439cb11bb0"
registrationTime: 2019-12-01T14:30:31.491+00:00
> icons: Array
__v: 0
eno: 1
```

なお現状登録完了しても特にページ側からのリアクションがほぼなく非常に分かりづらいですが、これは後々変更します。とりあえず今はこのままで構いません。なお、ここで設定したパスワードは後で使うので覚えておいてください。

4.4 ログインの仕組みと実装

4.4.1　ログインの仕組み

キャラクターが登録できてもログインできなければ意味がありません。ということで次はログイン機能を作っていきましょう。まずはログイン機能を作る前にこれから作るログイン機能がどういう仕組みになっているかについて学んでおきます。

ログイン機能の実現には**クッキー**(cookie)という仕組みを利用します。クッキーとはWebサイトがブラウザに値をセットさせられ、ブラウザはそのWebサイトに再度アクセスした際にセットした値をWebサイト側に自動的に送信する、という仕組みのことを指します。

これにより利用者のブラウザに識別のためのキーを保存させアクセスのたびに送信させることでWebサイト側はアクセスしたのが誰であるのかを識別することができるようになります。このようなログインの仕組みおよびそのためのキーのことを**セッション**(session)と呼びます。

ログインにはこれを利用します。ユーザー側がログインしたときにWebサイト側はブラウザのクッキーにセッションキーを保存させます。するとブラウザはそのWebサイトにアクセスするたびにそのWebサイトにセッションキーを送信するようになります。また、Webサイト側ではどのセッションキーがどのユーザーかといった情報(セッション情報)を保持するようにしておきます。

これによってWebサイトにアクセスがあった際、送られてきたセッションキーを見て保持してあるセッション情報と照らし合わせ、ユーザーごとに個別の内容を送れるようになります。これがログインの仕組みになります。

4.4.2　ログインAPIの実装

それではログインAPIを作っていきましょう。その前にセッションに関する設定を「backend/config/default.json」に記入しておきましょう。以下のように追記し保存します。sessionSecretKeyについてはセッションキーの暗号化に使われるキーのため独自のものを使用してください。

backend/config/default.json

```
{
  "port": 4000,
  "directoryUrl": "/ta/api/",
  "dbName": "teikiadv",
  "dbUser": "teikiadv",
  "dbPassword": "2$)L6WK,|dwc",
  "dbPort": 17089,
  "csrfTokenLength": 32,
  "nameMaxLength": 16,
  "nicknameMaxLength": 8,
  "initialNp": 20,
  "redisPort": 6379,
  "redisSessionPrefix": "ta:sid:",
  "sessionSecretKey": "IdYM#Vf,w%ZK",
  "cookieMaxAgeDays": 14,
  "cookiePath": "/ta/"
}
```

それぞれ追加された設定の意味は以下のようになります。

redisPort	今回セッション情報の保存に使用するRedisの動作ポート デフォルトから変更していないはずなのでRedisデフォルトポートの6379を設定
redisSessionPrefix	セッション情報の保存の際にRedisの内部で付けられる接頭辞
sessionSecretKey	セッションキーの暗号化に使われるキー
cookieMaxAgeDays	クッキーが有効な日数
cookiePath	クッキーの有効範囲 "/ta/"ディレクトリ以外に送信する必要はないので設定しておく

記入したらExpressでセッション情報を扱えるようにするためミドルウェア関数を設定します。「backend/server/index.js」を以下のように書き換えましょう。

backend/server/index.js

```
const express      = require('express');
const config       = require('config');
const redis        = require('redis');
const session      = require('express-session');
const connectRedis = require('connect-redis')(session);
const api          = require('./api.js');
const app          = express();
const port         = config.port;

app.use(express.json());
app.use(session({
  store: new connectRedis({
    client: redis.createClient(),
    host:   '127.0.0.1',
    port:   config.redisPort,
    prefix: config.redisSessionPrefix
  }),
  secret: config.sessionSecretKey,
  resave: false,
  saveUninitialized: false,
  cookie: {
    path:     config.cookiePath,
    httpOnly: true,
    maxAge:   1000 * 60 * 60 * 24 * config.cookieMaxAgeDays
  }
}));

app.use(config.directoryUrl, api);
app.listen(port);
```

次に実際にログインを行うAPIを実装していきます。これにはPassportというライブラリを使用します。PassportはGoogleやTwitterでのログイン、ユーザー名とパスワードによるログインなど様々な方法によるログイン方法を提供しているライブラリです。Passportではその様々なログイン方法をストラテジーと呼び、今回使用するのは最もシンプルなユーザー名とパスワードによるストラテジーであるpassport-localです。

それではログイン機構を記述していきましょう。ログインの実際の機能部分は別ファイルに分割することにします。「backend/server」内に「passport.js」を作成します。ファイル構造は以下のようになります。

```
∨ server
  JS api.js
  JS index.js
  JS model.js
  JS passport.js
```

作成したら内容を記述します。Passportではログイン機構の他に、セッション情報を保存する際のキー(passport.serializeUser)と保存したキーからデータを復元する処理(passport.deserializeUser)も記述します。

backend/server/passport.js

```
const LocalStrategy = require("passport-local").Strategy;
const Character     = require("./model.js").Character;

module.exports = (passport) => {
  // 内部的に使われる保存の際のキー
  // 今回はenoを使用
  passport.serializeUser((character, done) => {
    done(null, character.eno);
  });

  // 内部的に保存されたキーからデータを復元
  // 今回はenoが内部キーになっているので、それでデータを検索
  passport.deserializeUser(async (eno, done) => {
    let character;
    try {
      // 削除されていない、指定のENoのキャラクターを検索
      character = await Character.findOne({
        eno:     eno,
        deleted: false
      }, {
        eno:      1,
        password: 1,
        csrf:     1
      });
    } catch (e) {
      return done(e, null);
    }

    if (!character) {
      done('Character Not Found', null);
    } else {
      done(null, {
        _id:  character._id,
        eno:  character.eno,
        csrf: character.csrf
      });
    }
  });
```

```
  // ログイン機構
  passport.use(
    new LocalStrategy({
      usernameField: 'eno', // クライアントから送信されたデータのenoをユーザー名として扱う
      passwordField: 'password' // 〃のpasswordをパスワードとして扱う
    }, async (eno, password, done) => {
      let character;
      try {
        // 削除されていない、指定のENoのキャラクターを検索
        character = await Character.findOne({
          eno:     eno,
          deleted: false
        }, {
          eno:      1,
          password: 1,
          csrf:     1
        });
      } catch (e) {
        return done(null, false, 'ログイン処理中にエラーが発生しました');
      }

      if (!character) {
        return done(null, false, 'そのENoのキャラクターは存在しません');
      } else if (password != character.password) {
        return done(null, false, 'パスワードが一致しません');
      } else {
        // 認証に成功したらセッションにユーザー情報をセットする
        done(null, {
          _id:  character._id,
          eno:  character.eno,
          csrf: character.csrf
        });
      }
    })
  );
};
```

ログイン処理が行われるときここではenoとpasswordという入力を受け取ります。enoの指定を元にしてキャラクターの検索を行い、存在しなかったりパスワードが不一致だったりすればエラーを返します。そうでなければログインしたキャラクターの内容を返し、セッション情報に保存します。

さて、次にログインに関わる設定を「backend/config/default.json」に記述します。今回はログアウトしたらどのページに移動するかを設定します。以下の設定を追加しましょう。なお、https化していない場合はhttpsの部分をhttpに書き換えておいてください。

backend/config/default.json

```
"logoutRedirect": "https://dev.siroisakana.com/ta/"
```

次にPassportをミドルウェアとして設定します。「backend/server/index.js」を以下のように書き換えて保存しま

す。

backend/server/index.js

```
const express       = require('express');
const config        = require('config');
const redis         = require('redis');
const session       = require('express-session');
const connectRedis  = require('connect-redis')(session);
const passport      = require('passport');
const localStrategy = require('./passport.js');
const api           = require('./api.js');
const app           = express();
const port          = config.port;

app.use(express.json());
app.use(session({
  store: new connectRedis({
    client: redis.createClient(),
    host:   '127.0.0.1',
    port:   config.redisPort,
    prefix: config.redisSessionPrefix
  }),
  secret: config.sessionSecretKey,
  resave: false,
  saveUninitialized: false,
  cookie: {
    path:     config.cookiePath,
    httpOnly: true,
    maxAge:   1000 * 60 * 60 * 24 * config.cookieMaxAgeDays
  }
}));

app.use(passport.initialize());
app.use(passport.session());
localStrategy(passport);

app.use(config.directoryUrl, api);
app.listen(port);
```

最後にログイン、ログアウト、認証確認のAPIを実装しましょう。それぞれ「/login」「/logout」「/auth」からアクセスできるようにします。「backend/server/api.js」の冒頭部分を以下のように書き換えます。

backend/server/api.js

```
const express   = require('express');
const config    = require('config');
const passport  = require('passport');
const Character = require('./model.js').Character;
const router    = express.Router();

const checkAuthentication = (req, res, next) => {
  // ログインしているか確認するミドルウェア関数
  if (req.user) { // ログインしていれば（req.userがあれば）次へ
    next();
```

```
  } else { // ログインしていなければ401を返して処理を中断する
    res.status(401).send();
  }
};

router.post('/login', (req, res, next) => {
  passport.authenticate('local', (err, character, info) => {
    if (err) { // 認証中にエラーが発生した場合
      return next(err);
    } else if (!character) { // 何らかの原因でキャラクターが取得できなかった場合
      return res.status(401).send(info);
    } else { // 認証に成功した場合
      req.login(character, (err) => { // ログイン処理中にエラーが発生した場合
        return next(err);
      });
      return res.status(200).send({ // 問題なければキャラクターのENoを返す
        eno: character.eno
      });
    }
  })(req, res, next);
});

router.get('/logout', (req, res) => {
  req.logout(); // ログアウトして
  res.redirect(config.logoutRedirect); // 指定のページへリダイレクト
});

router.get('/auth', checkAuthentication, (req, res) => {
  return res.status(200).send({
    eno:  req.user.eno,
    csrf: req.user.csrf
  });
});

router.post('/characters', async (req, res) => {

  （省略）
```

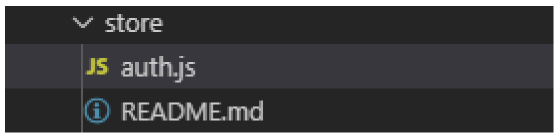
ログインではpassport-localを呼び出してログイン処理を行っています。ログインに成功するとreq.userからログインキャラクター情報が取得できるようになります。ログアウトではログアウト処理を呼び出してからリダイレクトを行っています。

またcheckAuthenticationというミドルウェア関数を定義しています。ログインしているか否かはreq.userが存在していることで見分けることができます。これによりreq.userが存在していれば、すなわちログインしていれば処理を続行、ログインしていなければ処理を中断するAPI（＝ログインしていないとアクセスできないAPI）を作ることができます。

「/auth」はこれを利用したAPIです。checkAutenticationの指定によりログインしていなければステータスコード401が、ログインしていればステータスコード200とともにログインしているキャラクターのENoとCSRFトークンが帰ってくるAPIになっています。

4.4.3 クライアント側のログイン処理の実装

ログインしているかという情報は全体から参照できたほうが便利でしょう。そのためログイン情報はVuexに格納するようにします。「frontend/store」内に「auth.js」を作成しましょう。ファイル構成は以下のようになります。



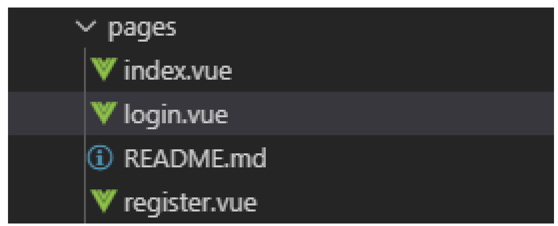
作成したら以下の内容を記述して保存します。

frontend/store/auth.js

```
const state = () => ({
  loginCharacter: null
});

const getters = {
  isAuthenticated(state) {
    return !!state.loginCharacter;
  },
  loginCharacter(state) {
    return state.loginCharacter;
  }
};

const mutations = {
  login(state, character) {
    state.loginCharacter = character;
  },
  logout(state) {
    state.loginCharacter = null;
  }
};

const actions = {
  login: async function ({commit}) {
    try {
      const response = await this.$axios.get('/api/auth'); // ログイン状態をチェック
      commit('login', response.data); // ログインしていたらデータを受け取ってログイン状態に
    } catch (e) {
      commit('logout'); // ログインしていなければログアウト状態にする
    }
  }
};

export default {
  state,
  getters,
  mutations,
  actions
}
```

「auth.js」ではloginCharacterという情報を管理します。この値は「login」というmutationで登録され「logout」というmutationで解除されます。actionsの「login」という操作からmutationsの「login」「logout」は呼び出されます。具体的にどうなっているかというと、とりあえず「/api/auth」に接続を試みてログイン状況が帰ってくればmutationsの「login」でその値をセット、ログインしていないようだったら「logout」を呼び出します。loginCharacterはgettersの「loginCharacter」からそのまま受け取れる他、「isAuthencated」からログインしているかどうかをtrue/falseで受け取ることができます。

さて、ここまで設定したら実際にログインページを作りましょう。「frontend/pages」内に「login.vue」を作成します。ファイル構成は以下のようになります。



作成したら以下の内容を記述して保存します。

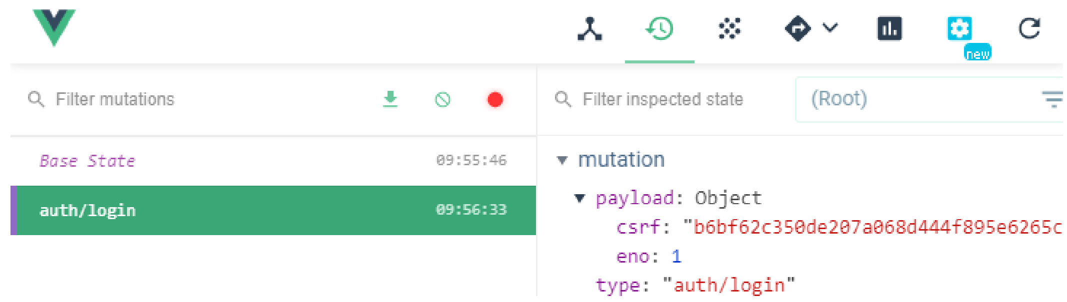
frontend/pages/login.vue

```
<template>
  <section>
    <section>
      <sub-heading>ログイン</sub-heading>
      <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
      <section class="form">
        <div class="form-title">ENo</div>
        <input class="form-input" type="text" v-model="eno" placeholder="ENo">
      </section>
      <section class="form">
        <div class="form-title">パスワード</div>
        <input class="form-input" type="password" v-model="password" placeholder="パスワード">
      </section>
      <div class="button-wrapper">
        <button class="button" @click="login">ログイン</button>
      </div>
    </section>
  </section>
</template>

<script>
import jsSHA         from 'jssha'
import SubHeading    from '~/components/SubHeading.vue'
import MessageBanner from '~/components/MessageBanner.vue'

export default {
  components: {
    SubHeading,
    MessageBanner
```

```
},
head() {
  return {
    title: 'ログイン'
  };
},
data() {
  return {
    passwordMinLength: process.env.passwordMinLength, // パスワードの最小の長さ
    salt:              process.env.salt,    // ソルト
    stretch:           process.env.stretch, // ストレッチング回数

    eno:              '', // ENo
    password:         '', // パスワード
    errorMessage:     '', // エラーメッセージ
    waitingResponse: false // 通信待ち中かどうか
  }
},
methods: {
  login: async function() {
    if (this.waitingResponse) {
      // 接続待ち状態であればアラートを表示しreturn、以降の処理を行わない
      return alert('しばらくお待ち下さい');
    }

    // 入力内容の検証を行う、問題があればエラーメッセージを表示、returnして処理を中断
    if (!this.eno) { // ENoが入力されていない
      return this.errorMessage = 'ENoが入力されていません';
    }
    if (this.password.length < this.passwordMinLength) {
      return this.errorMessage = 'パスワードが短すぎます';
    }

    // 問題がなければハッシュ化と接続に入る、接続待ち状態をON(true)に
    this.waitingResponse = true;

    // パスワードのハッシュ化、ソルティング、ストレッチングを行う
    let s = this.password + this.salt;
    for (let i = 0; i < this.stretch; i++) {
        const shaObj = new jsSHA("SHA-256", "TEXT");
        shaObj.update(s);
        s = shaObj.getHash("HEX");
    }

    try {
      // ハッシュ化されたパスワードを送信、サーバー側をログイン状態に
      await this.$axios.post('/api/login', {
        eno:      this.eno,
        password: s,
      });

      // サーバーにログインできているか問い合わせ、できていればログイン情報をVuexにセット
      await this.$store.dispatch('auth/login');
    } catch (e) {
      // エラーが発生した場合接続待ち状態をOFF(false)に
      this.waitingResponse = false;
```

```
        if (e.response && e.response.data) { // HTTP通信中のエラーかつメッセージが存在すれば
          this.errorMessage = e.response.data; // サーバーから送られたエラーメッセージを表示
        } else { // そうでなければ
          this.errorMessage = 'ログイン中にエラーが発生しました'; // 汎用エラーメッセージを表示
        }
      }
    }
  }
}
</script>
```

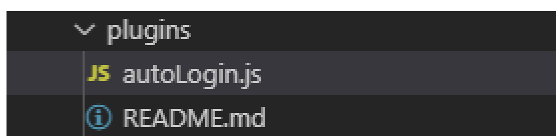
保存したら『クライアントの改良』のところで作成したキャラクターでログインしてみましょう。「http://dev.siroisakana.com/ta/login」にアクセスします。ENo.1にキャラクターが登録されているはずなのでENoのところに「1」、パスワードのところに登録時に設定したパスワードを入力してログインを押します。ページ側では特にリアクションがありませんが、Chromeの開発者ツール(F12キーを押して表示)を開きVueと書かれたタブのVuexを確認する欄からステートを確認してみましょう。以下のようになっていればログインに成功しています。



4.4.4 自動ログイン

今のままではたとえセッションの有効期間内であってもブラウザを閉じるたびに手動でのログインが必要になり非常に不便です。なのでサイトにアクセスした際に「/api/auth」にまだログインが有効であるかどうか自動で問い合わせ、有効であれば自動的にログイン状態にするようにします。これにはプラグインを利用します。

それではそのためのプラグインを作成しましょう。「frontend/plugins」内に「autoLogin.js」を作成します。ファイル構成は以下のようになります。



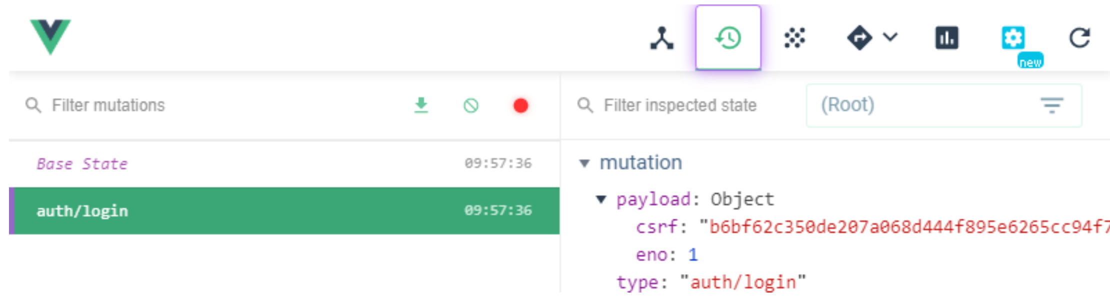
作成したら以下の内容を入力して保存しましょう。

plugins/autoLogin.js
```
export default async (context) => {
  await context.store.dispatch('auth/login');
}
```

プラグインが作成できたらNuxt.jsで読み込むようにします。「nuxt.config.js」を開き、pluginsの部分を以下のように書き換えます。

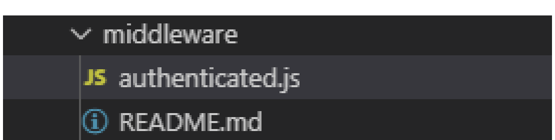
nuxt.config.js
```
（省略）

plugins: [
  '~/plugins/autoLogin.js'
],

（省略）
```

自動ログインできているか確認してみましょう。一度Chromeを閉じ、「http://dev.siroisakana.com/ta/」にアクセスしてみます。Chromeの開発中ツールからVuexを確認し、以下のように自動でログインされていれば成功です。



4.4.5　ログインが必要なページ

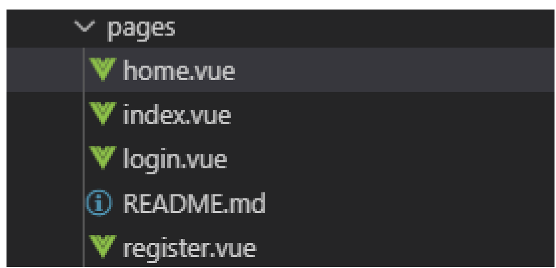
次にログインしないと見られないページを作ってみましょう。これにはmiddlewareを使用します。「frontend/middleware」内に「authenticated.js」を作成します。ファイル構造は以下のようになります。



作成したら以下の内容を記述します。Vuexを確認し、ログインしていなければログインページにリダイレクトするようになっています。

middleware/authenticated.js
```
export default function ({ store, redirect }) {
  if (!store.getters['auth/isAuthenticated']) {
    return redirect('/login');
  }
}
```

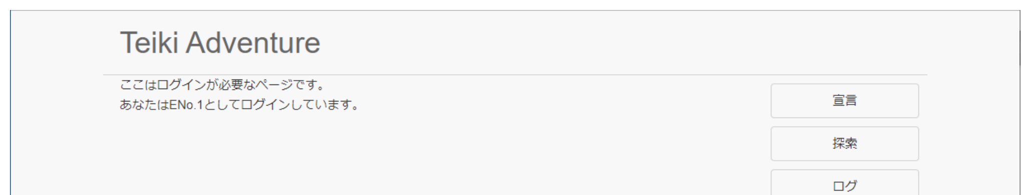
それではこのmiddlewareを利用したページを作りましょう。「frontend/pages」内に「home.vue」を作成します。ファイル構造は以下のようになります。



作成したら以下の内容を記述して保存します。

```
frontend/pages/home.vue
<template>
  <section>
    ここはログインが必要なページです。<br>
    あなたは ENo.{{ $store.getters['auth/loginCharacter'].eno }}としてログインしています。
  </section>
</template>

<script>
export default {
  middleware: 'authenticated'
}
</script>
```

「http://dev.siroisakana.com/ta/home」にアクセスすると以下のようになり、ログインできていることが確認できます。



4.4.6　ログアウト

ログアウトは単にログアウト用のURLへアクセスするだけで実現できます。「SideMenu.vue」にログアウトURLへのリンクを作りましょう。「frontend/components/SideMenu.vue」のmini-link-wrapper内を以下のように書き換え保存します。ログアウトURLはNuxt.js外になるので<nuxt-link>ではなく単純に<a>タグを利用しています。また、「nuxt.config.js」のrouterの設定内容を利用してURLを組んでいます。

```
frontend/components/SideMenu.vue
（省略）

  <div class="mini-link-wrapper">
    <nuxt-link class="mini-link" to="/inquiry">>> 問い合わせ</nuxt-link>
    <a class="mini-link" :href="`${$router.options.base}api/logout`">>> ログアウト</a>
  </div>

（省略）
```

保存したらメニューにログアウトというリンクが作られるのでクリックしてみましょう。バックエンド側でトップページに転送されるようになっているので、ログアウトされた上でトップページに転送されます。この状態で「http://dev.siroisakana.com/ta/home」にアクセスしてみます。ログアウトされているのでアクセスできずログインページに転送されるはずです。

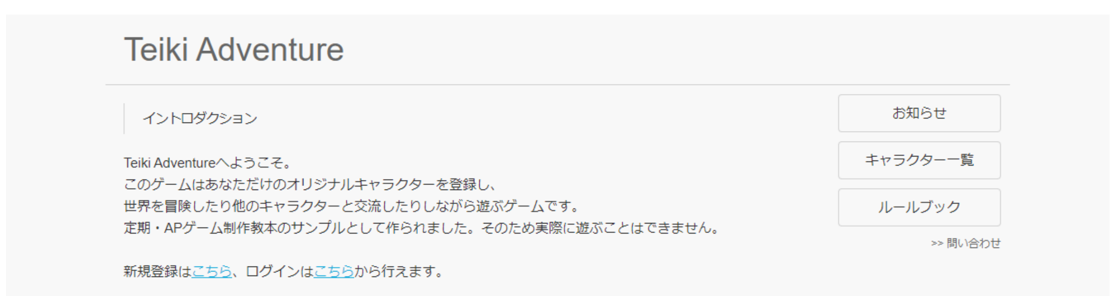
4.4.7　非ログイン状態のメニュー

ログイン中にアクセスできないメニューについては非表示にしておきましょう。「SideMenu.vue」を以下のように書き換え保存します。v-if全てにVuexへのアクセスを書いてもいいのですが、それだとすこし見づらくなるため算出プロパティを使用しています。

```
frontend/components/SideMenu.vue
<template>
  <div>
    <side-menu-button to="/notice">お知らせ</side-menu-button>
    <side-menu-button v-if="auth" to="/declare">宣言</side-menu-button>
    <side-menu-button v-if="auth" to="/explore">探索</side-menu-button>
    <side-menu-button v-if="auth" to="/log">ログ</side-menu-button>
    <side-menu-button v-if="auth" to="/skill">戦闘設定</side-menu-button>
    <side-menu-button v-if="auth" to="/talk/public">交流</side-menu-button>
    <side-menu-button v-if="auth" to="/profile/main/edit">キャラクター設定</side-menu-button>
    <side-menu-button to="/list">キャラクター一覧</side-menu-button>
    <side-menu-button to="/rulebook">ルールブック</side-menu-button>
    <div class="mini-link-wrapper">
      <nuxt-link class="mini-link" to="/inquiry">&gt;&gt; 問い合わせ</nuxt-link>
      <a v-if="auth" class="mini-link" :href="`${$router.options.base}api/logout`">&gt;&gt; ログアウト</a>
    </div>
  </div>
</template>

<script>
import SideMenuButton from '~/components/SideMenuButton.vue'

export default {
  components: {
    SideMenuButton
  },
  computed: {
    auth: {
      get() {
        return this.$store.getters['auth/isAuthenticated'];
      }
```

```
    }
  }
}
</script>

<style lang="scss" scoped>
.mini-link-wrapper {
  display: flex;
  flex-direction: column;
  align-items: flex-end;
  padding: 10px;

  .mini-link {
    text-decoration: none;
    color: #666;
    font-size: 12px;
  }
}
</style>
```

保存すると非ログイン状態ではメニューが以下のようになるはずです。



また、ログインしているときは画面一番上のタイトルをクリックしたときのリンク先がトップページではなくホームになるようにしましょう。「frontend/layouts/default.vue」のtitle部分を以下のように書き換え保存します。

```
frontend/layouts/default.vue
（省略）

    <div id="title">
      <nuxt-link id="title-logo" :to="$store.getters['auth/isAuthenticated'] ? '/home' : '/'">Teiki Adventure</nuxt-link>
    </div>

（省略）
```

4.4.8　ログイン / 登録時のリダイレクト

また、ログインや登録をしたときにホームにリダイレクトするようにしましょう。methods内でのリダイレクトにはthis.$router.pushを利用します。「frontend/pages」内の「login.vue」と「register.vue」を開き、それぞれログイン/登

録処理中のtry/catch部分を以下のように書き換え保存します。

frontend/pages/login.vue

```
(省略)

    try {
      // ハッシュ化されたパスワードを送信、サーバー側をログイン状態に
      await this.$axios.post('/api/login', {
        eno:      this.eno,
        password: s,
      });

      // サーバーにログインできているか問い合わせ、できていればログイン情報をVuexにセット
      await this.$store.dispatch('auth/login');

      // ログインしたらhomeへリダイレクト
      this.$router.push('/home');
    } catch (e) {
      // エラーが発生した場合接続待ち状態をOFF(false)に
      this.waitingResponse = false;

      if (e.response && e.response.data) { // HTTP通信中のエラーかつメッセージが存在すれば
        this.errorMessage = e.response.data; // サーバーから送られたエラーメッセージを表示
      } else { // そうでなければ
        this.errorMessage = 'ログイン中にエラーが発生しました'; // 汎用エラーメッセージを表示
      }
    }

(省略)
```

frontend/pages/register.vue

```
(省略)

    try {
      // ハッシュ化されたパスワードを送信、キャラクターを登録
      // レスポンスを受け取る
      const response = await this.$axios.post('/api/characters', {
        name:     this.name,
        nickname: this.nickname,
        password: s,
      });

      // 受け取ったレスポンスのENoを使いサーバー側をログイン状態に
      await this.$axios.post('/api/login', {
        eno:      response.data.eno,
        password: s,
      });

      // サーバーにログインできているか問い合わせ、できていればログイン情報をVuexにセット
      await this.$store.dispatch('auth/login');

      // ログインしたらhomeへリダイレクト
      this.$router.push('/home');
    } catch (e) {
      // エラーが発生した場合接続待ち状態をOFF(false)に
```

```
        this.waitingResponse = false;
        this.errorMessage = '登録中にエラーが発生しました'; // エラーメッセージを表示
      }

（省略）
```

　処理が完了したらリダイレクトを呼び出しています。登録側では登録APIからENoのレスポンスを受け取って自動でログインするようになっています。Axiosで受け取ったデータの内容は.dataから取得することができます。今回はレスポンスをresponseとして取得しており、enoを取得したいのでresponse.data.enoからENo情報を取得しています。

　ではこの状態でログインや新規登録を試してみましょう。それぞれホームにリダイレクトするようになっているはずです。

4.5 ホーム

4.5.1　ホームAPI

　次にホームの内容を作っていきましょう。ホームではキャラクターのプロフィールと宣言状況などを確認できるようにします。また、このAPIへのアクセスにはログインが必要になることとします。ログインが必要なAPIは『ログインAPIの実装』で作成したcheckAuthenticationミドルウェア関数を設定することで簡単に実装することができます。それでは実装していきましょう。「/characters/main/home」からホームの表示に必要な情報を返すことにします。他のAPIと同様に「backend/server/api.js」に以下のAPIを追記します。

backend/server/api.js

```
  （省略）

router.get('/characters/main/home', checkAuthentication, async (req, res) => {
  try {
    const character = await Character.findById(req.user._id, {
      eno:           1,
      name:          1,
      tags:          1,
      profile:       1,
      icons:         1,
      ap:            1,
      np:            1,
      status:        1,
      declare:       1,
      profileImages: 1
    });

    return res.status(200).send({
      eno:     character.eno,
      name:    character.name,
      tags:    character.tags,
      profile: character.profile,
      icons:   character.icons,
      ap:      character.ap,
      np:      character.np,
      status:  character.status,
      isDeclared: { // 宣言をしているかどうか
        diary: character.declare.diary != null,
        story: character.declare.storyId != null,
        party: character.declare.party.length != 0
      },
      profileImage: character.profileImages.length ? character.profileImages[Math.floor(Math.rando
m() * character.profileImages.length)] : null
      // プロフィール画像配列が存在するときランダムに1つを返す、なければnull
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

```
（省略）
```

ログインしている際、req.user._idにアクセスしたユーザーのキャラクターのキャラクターのドキュメントIDが格納されているのでそれを参照してキャラクター情報を取得しています。また、宣言状況やプロフィール画像などはそれぞれ必要になる最低限の情報のみを返すようにしてあります。

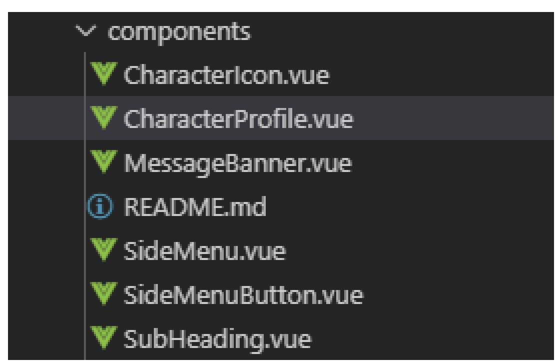
4.5.2　APIの結果を受け取ってページを表示

それでは実際にページを作っていきましょう。その前に表示に必要な設定を「nuxt.config.js」に記載しておきます。「frontend/nuxt.config.js」を開き「env」を以下のように書き換えます。ここでは設定できるアイコンの数を設定してあります。

frontend/nuxt.config.js

```
（省略）

env: {
  nameMaxLength: 16,
  nicknameMaxLength: 8,
  passwordMinLength: 8,
  salt: 'Xln&F0DZpqo1',
  stretch: 1000,
  iconsMaxLength: 30
},

（省略）
```

次に表示に必要なコンポーネントを作っていきます。ここではキャラクターアイコンとキャラクタープロフィールのコンポーネントを作っていきます。「frontend/components」内に「CharacterIcon.vue」と「CharacterProfile.vue」をそれぞれ作成してください。ファイル構成は以下のようになります。



作成したらそれぞれ以下の内容を入力して保存してください。

frontend/components/CharacterIcon.vue

```
<template>
  <div class="icon-wrapper">
```

```
    <img v-if="src" class="icon" :src="src"/>
    <div v-else class="noimage"></div>
  </div>
</template>

<script>
export default {
  props: {
    src: String
  }
}
</script>

<style lang="scss" scoped>
.icon-wrapper {
  display: block;
  width: 60px;
  height: 60px;

  .icon {
    width: 100%;
    height: 100%;
  }

  .noimage {
    box-sizing: border-box;
    background: #CCC;
    width: 100%;
    height: 100%;
  }
}
</style>
```

frontend/components/CharacterProfile.vue

```
<template>
  <section>
    <h2 class="name">ENo.{{ eno }} {{ name }}</h2>
    <section class="main">
      <section class="image-wrapper">
        <img v-if="profileImage" class="image" :src="profileImage">
      </section>
      <section class="summary">
        <div v-if="mode == 'home'" class="ap-wrapper">
          <div class="ap">AP</div>
          <div class="ap-value">{{ ap }}</div>
        </div>
        <table class="statuses">
          <tbody>
            <tr>
              <td class="status-name">ATK</td>
              <td class="status-value">{{ status.atk }}</td>
              <td class="status-name">DEX</td>
              <td class="status-value">{{ status.dex }}</td>
              <td class="status-name">MND</td>
              <td class="status-value">{{ status.mnd }}</td>
            </tr>
```

```
        <tr>
          <td class="status-name">AGI</td>
          <td class="status-value">{{ status.agi }}</td>
          <td class="status-name">DEF</td>
          <td class="status-value">{{ status.def }}</td>
        </tr>
      </tbody>
    </table>
    <div class="skills">
      <div class="skill">
        <div class="skill-prop">
          <div class="skill-name">フレア</div>
          <div class="skill-cond">50SP / 9行動毎</div>
        </div>
        <div class="skill-effect">敵：攻撃＋SPが30%以上なら敵全：攻撃</div>
      </div>
      <div class="skill">
        <div class="skill-prop">
          <div class="skill-name">エリアヒール</div>
          <div class="skill-cond">50SP / 味方重傷</div>
        </div>
        <div class="skill-effect">味全：HP回復</div>
      </div>
      <div class="skill">
        <div class="skill-prop">
          <div class="skill-name">反撃</div>
          <div class="skill-cond">攻撃回避時</div>
        </div>
        <div class="skill-effect">敵：攻撃</div>
      </div>
      <div class="skill">
        <div class="skill-prop">
          <div class="skill-name">シフトアップ</div>
          <div class="skill-cond">戦闘開始時</div>
        </div>
        <div class="skill-effect">自：AGI増</div>
      </div>
      <div class="skill">
        <div class="skill-prop">
          <div class="skill-name">自己修復</div>
          <div class="skill-cond">被攻撃時</div>
        </div>
        <div class="skill-effect">自：HP回復</div>
      </div>
    </div>
  </section>
</section>
<section v-if="tags.length">
  <span class="tag" v-for="(tag, index) in tags" :key="index">{{ tag }}</span>
</section>
<section>
  <sub-heading>プロフィール</sub-heading>
  <div class="profile">{{ profile }}</div>
</section>
<section>
  <sub-heading>アイコン</sub-heading>
  <div class="character-icons-wrapper">
```

```
        <div class="character-icon-wrapper" v-for="i in iconsMaxLength" :key="i">
          <character-icon :src="icons[i - 1] ? icons[i - 1].url : ''"/>
        </div>
      </div>
    </section>
  </section>
</template>

<script>
import SubHeading    from '~/components/SubHeading.vue'
import CharacterIcon from '~/components/CharacterIcon.vue'

export default {
  components: {
    SubHeading,
    CharacterIcon
  },
  props: {
    mode:         String, // 表示モード home = ホーム、profile = キャラクターページ、preview = プレビュー
    eno:          Number,
    name:         String,
    tags:         Array,
    profile:      String,
    profileImage: String,
    icons:        Array,
    ap:           Number,
    status:       Object,
  },
  data() {
    return {
      iconsMaxLength: process.env.iconsMaxLength
    };
  }
}
</script>

<style lang="scss" scoped>
$font: 'BIZ UDPGothic', 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ', Meiryo, 'ＭＳ Ｐゴシック
', sans-serif;

.name {
  display: block;
  background: #444;
  color: #EEE;
  margin: 10px 0;
  padding: 10px;
  font-size: 18px;
  font-family: $font;
  letter-spacing: 0;
}

.main {
  display: flex;

  .image-wrapper {
    flex-grow: 0;
    flex-shrink: 0;
```

```
  width: 400px;
  height: 600px;

  .image {
    width: 100%;
    height: 100%;
  }
}

.summary {
  flex-grow: 1;
  padding: 10px;
  display: flex;
  align-items: center;
  justify-content: center;
  flex-direction: column;

  .ap-wrapper {
    width: 100%;
    display: flex;
    align-items: baseline;
    justify-content: center;
    font-family: $font;

    .ap {
      font-weight: bold;
      font-size: 24px;
      color: #AAA;
    }

    .ap-value {
      margin-left: 40px;
      font-weight: bold;
      font-size: 48px;
      color: #666;
    }
  }

  .statuses {
    box-sizing: border-box;
    margin: 0;
    padding: 12px 20px;
    width: 100%;
    border-top: 1px solid lightgray;
    border-bottom: 1px solid lightgray;
    font-family: $font;

    td {
      border: none;
      padding: 6px 12px;
    }

    .status-name {
      font-weight: bold;
      color: #555;
      margin-right: 20px;
    }
```

```
    }
  }
}

.skills {
  box-sizing: border-box;
  width: 100%;
  padding: 16px 24px 0 24px;

  .skill {
    margin-bottom: 12px;

    .skill-prop {
      display: flex;
      align-items: baseline;

      .skill-name {
        font-weight: bold;
        color: #555;
        font-size: 18px;
        margin-right: 10px;
      }

      .skill-cond {
        font-size: 14px;
        color: #888;
      }
    }

    .skill-effect {
      margin-left: 10px;
      color: #333;
    }

    &:last-child {
      margin: 0;
    }
  }
}

.tag {
  display: inline-flex;
  justify-content: center;
  min-width: 50px;
  padding: 4px;
  border: 1px solid #333;
  border-radius: 4px;
  margin-right: 10px;
  text-decoration: none;
  font-weight: bold;
  color: #333;
}

.profile {
  margin: 0 20px;
}
```

```
.character-icons-wrapper {
  display: flex;
  flex-wrap: wrap;
  justify-content: center;

  .character-icon-wrapper {
    margin: 5px;
  }
}
</style>
```

それぞれ受け取った値を表示するだけのコンポーネントです。キャラクターアイコンではsrc属性でアイコンのURLを指定します。受け取った値が空文字列だったりすると代わりに灰色のボックスを表示します。

「CharacterProfile.vue」は値を受け取ってキャラクタープロフィールを表示します。これは「ホーム」と「キャラクターリストからアクセスするキャラクターページ」で使いまわしますが、所持APを表示するか等その表示内容は若干異なるためmodeという属性を作りその値によって表示を変えるようにしてあります。なお、まだスキル関連の表示APIはありませんが、ひとまず見た目のテスト用にスキルについても仮表示を行っています。

「CharacterProfile.vue」のアイコンにまつわる部分では要素ではなく数に対してv-forを行っています。特定の配列の要素を繰り返し表示するのではなく指定の回数繰り返したい、といった場合にはこのようにします。ここでは要素の数に関わらずアイコンの入力欄などの表示をiconsMaxLength(30)個分行っています。

注意点として、JavaScriptなどで「for (let i = 0; i < 30; i++)」というようにするとiは0から始まり29で終わりますが、「v-for="i in 30"」としたときiは1から始まり30で終わります。配列のインデックスにはi - 1というように参照しなければなりません。

さて、それでは実際にページを作っていきましょう。ホームなどのページではAPIにアクセスしてからページの内容を表示しなければなりません。そのようなときに使うのがasyncDataです。Nuxt.jsではasyncData内でAPIにアクセスしてデータを受け取ることでAPIへのアクセスを待ってからページを表示することができます。asyncDataから値をreturnで返すことで他の所でdata()同様に扱うことができます。注意点として、ページのコンポーネントでしか使うことができません。(components内のコンポーネントでは使えません。)

axiosをインストールしてあるのでasyncDataで受け取れる値であるcontextからcontext.$axiosとすることでaxiosを使うことができるようになっています。さて、APIへのアクセスは以下のように行います。この例では先程作ったAPIにアクセスしています。インターネットへのアクセスなのでasync/awaitを使っていることに留意してください。

```
asyncData: async function(context) {
  const response = await context.$axios.get('/api/characters/main/home');
  return {
    eno:          response.data.eno,
    name:         response.data.name,
    tags:         response.data.tags,
    profile:      response.data.profile,
    profileImage: response.data.profileImage,
    icons:        response.data.icons,
    ap:           response.data.ap,
    np:           response.data.np,
```

```
      isDeclared:   response.data.isDeclared,
      status:       response.data.status
    }
  }
```

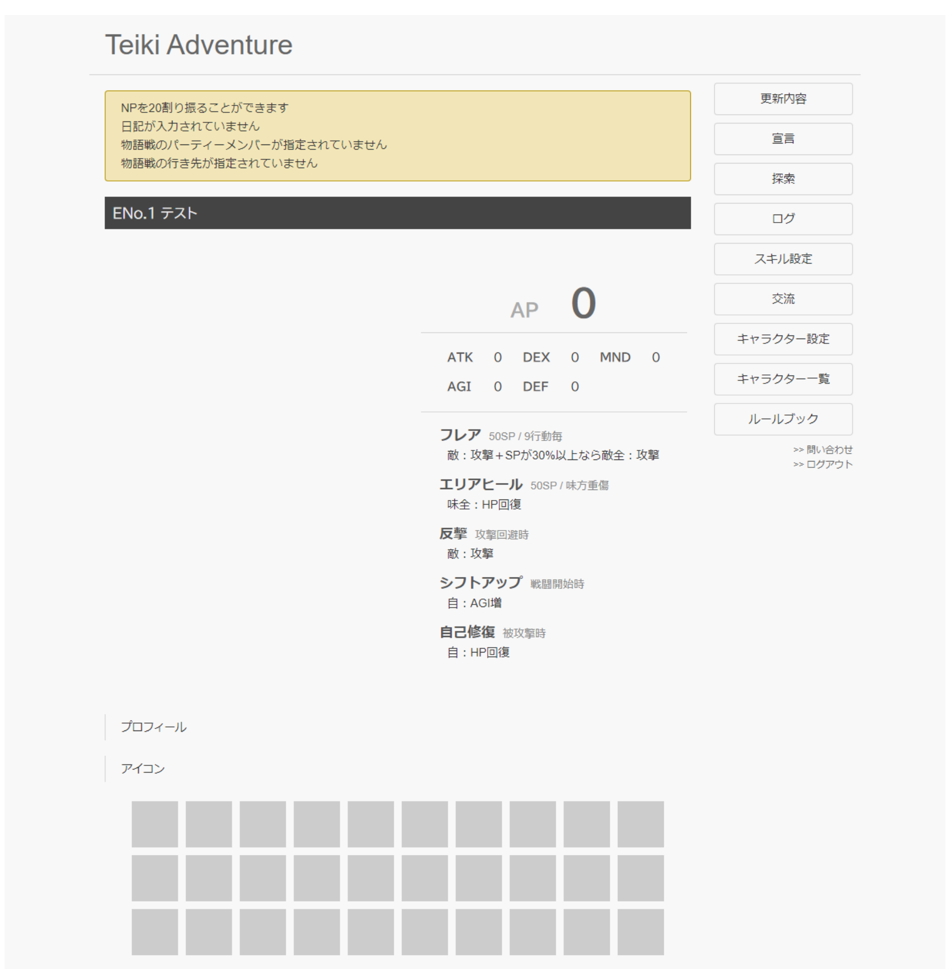
それではそれを踏まえてホームを作っていきましょう。「frontend/pages/home.vue」を以下のように書き換え保存します。

frontend/pages/home.vue

```
<template>
  <section>
    <message-banner v-if="np || !isDeclared.diary || !isDeclared.story || !isDeclared.party" type=
"warning">
      <div v-if="np">
        NP を{{ np }}割り振ることができます
      </div>
      <div v-if="!isDeclared.diary">
        日記が入力されていません
      </div>
      <div v-if="!isDeclared.story">
        物語戦の行き先が指定されていません
      </div>
      <div v-if="!isDeclared.party">
        物語戦のパーティーメンバーが指定されていません
      </div>
    </message-banner>
    <character-profile
      mode="home"
      :eno="eno"
      :name="name"
      :tags="tags"
      :profile="profile"
      :profileImage="profileImage"
      :icons="icons"
      :ap="ap"
      :status="status" />
  </section>
</template>

<script>
import MessageBanner    from '~/components/MessageBanner.vue'
import CharacterProfile from '~/components/CharacterProfile.vue'

export default {
  components: {
    MessageBanner,
    CharacterProfile
  },
  middleware: 'authenticated',
  head() {
    return {
      title: `ENo.${this.eno} ${this.name}`
    };
  },
```

```
  asyncData: async function(context) {
    const response = await context.$axios.get('/api/characters/main/home');
    return {
      eno:         response.data.eno,
      name:        response.data.name,
      tags:        response.data.tags,
      profile:     response.data.profile,
      profileImage: response.data.profileImage,
      icons:       response.data.icons,
      ap:          response.data.ap,
      np:          response.data.np,
      isDeclared:  response.data.isDeclared,
      status:      response.data.status
    }
  }
}
</script>
```

保存したら「http://dev.siroisakana.com/ta/home」にログインした状態でアクセスしてみましょう。以下のようになるはずです。



このページは<script>内の「middleware: 'authenticated'」の指定によりログインしていなければ表示できま

せん。またページを表示した際にasyncDataから先程作成したAPIへのアクセスが走り表示に必要なデータを読み取り、その値をCharacterProfileコンポーネントにわたすことでキャラクタープロフィールを表示しています。また、割り振れるNPがあったり宣言がまだだったりした場合はメッセージを表示するようにしてあります。登録当初は割り振れるNPがあり宣言もまだのはずなので色々とメッセージが表示されるはずです。

4.6 キャラクターリスト

4.6.1　キャラクターリストAPIの作成

それでは次にキャラクター一覧ページを作りましょう。キャラクター一覧ページではENo.1～100、ENo.101～200というふうにページごとに指定範囲のENoを表示できるようにします。具体的には以下のようなURLでAPIにアクセスできるようにしていきます。

URL	ENo
/api/characters?page=0 /api/characters	ENo. 1 ～ 100
/api/characters?page=1	ENo. 101 ～ 200
/api/characters?page=2	ENo. 201 ～ 300
…	…
/api/characters?page=n	ENo. n×100+1 ～ (n+1)×100

URLでは?以降に「変数名=値」というように付け加えることでサーバーに値を渡すことができます。これを**URLパラメーター**と呼びます。この例ではpageに何ページ目を取得したいかという情報が格納されるようになります。それでは実装していきましょう。まずは1ページあたりに表示するキャラクターの数を設定ファイルに記述します。「backend/config/default.json」に以下の内容を追記して保存してください。

backend/config/default.json

```
"characterListItemsPerPage": 100
```

保存したら「backend/server/api.js」に以下のAPIを追記します。「/characters」にGETでアクセスされることでキャラクター配列を返します。

backend/server/api.js

```
（省略）

router.get('/characters', async (req, res) => {
  try {
    const currentPage = Number(req.query.page) || 0;
    const characters = await Character.find({
      deleted: false, // 削除されておらず
      eno: { // ENoがページの範囲内のものを検索
        $gte: currentPage * config.characterListItemsPerPage + 1,
        $lte: (currentPage + 1) * config.characterListItemsPerPage
      }
    }).select({
      _id:      0,
      eno:      1,
      nickname: 1,
```

```
      mainicon: 1,
      tags:     1,
      summary:  1,
      status:   1
    }).exec();

    const lastCharacter = await Character.find({
      deleted: false,
      eno:     {$gt: 1}
    }).sort({eno: -1}).select({_id: 0, eno: 1}).limit(1).lean().exec();
    // 最後のキャラクターのENoを取得

    const maxEno = lastCharacter[0] ? lastCharacter[0].eno : 0;
    // キャラが1人も登録されていないときはmaxEno = 0;とする
    // （1人も登録されていないときはlastCharacter[0] = undefinedでenoを参照しようとしたときエラーになる）

    return res.status(200).send({
      list:    characters,
      maxEno: maxEno
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});

（省略）
```

処理を追っていきましょう。ExpressではURLパラメーターで渡された値はreq.queryから受け取ることができます。今回はpageに値が格納されているはずなのでreq.query.pageから取得しています。req.queryから受け取れる値は文字列になっているのでNumberで値を数値に変換してcurrentPageに格納しています。また、pageが指定されていない場合もありますが、そのようなときは0になるようにしてあります。「||」はifなどの条件のときにOR条件として使ったりしますが、その本当の役割は前の値がfalseとみなせる値の場合後の値を返すというものです。（後の値もfalseとみなせる場合は全体としてfalseとなるためORとしても使えるというわけです。）そのためこの書き方でデフォルト値を0にすることができます。

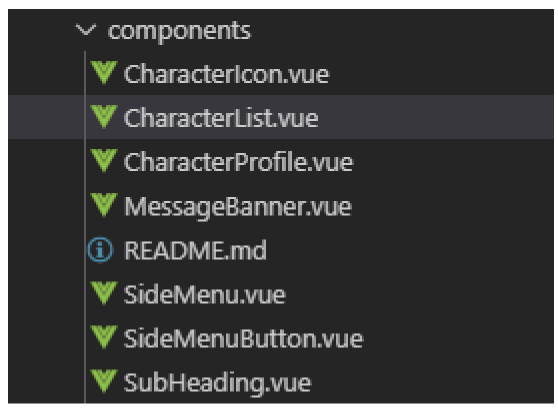
次に検索条件です。削除されておらず、ENoの範囲が「ページ×100+1」～「(ページ+1)×100」のものを検索しキャラクターリストに必要な値を取り出しています。これは単純ですね。

次は最大ENoを取得する処理です。ENoがどこまで続くのかがわからないとページへのリンクリストを作ることができないため取得しています。ここでは削除されていなくてENoが1以上のものを取得し、それをENo降順に並べ替えて最初の1件のみを取得しています。これによって取得されたドキュメントはENoが削除されていない全キャラクター中ENoが最大のキャラクターになります。なお、1人もキャラクターが登録されていないときは結果がundefinedになってしまいます。そのため取得したドキュメントのenoを参照する前にundefinedになっていないか確認し、undefinedだった場合はmaxEno = 0;となるようにしてあります。

以上の処理が終わったらキャラクターリストと最大ENoを返します。これでキャラクターリストAPIの作成は完了です。

4.6.2　キャラクターリストの作成

それではAPIにアクセスしてキャラクターリストを表示できるようにしましょう。ページを作る前にまずはキャラクターリストのコンポーネントを作成します。「frontend/components」内に「CharacterList.vue」を作成します。ファイル構成は以下のようになります



作成したら以下の内容を記入して保存します。CharacterListではキャラクター配列を受け取ってv-forで繰り返し表示することでキャラクターリストを表示しています。

frontend/components/CharacterList.vue

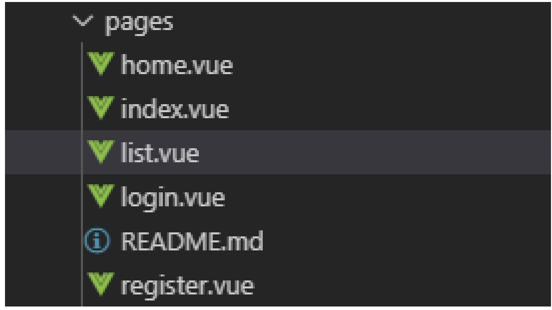
```
<template>
  <ul class="character-list">
    <li v-for="character in characters" :key="character.eno">
      <character-icon :src="character.mainicon"/>
      <div class="profile">
        <div class="info">
          <nuxt-link class="profile-link" :to="`/profile/${character.eno}`">
            <span class="name">{{ character.nickname }}</span>
            <span class="eno">&lt; ENo.{{ character.eno }} &gt;</span>
          </nuxt-link>
        </div>
        <div class="tags">
          <span v-for="tag in character.tags" :key="tag">
            {{ tag }}
          </span>
        </div>
        <div class="summary">
          {{ character.summary }}
        </div>
      </div>
    </li>
  </ul>
</template>

<script>
import CharacterIcon from '~/components/CharacterIcon.vue'

export default {
  components: {
```

```
    CharacterIcon
  },
  props: ['characters']
}
</script>

<style lang="scss" scoped>
.character-list {
  list-style: none;
  border-top: 1px solid lightgray;
  margin: 20px;

  li {
    border-bottom: 1px solid lightgray;
    display: flex;
    margin: 0;
    padding: 10px;

    .profile {
      padding-left: 10px;

      .profile-link {
        text-decoration: none;
      }

      .name {
        font-size: 16px;
        font-weight: bold;
        color: #222222;
      }

      .eno {
        font-size: 13px;
        margin-left: 3px;
        color: gray;
      }

      .tags {
        font-size: 14px;
        text-decoration: none;
        color: gray;
      }
    }
  }
}
</style>
```

次にキャラクターリストページを作ります。まずはキャラクターリストページの1ページあたりに表示するキャラクター数を「nuxt.config.js」に設定します。「frontend/nuxt.config.js」を開き「env」に以下の内容を追記して保存してください。なおこの数字はバックエンド側の設定と合わせるようにしてください。違う値を設定してしまうと整合性がとれなくなり表示がおかしくなってしまいます。

```
frontend/nuxt.config.js
    characterListItemsPerPage: 100
```

設定したら実際にページを作りましょう。「frontend/pages」内に「list.vue」を作成します。ファイル構成は以下のようになります。



作成したら以下の内容を記入して保存します。

```
frontend/pages/list.vue
<template>
  <section>
    <sub-heading>キャラクター一覧</sub-heading>
    <div class="pagelink-wrapper pagelink-head">
      <nuxt-link
        v-for="pagelink in pagelinks"
        :key="pagelink.index"
        :class="['pagelink', {current: pagelink.isCurrent}]"
        :to="`?page=${pagelink.index}`">
        {{ pagelink.desc }}
      </nuxt-link>
    </div>
    <character-list :characters="list"/>
    <div class="pagelink-wrapper pagelink-foot">
      <nuxt-link
        v-for="pagelink in pagelinks"
        :key="pagelink.index"
        :class="['pagelink', {current: pagelink.isCurrent}]"
        :to="`?page=${pagelink.index}`">
        {{ pagelink.desc }}
      </nuxt-link>
    </div>
  </section>
</template>

<script>
import SubHeading    from '~/components/SubHeading.vue'
import CharacterList from '~/components/CharacterList.vue'

export default {
  components: {
    SubHeading,
```

```
    CharacterList
  },
  watchQuery: true,
  head() {
    return {
      title: 'キャラクター一覧'
    }
  },
  asyncData: async function(context) {
    const response = await context.$axios.get(`/api/characters?page=${context.query.page}`);
    const pagelinks = [];

    let i = 0;
    do {
      pagelinks.push({
        index:     i,
        desc:      `${i * context.env.characterListItemsPerPage + 1}-`,
        isCurrent: i == (context.query.page || 0)
      });

      i++;
    } while (i * context.env.characterListItemsPerPage < response.data.maxEno);

    return {
      list:      response.data.list,
      pagelinks: pagelinks
    }
  }
};
</script>

<style lang="scss" scoped>
.pagelink-wrapper {
  padding: 10px;

  .pagelink {
    display: inline-flex;
    justify-content: center;
    min-width: 50px;
    padding: 4px;
    border: 1px solid #333;
    border-radius: 4px;
    margin-right: 10px;
    text-decoration: none;
    font-weight: bold;
    color: #666;
  }

  .current {
    background-color: #666;
    color: #EEE;
  }
}
</style>
```

「http://dev.siroisakana.com/ta/list」にアクセスしてみましょう。右メニューの「キャラクター一覧」からもアクセスすることができます。ページを表示すると以下のような感じになっているはずです。（この例では5キャラクター登録してあります。）

処理の流れを追っていきます。まずasyncData内で「/api/characters」のAPIへアクセスしています。Nuxt.jsのasyncData内ではcontext.queryからURLパラメーターを取得することができます。そのためこうすることでURLパラメーターをそのままAPIに渡すことができます。例えば「http://dev.siroisakana.com/ta/list?page=2」にアクセスすれば「/api/characters?page=2」というようにAPIにアクセスされる、というわけです。

次にページへのリンクを作る処理になっています。ここでは受け取ったmaxEnoをそのまま使うのではなくmaxEnoの情報を元にして作るべきリンクの配列pagelinksを作成しています。このようにしたほうが<template>での取り回しが楽になるためです。ここではindexにリンク先のページ番号、descにリンクに表示する内容、isCurrentにリンク先が今のページかどうかを格納しています。

そしてこの情報を元にして<template>で表示を行っています。pagelinkはv-forで繰り返し表示しており、isCurrentがtrueのときcurrentというクラスが付与されるようにしてあります。これで現在のページへのリンクは色を変えて表示できるようになっています。また、CharacterListも表示してある他、CharacterListは長いのでその後にもリンクリストを作成するようにしています。

また、注目するべきなのは<script>内にこっそり追加されている「watchQuery: true」という指定です。実はNuxt.jsはデフォルトではURLパラメーターの違いを確認していません。そのためデフォルトのままだと「http://dev.siroisakana.com/ta/list?page=1」のリンクから「http://dev.siroisakana.com/ta/list?page=2」に移動したときもページの内容が更新されないままです。そこで指定するのがwatchQueryで、これにtrueを指定することでURLパラメーターの違いをきちんと認識してページの内容を更新してくれるようになります。

4.6.3 キャラクターページAPIの作成

キャラクターリストを作ったらそれぞれのキャラクターページを作っていきましょう。「http://dev.siroisakana.co

m/ta/profile/(ENo.)」というように動的なルーティングを作成するようにします。また、ENo.のかわりに"main"と指定することで自身のキャラクターページを取得するようにします。これにより「http://dev.siroisakana.com/ta/profile/main」へのリンクを行うだけで簡単に自身のキャラクターページの表示テストを行えるようになります。ここではENoもしくはmainが指定される部分のことをkeyとして扱うことにしましょう。

さて、それではそれを踏まえてまずはAPIから作っていきましょう。バックエンド側でも同様にkeyで動的なルーティングのAPIを作ることにします。ここでは「/characters/(key)」で指定のキャラクターページの表示に必要な情報を返します。「backend/server/api.js」に以下のAPIを追加して保存してください。

backend/server/api.js

```
  (省略)

router.get('/characters/:key(\\d+|main\)', async (req, res) => {
  if (/main/.test(req.params.key) && !req.user) {
    return res.status(401).send(); // 自キャラ表示なのにログインしていなければ401
  }

  const query = /main/.test(req.params.key) ? {_id: req.user._id} : {eno: Number(req.params.key)};
  // 自キャラ表示であればセッションユーザーのキャラ、そうでなければ指定のENoで検索

  query.deleted = false;
  // 削除フラグが立っていないことを検索条件に追加

  try {
    const character = await Character.findOne(query, {
      _id:           0,
      eno:           1,
      name:          1,
      tags:          1,
      profile:       1,
      icons:         1,
      status:        1,
      profileImages: 1
    });

    if (!character) {
      return res.status(404).send(); // キャラクターの検索結果が存在しなければ404
    } else {
      return res.status(200).send({ // そうでなければ結果を返す
        eno:     character.eno,
        name:    character.name,
        tags:    character.tags,
        profile: character.profile,
        icons:   character.icons,
        status:  character.status,
        profileImage: character.profileImages.length ? character.profileImages[Math.floor(Math.ran
dom() * character.profileImages.length)] : null
      });
    }
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

```
（省略）
```

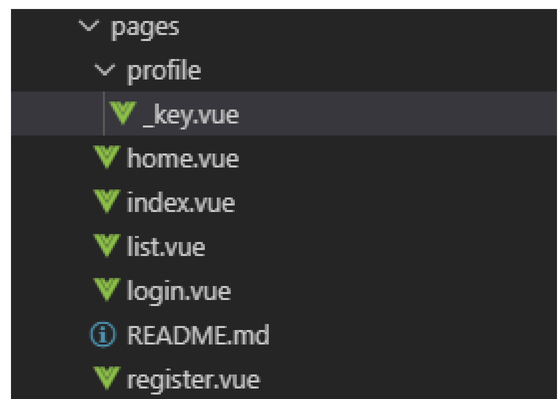
Expressでは:(セミコロン)を使うことで動的なルーティングを表現することができ、()でルーティングのキーの内容の指定を行うことも可能です。ここではkeyを値として動的なルーティングを行い、正規表現によってkeyの内容を数値もしくはmainに絞り込んでいます。ルーティングのキーはreq.paramsから取得できます。今回はkeyなので、req.params.keyがその値の内容です。

APIではまずログインチェックを行います。mainへのアクセスにも関わらずログインしていない場合は401を返しています。特に問題がなければ取得するべきキャラクターを返します。mainが指定されていればログインしているユーザーのキャラクター、そうでなければ指定のENoのキャラクターを取得します。また、検索条件にキャラクターが削除されていないことという条件も加えておきます。

検索条件が整ったらDBから値を取得して返します。なお、キャラクターが削除されていたりENoが範囲外だったりなどの理由で指定のキャラクターが存在しないことがあるのでその場合は404を返すようにしています。

4.6.4　キャラクターページの作成

APIを作成したらキャラクターページを作りましょう。今回は「profile」ディレクトリ内に動的なルーティングを作成します。まずは「frontend/pages」内に「profile」ディレクトリを作成し、さらにその中に「_key.vue」を作成しましょう。ファイル構成は以下のようになります。



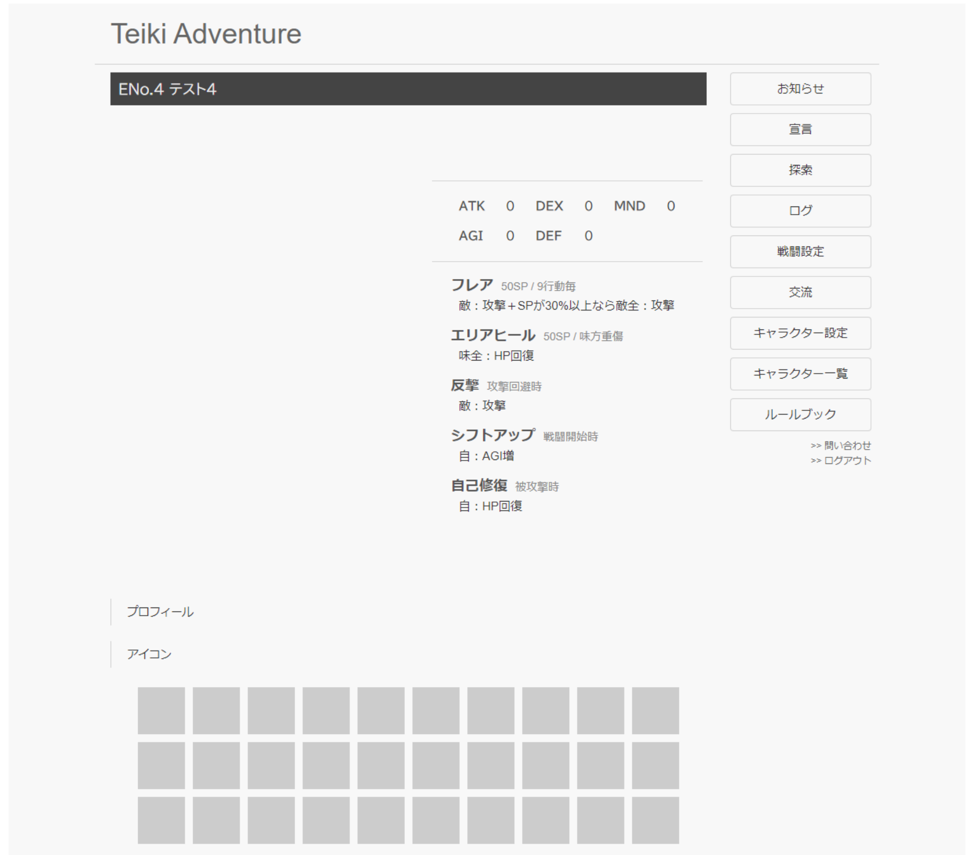
作成したら以下の内容を記入して保存します。

frontend/pages/profile/_key.vue

```
<template>
  <section>
    <character-profile
      mode="profile"
      :eno="eno"
      :name="name"
      :tags="tags"
      :profile="profile"
      :profileImage="profileImage"
      :icons="icons"
```

```
      :status="status" />
  </section>
</template>

<script>
import CharacterProfile from '~/components/CharacterProfile.vue'

export default {
  components: {
    CharacterProfile
  },
  validate({params}) {
    return /^[1-9][0-9]*$/.test(params.key) || params.key == "main";
  },
  head() {
    return {
      title: `ENo.${this.eno} ${this.name}`
    };
  },
  asyncData: async function(context) {
    const response = await context.$axios.get(`/api/characters/${context.params.key}`);
    return {
      eno:          response.data.eno,
      name:         response.data.name,
      tags:         response.data.tags,
      profile:      response.data.profile,
      profileImage: response.data.profileImage,
      icons:        response.data.icons,
      status:       response.data.status
    }
  }
}
</script>
```

ほぼCharacterProfileコンポーネントを呼び出すだけなのでかなりシンプルです。validateでENo(自然数)とmainしかkeyに指定できないようにしています。また、Nuxt.jsのasyncData内では動的なルーティングのキーはcontext.paramsからアクセスできる「/api/characters/${context.params.key}」というようにURLへkeyを連結することでkeyをAPIへそのまま渡して呼び出しています。ページを表示してみると以下のような感じになります。


Teiki Adventure
ENo.4 テスト4
ATK 0 DEX 0 MND 0
AGI 0 DEF 0
フレア 50SP / 9行動毎
敵：攻撃＋SPが30%以上なら敵全：攻撃
エリアヒール 50SP / 味方重傷
味全：HP回復
反撃 攻撃回避時
敵：攻撃
シフトアップ 戦闘開始時
自：AGI増
自己修復 被攻撃時
自：HP回復
プロフィール
アイコン
お知らせ
宣言
探索
ログ
戦闘設定
交流
キャラクター設定
キャラクター一覧
ルールブック
>> 問い合わせ
>> ログアウト

4.7 キャラクター設定

4.7.1　装飾システムの作成

キャラクターページを作ったので次はプロフィールの内容を編集できるページを作っていきましょう。プロフィール文は何かしら装飾できた方が楽しいので、まずプロフィール文の装飾システムを作成します。プロフィール文は以下の書式が使えるものとします。

<s> 〜 </s>	囲んだ文字を小さくする
<l> 〜 </l>	囲んだ文字を大きくする
 〜 	囲んだ文字を太くする

装飾システムを作る際は元の文字列から指定の装飾を読み取り、それをHTMLに変換することで実装できます。それではバックエンド側に装飾システムを作っていきましょう。装飾システムは別ファイルに分割することにします。「backend/server」内に「styling.js」を作成してください。ファイル構成は以下のようになります。

```
server
  api.js
  index.js
  model.js
  passport.js
  styling.js
```

作成したら以下の内容を入力して保存してください。

backend/server/styling.js

```
const profile = (str) => {
  // 受け取った文字列をサニタイズして指定のタグで装飾し、改行させる
  // <s> 〜 </s> 文字を小さくする
  // <l> 〜 </l> 文字を大きくする
  // <b> 〜 </b> 文字を太くする

  if (!str) {
    return ''; // 変換するものがなければ空文字列を返す
  }

  return (
    str
      .replace(/<s>(.+)<\/s>/sig , '<span class="small">$1</span>')
      .replace(/<l>(.+)<\/l>/sig , '<span class="large">$1</span>')
      .replace(/<b>(.+)<\/b>/sig , '<span class="bold">$1</span>')
      .replace(/\n/g, '<br>')
  );
};
```

```
module.exports = {
  profile: profile
};
```

ここでは指定のタグをそれぞれclassが付与された<span>に変換している他、改行を
に変換しています。

フロントエンド側でこの装飾を表示するために「frontend/assets/css/theme.scss」の末尾に以下のスタイル設定を書き加えて保存しておいてください。

```
frontend/assets/css/theme.scss
  （省略）

.small {
  font-size: 66%;
}

.large {
  font-size: 150%;
}

.bold {
  font-weight: bold;
}
```

4.7.2 キャラクター設定APIの作成

それでは実際にキャラクター設定APIを作ります。「/characters/main/profile」にGETリクエストでアクセスすることでキャラクター設定ページの表示に必要な値を返し、PUTリクエストでアクセスすることで指定の値で情報をアップデートすることにします。「backend/server/api.js」の冒頭部分で「styling.js」を読み込むようにしてから、他のAPI同様に以下の2つのAPIを追記して保存してください。

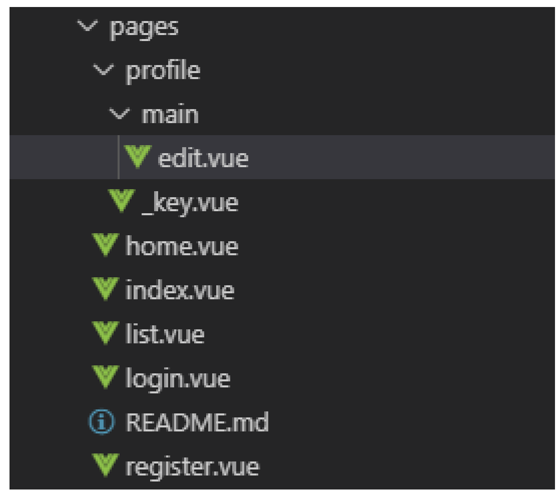
```
backend/server/api.js
const express   = require('express');
const config    = require('config');
const passport  = require('passport');
const styling   = require('./styling.js');
const Character = require('./model.js').Character;
const router    = express.Router();

  （省略）

router.get('/characters/main/profile', checkAuthentication, async (req, res) => {
  try {
    const character = await Character.findById(req.user._id, {
      _id:          0,
      eno:          1,
      name:         1,
      nickname:     1,
      tags:         1,
```

```
      summary:       1,
      mainicon:      1,
      profileImages: 1,
      profile:       1,
      icons:         1,
      status:        1
    });

    return res.status(200).send(character);
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});

router.put('/characters/main/profile', checkAuthentication, async (req, res) => {
  try {
    if (config.iconsMaxLength < req.body.icons.length) { // アイコンの数が多すぎる場合
      return res.status(400).send();
    }

    await Character.findByIdAndUpdate(req.user._id, {
      name:          req.body.name,
      nickname:      req.body.nickname,
      tags:          req.body.tags,
      summary:       req.body.summary,
      mainicon:      req.body.mainicon,
      profile:       styling.profile(req.body.profile),
      icons:         req.body.icons,
      profileImages: req.body.profileImages
    });

    return res.status(200).send();
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});

  （省略）
```

GET側ではMongoDBから値を取得してそのまま返しており、PUT側ではプロフィール文の装飾を行ってから情報の更新を行っています。

4.7.3　キャラクター設定ページの作成

それではキャラクター設定ページを作ります。「http://dev.siroisakana.com/ta/profile/main/edit」というURLでキャラクター設定ページにアクセスできるようにしましょう。「frontend/pages/profile」内に「main」ディレクトリを作成し、さらにその中に「edit.vue」を作成しましょう。ファイル構成は以下のようになります。



作成したら以下の内容を記述して保存してください。

frontend/pages/profile/main/edit.vue

```
<template>
  <section>
    <section>
      <sub-heading>キャラクターリスト関連</sub-heading>
      <section class="form">
        <div class="form-title">キャラクターの短縮名（{{ nickname.length }} / {{ nicknameMaxLength }}文字）</div>
        <input class="form-input" type="text" v-model="nickname" placeholder="短縮名">
      </section>
      <section class="form">
        <div class="form-title">タグ</div>
        <div class="form-description">
          スペースで区切ることで複数指定できます。
        </div>
        <input class="form-input-long" type="text" v-model="joinedTags" placeholder="タグ">
      </section>
      <section class="form">
        <div class="form-title">サマリー</div>
        <div class="form-description">
          キャラクターリストで表示される短い文章です。
        </div>
        <input class="form-input-long" type="text" v-model="summary" placeholder="サマリー">
      </section>
      <section class="form">
        <div class="form-title">メインアイコンURL（横60px 縦60px）</div>
        <div class="form-description">
          キャラクターリストで表示されるアイコンです。
        </div>
        <input class="form-input-long" type="text" v-model="mainicon" placeholder="メインアイコンURL">
      </section>
      <character-list :characters="[{
        nickname: nickname,
        mainicon: mainicon,
        summary:  summary,
        eno:      eno,
        tags:     joinedTags ? joinedTags.split(/\s+/) : [],
```

```
      }]"/>
    </section>
    <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
    <div class="button-wrapper">
      <button class="button" @click="update">更新</button>
    </div>
    <section>
      <sub-heading>プロフィール</sub-heading>
      <section class="form">
        <div class="form-title">キャラクターのフルネーム（{{ name.length }} / {{ nameMaxLength }}文字）</div>
        <input class="form-input" type="text" v-model="name" placeholder="フルネーム">
      </section>
      <section class="form">
        <div class="form-title">プロフィール画像（横400px 縦600px）</div>
        <div class="form-description">
          改行することで複数指定できます。複数指定した場合ランダムで表示されます。
        </div>
        <textarea class="form-textarea" v-model="joinedProfileImages" placeholder="プロフィール画像"></textarea>
      </section>
      <section class="form">
        <div class="form-title">プロフィール文</div>
        <div class="form-description">
          プロフィール文は指定のタグで囲むことで装飾することができます。<br>
          詳しくはルールブックを確認してください。
        </div>
        <textarea class="form-textarea" v-model="profile" placeholder="プロフィール文"></textarea>
      </section>
    </section>
    <section>
      <section class="form">
        <div class="form-title">アイコン</div>
        <div class="form-description">
          アイコンは最大{{ iconsMaxLength }}件まで登録できます。
        </div>
        <table class="icon-editor">
          <thead>
            <th>プレビュー</th>
            <th>アイコン名</th>
            <th>URL</th>
          </thead>
          <tbody>
            <tr v-for="i in iconsMaxLength" :key="i" class="icon">
              <td class="icon-preview">
                <character-icon :src="icons[i - 1].url"/>
              </td>
              <td class="icon-name">
                <input type="text" v-model="icons[i - 1].name">
              </td>
              <td class="icon-url">
                <input type="text" v-model="icons[i - 1].url">
              </td>
            </tr>
          </tbody>
        </table>
        <div class="icon-add-button-wrapper">
          <button v-if="icons.length < iconsMaxLength" class="icon-add-button" @click="addIcon">アイコン追加
</button>
```

```
        </div>
      </section>
    </section>
    <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
    <div class="button-wrapper">
      <button class="button" @click="update">更新</button>
    </div>
    <character-profile
      mode="preview"
      :eno="eno"
      :name="name"
      :tags="joinedTags ? joinedTags.split(/\s+/) : []"
      :profile="profile | styling"
      :profileImage="joinedProfileImages ? joinedProfileImages.split(/\n/)[Math.floor(Math.random() * joinedPr
ofileImages.split(/\n/).length)] : null"
      :icons="icons"
      :status="status" />
    <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
    <div class="button-wrapper">
      <button class="button" @click="update">更新</button>
    </div>
  </section>
</template>

<script>
import SubHeading       from '~/components/SubHeading.vue'
import CharacterList    from '~/components/CharacterList.vue'
import CharacterIcon    from '~/components/CharacterIcon.vue'
import CharacterProfile from '~/components/CharacterProfile.vue'
import MessageBanner    from '~/components/MessageBanner.vue'

export default {
  components: {
    SubHeading,
    CharacterList,
    CharacterIcon,
    CharacterProfile,
    MessageBanner
  },
  middleware: 'authenticated',
  head() {
    return {
      title: 'キャラクター設定'
    };
  },
  data() {
    return {
      nameMaxLength:     process.env.nameMaxLength,
      nicknameMaxLength: process.env.nicknameMaxLength,
      iconsMaxLength:    process.env.iconsMaxLength,

      errorMessage:    '',
      waitingResponse: false
    }
  },
  asyncData: async function(context) {
    const response = await context.$axios.get('/api/characters/main/profile');
```

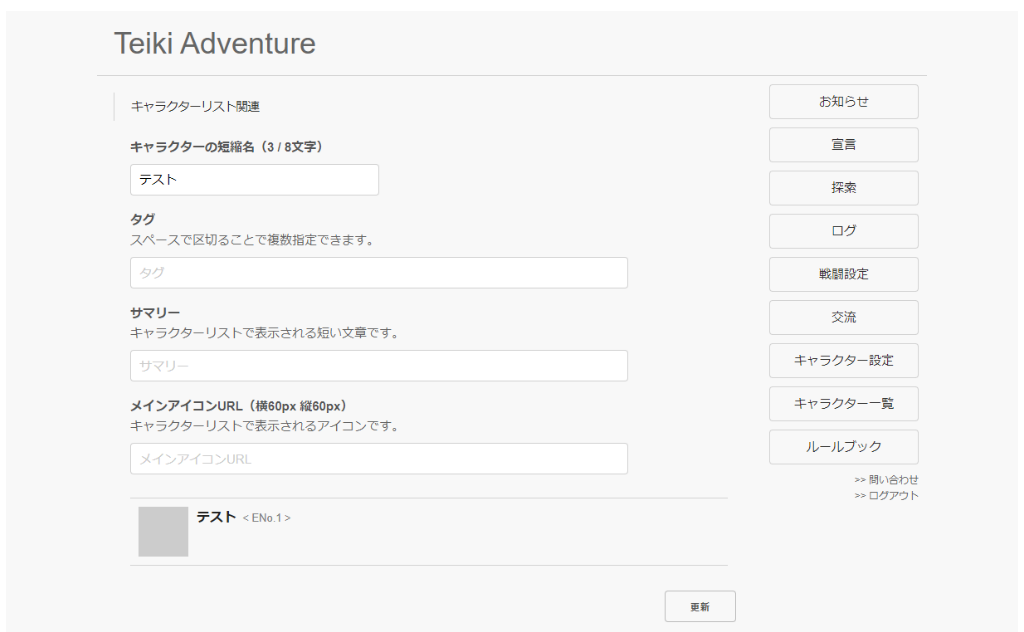
```
    // 実際のアイコンの量に関わらずアイコンの入力欄を作るため
    // アイコン配列が最大長に満たない場合延長
    const icons = [];
    for (let i = 0; i < process.env.iconsMaxLength; i++) {
      if (response.data.icons[i]) {
        icons.push(response.data.icons[i]);
      } else {
        icons.push({
          name: '',
          url:  ''
        });
      }
    }

    return {
      eno:        response.data.eno,
      name:       response.data.name,
      nickname:   response.data.nickname,
      summary:    response.data.summary,
      profile:    response.data.profile,
      mainicon:   response.data.mainicon,
      icons:      icons,
      status:     response.data.status,
      joinedTags: response.data.tags.join(' '),
      joinedProfileImages: response.data.profileImages.join('\n')
    };
  },
  filters: {
    styling: function(profile) {
      if (!profile) {
        return ''; // 変換するものがなければ空文字列を返す
      }

      return (
        profile // 装飾
          .replace(/<s>(.+)<\/s>/sig , '<span class="small">$1</span>')
          .replace(/<l>(.+)<\/l>/sig , '<span class="large">$1</span>')
          .replace(/<b>(.+)<\/b>/sig , '<span class="bold">$1</span>')
          .replace(/\n/g, '<br>')
      );
    }
  },
  methods: {
    update: async function() {
      if (this.waitingResponse) {
        // 接続待ち状態であればアラートを表示しreturn、以降の処理を行わない
        return alert('しばらくお待ち下さい');
      }

      // 入力内容の検証を行う、問題があればエラーメッセージを表示、returnして処理を中断
      if (!this.name) {
        return this.errorMessage = 'フルネームが入力されていません';
      }
      if (!this.nickname) {
        return this.errorMessage = '短縮名が入力されていません'
      }
```

```
      if (this.nameMaxLength < this.name.length) {
        return this.errorMessage = 'フルネームが長すぎます';
      }
      if (this.nicknameMaxLength < this.nickname.length) {
        return this.errorMessage = '短縮名が長すぎます';
      }

      // 問題がなければ接続に入る、接続待ち状態をON(true)に
      this.waitingResponse = true;

      try {
        // 間違えてプロフィール画像のところに空行を作ってしまう可能性が高いので
        // 空行は登録されないようにする
        const unfilteredProfileImages = this.joinedProfileImages.split(/\n/);
        const profileImages = [];
        for (let i = 0; i < unfilteredProfileImages.length; i++) {
          if (unfilteredProfileImages[i]) {
            profileImages.push(unfilteredProfileImages[i]);
          }
        }

        // 更新を行う
        await this.$axios.put('/api/characters/main/profile', {
          name:          this.name,
          nickname:      this.nickname,
          summary:       this.summary,
          profile:       this.profile,
          mainicon:      this.mainicon,
          icons:         this.icons,
          tags:          this.joinedTags ? this.joinedTags.split(/\s+/) : [],
          profileImages: profileImages
        });

        // 更新したら/profile/mainへリダイレクト
        this.$router.push('/profile/main');
      } catch (e) {
        // エラーが発生した場合エラーメッセージを表示して接続待ち状態をOFF(false)に
        this.errorMessage = '登録中にエラーが発生しました';
        this.waitingResponse = false;
      }
    }
  }
}
</script>

<style lang="scss" scoped>
.icon-editor {
  box-sizing: border-box;
  width: 100%;
  margin: 0;

  tr {
    width: 100%;
  }

  th, td {
    text-align: center;
```

```
    box-sizing: border-box;
    padding: 8px;
  }

  th:nth-child(1), td:nth-child(1) {
    width: 100px;
  }

  td:nth-child(1) {
    display: flex;
    justify-content: center;
  }

  th:nth-child(2), td:nth-child(2) {
    width: 100px;
  }

  th:nth-child(3), td:nth-child(3) {
    width: auto;
  }

  input {
    margin: 0;
    width: 100%;
  }
}
</style>
```

このページではタグやプロフィール画像を空白や改行で区切って複数指定します。これを編集するため、編集するときはタグやプロフィール画像をそれぞれの文字で連結したものを値として使用しており、送信や表示をするときにそれぞれの文字で分割しています。このとき何も入力されていなければ空文字列の要素が生まれてしまうので連結された文字列が実際に入力されているかをチェックしたり空の要素を送信しないようにしたりしています。また、プレビューのためにプロフィール文を装飾するフィルターをこちらにも実装しています。

さて、実際にこのページを表示すると以下のようになります(長いので一部のみ)。「http://dev.siroisakana.com/ta/profile/main/edit」にアクセスしてみてください。



それでは実際にいろいろと更新してみましょう。プロフィール文も更新してみます。ここでは「<s>小さいプロフィール文</s>」と入力してみます。更新するとキャラクターページに飛びます。するとプロフィール文が以下のようになっているはずです。

> プロフィール
>
> 小さいプロフィール文

変換後のHTMLがそのまま表示されてしまっています。Vue.jsでは基本的に表示する値などはHTMLタグなどとしては解釈されず「そのまま」で出力されます。今回の例ではそれでは困るのできちんとHTMLタグとして解釈されるようにしましょう。受け取った値をHTMLタグなどが解釈される形で出力するためにはv-htmlを使います。v-htmlにHTMLとして解釈したい値を渡すことでHTMLタグとして解釈される形式でそのまま出力してくれます。

それでは改変していきましょう。「frontend/components/CharacterProfile.vue」を開きます。プロフィールを表示している部分を以下のように書き換えて保存してください。

frontend/components/CharacterProfile.vue

```
（省略）

  <section>
    <sub-heading>プロフィール</sub-heading>
    <div class="profile" v-html="profile"></div>
  </section>

（省略）
```

すると表示が以下のようになり、しっかりと装飾システムが働いていることが分かります。

プロフィール

小さいプロフィール文

4.7.4　XSS

　さて今作った装飾システムですが実は脆弱性があります。先程HTMLをそのままの形式で解釈できるようにしました。つまりプロフィール文にHTMLコードを書くと**それがどんなものであってもHTMLとして解釈されて実行されてしまいます。**オリジナルの装飾を作ったりする程度であれば構わないのですが、例えば「<script>」タグを使って任意のコードを実行してログイン情報を盗み出したりといったことができてしまいます。このような攻撃のことを**XSS**（Cross Site Scripting）と呼びます。それに対して今のような脆弱性のことを**XSS脆弱性**と呼んだりします。

　XSSに対応するためには指定の装飾以外に**サニタイズ**（エスケープとも呼ばれる）という処理を行う必要があります。HTMLで特別な意味を持つ文字（**制御文字**）である「<」「>」「&」「'」「"」を表示が同じになるものの制御に関わらない文字「<」「>」「&」「'」「"」にそれぞれ変換するのです。

　Vue.jsではデフォルトでサニタイズが行われおり、そのために先程プロフィール文がHTMLタグそのままの見た目で出力されていた、というわけです。

　さて、装飾システムを作る場合とりあえずひとまず全てサニタイズを行ってから指定のタグをHTMLタグに変換します。これで指定のタグ以外はサニタイズされ指定のタグだけがHTMLとして解釈されるコードにすることができます。「<s> 〜 </s>」などはサニタイズ化されると「<s> 〜 </s>」になるので、それを前提にコードを組んでいきましょう。「backend/server/styling.js」を以下のように書き換え保存します。

backend/server/styling.js

```
const sanitize = (str) => {
  // 受け取った文字列をサニタイズする
  if (!str) { // 値がない場合そのまま空文字列を返す
    return '';
  }

  return (
    str
      .replace(/&/g, '&amp;')
      .replace(/</g, '&lt;')
      .replace(/>/g, '&gt;')
      .replace(/"/g, '&quot;')
      .replace(/'/g, '&#39;')
  );
};

const profile = (str) => {
  // 受け取った文字列をサニタイズして指定のタグで装飾し、改行させる
  // <s> 〜 </s> 文字を小さくする
  // <l> 〜 </l> 文字を大きくする
  // <b> 〜 </b> 文字を太くする
  return (
    sanitize(str)
      .replace(/&lt;s&gt;(.+)&lt;\/s&gt;/sig , '<span class="small">$1</span>')
      .replace(/&lt;l&gt;(.+)&lt;\/l&gt;/sig , '<span class="large">$1</span>')
```

```
        .replace(/&lt;b&gt;(.+)&lt;\/b&gt;/sig , '<span class="bold">$1</span>')
        .replace(/\n/g, '<br>')
    );
};

module.exports = {
  sanitize: sanitize,
  profile:  profile
};
```

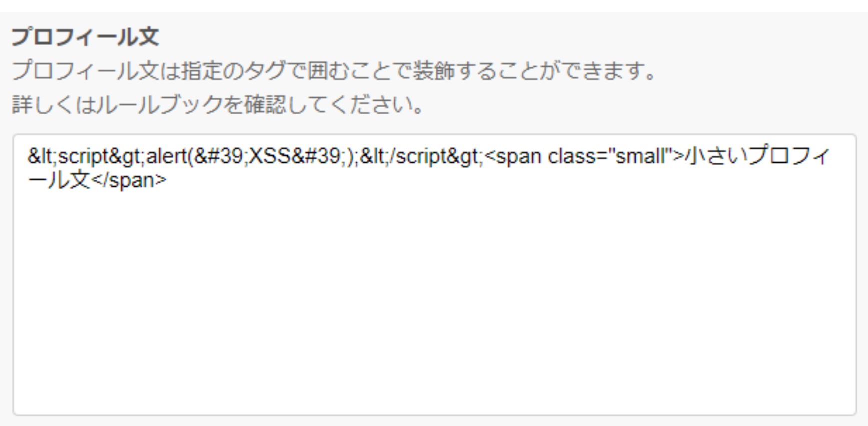
profile関数ではひとまずsanitize(str)を呼び出すことで受け取った文字列をサニタイズした後、正規表現によって「<s>」「</s>」などで囲まれた内容を「」「」などで囲み直しています。これにより指定のタグだけがHTMLタグとして変換されて残ります。

それではテストしてみましょう。キャラクター設定ページを開き、プロフィール文に「<script>alert('XSS');</script><s>小さいプロフィール文</s>」と入力して更新してみてください。以下の例のように<script>部分はサニタイズされてコードそのままの形式で表示されるようになっていますが、<s>で囲まれた部分だけはスタイルがしっかりと適用されていることが確認できます。

プロフィール

<script>alert('XSS');</script>小さいプロフィール文

なお、サニタイズ処理によって入力したときと保存してある形式が変わってしまっているので、次にこのプロフィール文を編集するときにサニタイズ処理された結果がエディターに表示されるようになってしまいます。この例では以下のようになります。



これは不便なのでエディターではサニタイズと逆の変換を行って入力したときの内容が表示されるようにしましょう。またエディターのフィルタリングシステムもXSS対策表示されたものが出るように更新します。「frontend/pages/profile/main/edit.vue」を開き、asyncDataとfiltersの部分を以下のように書き換えてください。

```
frontend/pages/profile/main/edit.vue
（省略）

asyncData: async function(context) {
  const response = await context.$axios.get('/api/characters/main/profile');

  // 実際のアイコンの量に関わらずアイコンの入力欄を作るため
  // アイコン配列が最大長に満たない場合延長
  const icons = [];
  for (let i = 0; i < process.env.iconsMaxLength; i++) {
    if (response.data.icons[i]) {
      icons.push(response.data.icons[i]);
    } else {
      icons.push({
        name: '',
        url:  ''
      });
    }
  }

  // サニタイズされたプロフィール文を逆変換する
  let profile = '';
  if (response.data.profile) {
      profile = response.data.profile
      // サニタイズを逆変換
        .replace(/&amp;/g , '&')
        .replace(/&lt;/g  , '<')
        .replace(/&gt;/g  , '>')
        .replace(/&quot;/g, '"')
        .replace(/&#39;/g , "'")
      // 変換された後のタグを元のタグへ
        .replace(/<span class="small">(.+)<\/span>/g, '<s>$1</s>')
        .replace(/<span class="large">(.+)<\/span>/g, '<l>$1</l>')
        .replace(/<span class="bold">(.+)<\/span>/g , '<b>$1</b>')
      // <br>を改行に
        .replace(/<br>/g, '\n')
  }

  return {
    eno:         response.data.eno,
    name:        response.data.name,
    nickname:    response.data.nickname,
    summary:     response.data.summary,
    profile:     profile,
    mainicon:    response.data.mainicon,
    icons:       icons,
    status:      response.data.status,
    joinedTags: response.data.tags.join(' '),
    joinedProfileImages: response.data.profileImages.join('\n')
  };
},
filters: {
  styling: function(profile) {
    if (!profile) {
      return ''; // 変換するものがなければ空文字列を返す
    }
```

```
      return (
        profile
        // サニタイズ
          .replace(/&/g, '&amp;')
          .replace(/</g, '&lt;')
          .replace(/>/g, '&gt;')
          .replace(/"/g, '&quot;')
          .replace(/'/g, '&#39;')
        // タグ置換、サニタイズしてあるのでその前提で置換する
          .replace(/&lt;s&gt;(.+)&lt;\/s&gt;/sig , '<span class="small">$1</span>')
          .replace(/&lt;l&gt;(.+)&lt;\/l&gt;/sig , '<span class="large">$1</span>')
          .replace(/&lt;b&gt;(.+)&lt;\/b&gt;/sig , '<span class="bold">$1</span>')
        // 改行置換
          .replace(/\n/g, '<br>')
      );
    }
  },

 （省略）
```

4.7.5　CSRF

XSS脆弱性以外にも実はまだ脆弱性があります。XSS以外にもう1つWebアプリへ想定される攻撃を学んでおきましょう。それは**CSRF**(Cross-Site Request Forgeries)と呼ばれるものです。

『ログインの仕組み』のところでセッションに使われるクッキーはWebサイト側に自動的に送信されると解説しました。これは**他のWebサイトからのリクエストであっても同様**です。つまりどういうことかというと、例えば「/characterdelete」のようなAPIがあるとして、悪意のあるユーザーが用意した全く別のサイトが勝手に「/characterdelete」APIへアクセスするようなコードを組んでしまうとブラウザ側はアクセス元が全く別のサイトであるにも関わらず「/characterdeleteがあるサイトのクッキーと一緒に」「/characterdelete」からのアクセスを行ってしまいます。認証方法がクッキーだけだとサーバー側は「これはユーザー側が行っているものだ」と認識してしまい悪意あるユーザーのページを表示しただけで勝手にキャラクターが削除されてしまう、という事態が起こりえます。このような攻撃がCSRFです。そのため何らかの操作を行うAPIにはCSRF対策を行うようにしましょう。

さて、それではCSRF対策はどのようにしたらよいのでしょうか。これにはブラウザの**同一生成元ポリシー**(Same Origin Policy)という特性を利用します。同一生成元ポリシーとはざっくりいってしまうと「同じFQDN、同じプロトコル(**同一オリジン**)からでないとリソースへのアクセスを許可しない」というもので、言い換えると「外部のサイトからはページやAPIなどのリソースにGETリクエストした結果を受け取れない」というものになります。(なお画像や動画・音声・JavaScriptなどはその例外になります。定期・APゲームで他のサイトのURLを設定してもアイコンやプロフィール画像が読み込めたり、CDNからjQueryが使えたりするのはそのおかげです。)

つまり「サーバー側は他者から推測されないユーザーごとの固有の値(**CSRFトークン**)をGETリクエストのAPIから渡し」「クライアント側はCSRFトークンを返送して同じドメインからのアクセスであることを証明する」ことでCSRF対策が実現できます。

それを踏まえてCSRFに関する今までのコードを確認してみましょう。まず「backend/server/model.js」のキャラクタースキーマのcsrfのところでランダムな値をCSRFトークンとして生成しています。(crypto.randomBytesは指定の長さの暗号学的に安全な乱数を生成する関数です。)

```
csrf: {type: String, default: () => crypto.randomBytes(config.csrfTokenLength).toString('hex')}
```

　また、「backend/server/api.js」の「/characters/main/auth」からCSRFトークンが返るようになっています。このAPIはGETリクエストから呼び出されるので、もし悪意あるユーザーが他のドメインから呼び出そうとしても同一生成元ポリシーにより内容を取得することができません。これによって「正しいCSRFトークンを持っている」＝「ユーザー自身がこのサイトからアクセスしている」という証明になります。

```
router.get('/characters/main/auth', checkAuthentication, (req, res) => {
  return res.status(200).send({
    eno: req.user.eno,
    csrf: req.user.csrf
  });
});
```

　Nuxt.js側のログイン処理によって「/characters/main/auth」からクライアント側にCSRFトークンが渡りVuexに格納されるようになっているので、あとはデータ送信時などにそのCSRFトークンを同時に送信するようにしサーバー側の持っているCSRFトークンと合致すれば認証成功ということになります。
　なお、これが働くのは同一ドメイン下にXSS脆弱性が存在しないことが前提になります。XSS脆弱性により任意のコードが実行可能だと同一ドメイン下で「/characters/main/auth」やVuexに不正にアクセスすることができてしまい、同一生成元ポリシーも働かず任意の操作が行われてしまいます。XSS対策もしっかりするようにしましょう。
　それを踏まえてキャラクター設定APIを改良していきましょう。これからいろいろAPIを作る上でいちいちCSRFトークンを照合する処理を書くのは面倒なので、まずはCSRFトークンを照合するミドルウェア関数を作成します。「backend/server/api.js」のcheckAuthencationを宣言している箇所の次の場所に以下のようにcheckCsrfを作りましょう。追記された箇所は斜体で表示してあります。

backend/server/api.js

```
  （省略）

const checkAuthentication = (req, res, next) => {
  // ログインしているか確認するミドルウェア関数
  if (req.user) { // ログインしていれば（req.userがあれば）次へ
    next();
  } else { // ログインしていなければ401を返して処理を中断する
    res.status(401).send();
  }
};

const checkCsrf = (req, res, next) => {
  // CSRFトークンのチェック
  // req.userへのアクセスが走るのでcheckAuthenticationの後に使うこと
  if (req.user.csrf == req.body.csrf) { // 合っていれば次へ
    next();
  } else { // CSRFトークンが合わなければ401を返して処理を中断する
```

```
    res.sendStatus(401);
  }
};

  （省略）
```

　このようにすることで送信されたデータのCSRFの項目とreq.userに格納されたCSRFトークンを照合し合致した場合のみ処理を進めることができます。req.userへのアクセスがあるので使用するときはcheckAuthencation, checkCsrfの順番で呼び出すようにしてください。
　このミドルウェア関数をAPIに適用しましょう。「/character/main/profile」APIのPUT側の宣言部分を以下のように書き換えてください。（間違えてGET側APIを変えないように注意してください。GET側APIにアクセスできなくなります。）

backend/server/api.js

```
  （省略）

router.put('/characters/main/profile', checkAuthentication, checkCsrf, async (req, res) => {

  （省略）
```

　さて、API側の対策が組み終わったらクライアント側で受け取ったCSRFを送信するようにします。「frontend/pages/profile/main/edit.vue」のmethodsのupdateのデータ送信部分を以下のように書き換えてください。Vuexに格納されたCSRFトークンを更新するデータの内容と一緒に送信するようにしています。これでCSRF対策も完了です。

```
        // 更新を行う
        await this.$axios.put('/api/characters/main/profile', {
          csrf:         this.$store.getters['auth/loginCharacter'].csrf,
          name:         this.name,
          nickname:     this.nickname,
          summary:      this.summary,
          profile:      this.profile,
          mainicon:     this.mainicon,
          icons:        this.icons,
          tags:         this.joinedTags ? this.joinedTags.split(/\s+/) : [],
          profileImages: profileImages
        });
```

4.8 お気に入り / ブロック / ミュート

4.8.1 APIの作成

キャラクターページを作ったらお気に入りやブロック・ミュートができるようにしましょう。それぞれの機能部分は別に実装するとして、とりあえず指定のキャラクターをお気に入り・ブロック・ミュートに入れられるようにします。それではそのためのAPIを作っていきましょう。それぞれ登録するAPIと解除するAPIが必要なので合計6個のAPIを作ることになります。「backend/server/api.js」を開き、他のAPIと同様に以下のAPIを追加して保存してください。

backend/server/api.js

```
router.post('/characters/:eno(\\d+\)/faved', checkAuthentication, checkCsrf, async (req, res) => {
  if (Number(req.params.eno) == req.user.eno) {
    return res.status(400).send(); // 自キャラをお気に入りしようとしたときは400を返し中断
  }

  try {
    const user = await Character.findById(req.user._id, {
      block: 1,
      blocked: 1
    });
    const target = await Character.findOne({eno: Number(req.params.eno)}, {_id: 1});

    if (user.block.concat(user.blocked).includes(target._id)) {
      // ブロックしているかされている場合は403を返し中断
      return res.status(403).send();
    }

    await user.update({$addToSet: {fav: target._id}}); // ユーザーのお気に入りに対象を追加

    return res.status(200).send();
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});

router.post('/characters/:eno(\\d+\)/blocked', checkAuthentication, checkCsrf, async (req, res) => {
  if (Number(req.params.eno) == req.user.eno) {
    return res.status(400).send(); // 自キャラをブロックしようとしたときは400を返し中断
  }

  try {
    const user = await Character.findById(req.user._id, {_id: 1});
    const target = await Character.findOne({eno: Number(req.params.eno)}, {_id: 1});

    await user.update({  // ユーザーのブロックに対象を追加しお気に入りから削除
      $addToSet: {block: target._id},
      $pull:     {fav:   target._id}
    });
    await target.update({ // 対象の被ブロックにユーザーを追加しお気に入りから削除
      $addToSet: {blocked: user._id},
      $pull:     {fav:     user._id}
    });
```

```
    return res.status(200).send();
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});

router.post('/characters/:eno(\\d+\)/muted', checkAuthentication, checkCsrf, async (req, res) => {
  if (Number(req.params.eno) == req.user.eno) {
    return res.status(400).send(); // 自キャラをミュートしようとしたときは400を返し中断
  }

  try {
    const user = await Character.findById(req.user._id, {_id: 1});
    const target = await Character.findOne({eno: Number(req.params.eno)}, {_id: 1});

    await user.update({$addToSet: {mute: target._id}}); // ユーザーのミュートに対象を追加

    return res.status(200).send();
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});

router.delete('/characters/:eno(\\d+\)/faved', checkAuthentication, checkCsrf, async (req, res) => {
  if (Number(req.params.eno) == req.user.eno) {
    return res.status(400).send(); // 自キャラをお気に入り解除しようとしたときは400を返し中断
  }

  try {
    const user = await Character.findById(req.user._id, {_id: 1});
    const target = await Character.findOne({eno: Number(req.params.eno)}, {_id: 1});

    await user.update({$pull: {fav: target._id}}); // ユーザーのお気に入りから対象を削除

    return res.status(200).send();
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});

router.delete('/characters/:eno(\\d+\)/blocked', checkAuthentication, checkCsrf, async (req, res) => {
  if (Number(req.params.eno) == req.user.eno) {
    return res.status(400).send(); // 自キャラをブロック解除しようとしたときは400を返し中断
  }

  try {
    const user = await Character.findById(req.user._id, {_id: 1});
    const target = await Character.findOne({eno: Number(req.params.eno)}, {_id: 1});

    await user.update({$pull: {block: target._id}}); // ユーザーのブロックから対象を削除
    await target.update({$pull: {blocked: user._id}}); // 対象の被ブロックからユーザーを削除

    return res.status(200).send();
```

```
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});

router.delete('/characters/:eno(\\d+\)/muted', checkAuthentication, checkCsrf, async (req, res) => {
  if (Number(req.params.eno) == req.user.eno) {
    return res.status(400).send(); // 自キャラをミュート解除しようとしたときは400を返し中断
  }

  try {
    const user = await Character.findById(req.user._id, {_id: 1});
    const target = await Character.findOne({eno: Number(req.params.eno)}, {_id: 1});

    await user.update({$pull: {mute: target._id}}); // ユーザーのミュートから対象を削除

    return res.status(200).send();
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

やっていることはどのAPIもあまり変わりません。ユーザーと対象のキャラクターを検索し、内容に応じた配列にキャラクターを追加したり削除したりしています。ブロックではお気に入りしている/されているときにそれを強制的に解除している他、被ブロックの情報を対象キャラクターに追加しています。ブロック解除でも同様に対象の被ブロック情報を削除しています。それぞれのAPIは書き並べると以下のようになります。

URL	メソッド	実行される処理
/api/characters/(ENo)/faved	POST	ユーザーのお気に入りに対象を追加
/api/characters/(ENo)/blocked	POST	ユーザーのブロックに対象を追加 対象の被ブロックにユーザーを追加 対象とユーザーのお気に入りからお互いを削除
/api/characters/(ENo)/muted	POST	ユーザーのミュートに対象を追加
/api/characters/(ENo)/faved	DELETE	ユーザーのお気に入りから対象を削除
/api/characters/(ENo)/blocked	DELETE	ユーザーのブロックから対象を削除 対象の被ブロックからユーザーを削除
/api/characters/(ENo)/muted	DELETE	ユーザーのミュートから対象を削除

また、ログインしている状態でキャラクターページのAPIにアクセスしたときに対象をお気に入り・ブロック・ミュートしているかという情報も返すようにしましょう。「/characters/:key(\\d+|main\)」のGETのAPIを以下のように書き換えて保存してください。

```
backend/server/api.js
router.get('/characters/:key(\\d+|main\)', async (req, res) => {
  if (/main/.test(req.params.key) && !req.user) {
    return res.status(401).send(); // 自キャラ表示なのにログインしていなければ401
  }

  const query = /main/.test(req.params.key) ? {_id: req.user._id} : {eno: Number(req.params.key)};
  // 自キャラ表示であればセッションユーザーのキャラ、そうでなければ指定のENoで検索

  query.deleted = false;
  // 削除フラグが立っていないことを検索条件に追加

  try {
    const character = await Character.findOne(query, {
      eno:           1,
      name:          1,
      tags:          1,
      profile:       1,
      icons:         1,
      status:        1,
      profileImages: 1
    });

    if (!character) {
      return res.status(404).send(); // キャラクターの検索結果が存在しなければ404
    } else if (req.user && !(/main/.test(req.params.key))) {
      // キャラクターが見つかり、かつログインしていて自キャラ表示ではない場合
      // 接続者のお気に入り、ブロック、ミュート情報を取得
      const user = await Character.findById(req.user._id, {
        _id:   0,
        fav:   1,
        block: 1,
        mute:  1
      });

      // キャラクター情報とお気に入り、ブロック、ミュートしているかという情報を返す
      return res.status(200).send({
        eno:           character.eno,
        name:          character.name,
        tags:          character.tags,
        profile:       character.profile,
        icons:         character.icons,
        status:        character.status,
        profileImage:  character.profileImages.length ? character.profileImages[Math.floor(Math.ra
ndom() * character.profileImages.length)] : null,
        isFaved:       user.fav.includes(character._id),
        isBlocked:     user.block.includes(character._id),
        isMuted:       user.mute.includes(character._id)
      });
    } else {
      // キャラクターは見つかったがログインはしていないもしくは自キャラ表示のとき
      // そのまま情報を返す
      return res.status(200).send({
        eno:           character.eno,
        name:          character.name,
        tags:          character.tags,
        profile:       character.profile,
```

```
        icons:         character.icons,
        status:        character.status,
        profileImage:  character.profileImages.length ? character.profileImages[Math.floor(Math.ra
ndom() * character.profileImages.length)] : null
      });
    }
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

4.8.2 キャラクターページの変更

それではこれらのAPIからそれぞれの操作を行えるようにしましょう。「frontend/components/CharacterProfile.vue」を開き以下のように書き換えて保存してください。変更のあった箇所は斜体で表示してあります。

```
frontend/components/CharacterProfile.vue
<template>
  <section>
    <h2 class="name">ENo.{{ eno }} {{ name }}</h2>
    <section
      class="relation"
      v-if="
        mode == 'profile' &&
        $store.getters['auth/isAuthenticated'] &&
        eno != $store.getters['auth/loginCharacter'].eno">
      <div
        class="relation-button"
        v-if="!isFaved && !isBlocked"
        @click="relation('fav')">
          お気に入りする
      </div>
      <div
        class="relation-button done"
        v-if="isFaved && !isBlocked"
        @click="relation('unfav')">
          お気に入り中
      </div>
      <div
        class="relation-button"
        v-if="!isBlocked"
        @click="relation('block')">
          ブロックする
      </div>
      <div
        class="relation-button done"
        v-if="isBlocked"
        @click="relation('unblock')">
          ブロック中
      </div>
      <div
        class="relation-button"
```

```
      v-if="!isMuted"
      @click="relation('mute')">
        ミュートする
    </div>
    <div
      class="relation-button done"
      v-if="isMuted"
      @click="relation('unmute')">
        ミュート中
    </div>
  </section>
  <section class="main">
    <section class="image-wrapper">
      <img v-if="profileImage" class="image" :src="profileImage">
    </section>
    <section class="summary">
      <div v-if="mode == 'home'" class="ap-wrapper">
        <div class="ap">AP</div>
        <div class="ap-value">{{ ap }}</div>
      </div>
      <table class="statuses">
        <tbody>
          <tr>
            <td class="status-name">ATK</td>
            <td class="status-value">{{ status.atk }}</td>
            <td class="status-name">DEX</td>
            <td class="status-value">{{ status.dex }}</td>
            <td class="status-name">MND</td>
            <td class="status-value">{{ status.mnd }}</td>
          </tr>
          <tr>
            <td class="status-name">AGI</td>
            <td class="status-value">{{ status.agi }}</td>
            <td class="status-name">DEF</td>
            <td class="status-value">{{ status.def }}</td>
          </tr>
        </tbody>
      </table>
      <div class="skills">
        <div class="skill">
          <div class="skill-prop">
            <div class="skill-name">フレア</div>
            <div class="skill-cond">50SP / 9行動毎</div>
          </div>
          <div class="skill-effect">敵：攻撃＋SPが30%以上なら敵全：攻撃</div>
        </div>
        <div class="skill">
          <div class="skill-prop">
            <div class="skill-name">エリアヒール</div>
            <div class="skill-cond">50SP / 味方重傷</div>
          </div>
          <div class="skill-effect">味全：HP回復</div>
        </div>
        <div class="skill">
          <div class="skill-prop">
            <div class="skill-name">反撃</div>
            <div class="skill-cond">攻撃回避時</div>
```

```
          </div>
          <div class="skill-effect">敵：攻撃</div>
        </div>
        <div class="skill">
          <div class="skill-prop">
            <div class="skill-name">シフトアップ</div>
            <div class="skill-cond">戦闘開始時</div>
          </div>
          <div class="skill-effect">自：AGI増</div>
        </div>
        <div class="skill">
          <div class="skill-prop">
            <div class="skill-name">自己修復</div>
            <div class="skill-cond">被攻撃時</div>
          </div>
          <div class="skill-effect">自：HP回復</div>
        </div>
      </div>
    </section>
  </section>
  <section v-if="tags.length">
    <span class="tag" v-for="(tag, index) in tags" :key="index">{{ tag }}</span>
  </section>
  <section>
    <sub-heading>プロフィール</sub-heading>
    <div class="profile" v-html="profile"></div>
  </section>
  <section>
    <sub-heading>アイコン</sub-heading>
    <div class="character-icons-wrapper">
      <div class="character-icon-wrapper" v-for="i in iconsMaxLength" :key="i">
        <character-icon :src="icons[i - 1] ? icons[i - 1].url : ''"/>
      </div>
    </div>
  </section>
 </section>
</template>

<script>
import SubHeading    from '~/components/SubHeading.vue'
import CharacterIcon from '~/components/CharacterIcon.vue'

export default {
  components: {
    SubHeading,
    CharacterIcon
  },
  props: {
    mode:         String, // 表示モード home = ホーム、profile = キャラクターページ、preview = プレビュー
    eno:          Number,
    name:         String,
    tags:         Array,
    profile:      String,
    profileImage: String,
    icons:        Array,
    ap:           Number,
    status:       Object,
```

```
    isFaved:      Boolean,
    isBlocked:    Boolean,
    isMuted:      Boolean
  },
  data() {
    return {
      iconsMaxLength: process.env.iconsMaxLength
    };
  },
  methods: {
    relation: function(action) {
      this.$emit('relation', action);
    }
  }
}
</script>

<style lang="scss" scoped>
$font: 'BIZ UDPGothic', 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ', Meiryo, 'ＭＳ Ｐゴシック
', sans-serif;

.name {
  display: block;
  background: #444;
  color: #EEE;
  margin: 10px 0;
  padding: 10px;
  font-size: 18px;
  font-family: $font;
  letter-spacing: 0;
}

.relation {
  display: flex;
  justify-content: flex-end;
  width: 100%;

  .relation-button {
    display: inline-flex;
    justify-content: center;
    padding: 4px 10px;
    border: 1px solid #666;
    border-radius: 4px;
    margin: 0 5px;
    text-decoration: none;
    font-weight: bold;
    color: #666;
    cursor: pointer;
  }

  .done {
    border: 1px solid #333;
    background: #666;
    color: #EEE;
  }
}
```

```
.main {
  display: flex;

  .image-wrapper {
    flex-grow: 0;
    flex-shrink: 0;
    width: 400px;
    height: 600px;

    .image {
      width: 100%;
      height: 100%;
    }
  }

  .summary {
    flex-grow: 1;
    padding: 10px;
    display: flex;
    align-items: center;
    justify-content: center;
    flex-direction: column;

    .ap-wrapper {
      width: 100%;
      display: flex;
      align-items: baseline;
      justify-content: center;
      font-family: $font;

      .ap {
        font-weight: bold;
        font-size: 24px;
        color: #AAA;
      }

      .ap-value {
        margin-left: 40px;
        font-weight: bold;
        font-size: 48px;
        color: #666;
      }
    }

    .statuses {
      box-sizing: border-box;
      margin: 0;
      padding: 12px 20px;
      width: 100%;
      border-top: 1px solid lightgray;
      border-bottom: 1px solid lightgray;
      font-family: $font;

      td {
        border: none;
        padding: 6px 12px;
      }
```

```
      .status-name {
        font-weight: bold;
        color: #555;
        margin-right: 20px;
      }
    }
  }
}

.skills {
  box-sizing: border-box;
  width: 100%;
  padding: 16px 24px 0 24px;

  .skill {
    margin-bottom: 12px;

    .skill-prop {
      display: flex;
      align-items: baseline;

      .skill-name {
        font-weight: bold;
        color: #555;
        font-size: 18px;
        margin-right: 10px;
      }

      .skill-cond {
        font-size: 14px;
        color: #888;
      }
    }

    .skill-effect {
      margin-left: 10px;
      color: #333;
    }

    &:last-child {
      margin: 0;
    }
  }
}

.tag {
  display: inline-flex;
  justify-content: center;
  min-width: 50px;
  padding: 4px;
  border: 1px solid #333;
  border-radius: 4px;
  margin-right: 10px;
  text-decoration: none;
  font-weight: bold;
  color: #333;
```

```
}

.profile {
  margin: 0 20px;
}

.character-icons-wrapper {
  display: flex;
  flex-wrap: wrap;
  justify-content: center;

  .character-icon-wrapper {
    margin: 5px;
  }
}
</style>
```

追加された部分を見ていくとまずpropsのところで「isFaved」「isBlocked」「isMuted」の値を受け取るようになっています。これは<template>のところで「プロフィール画面モード、閲覧者がログインしており、閲覧者のキャラクターと表示しているキャラクターが違う」場合にそれぞれの値の内容に応じてボタンが表示されるようになっています。なお、ブロックしているときはお気に入り関連のボタンは出ないようになっています。

そして、それぞれのボタンを押したときに直接APIにアクセスするのではなく、クリックしたときにメソッドrelationが呼び出されるようになっており、その中でカスタムイベントのrelationイベントがアクションの内容を示す値とともに呼び出されています。actionはクリックするボタンにより"fav", "unfav", "block", "unblock", "mute", "unmute"の値を取ります。

さてそれではこのrelationカスタムイベントを受け取ってページ側でAPIへアクセスする処理を行うようにしましょう。「frontend/pages/profile/_key.vue」を以下のように書き換えてください。変更があった箇所は斜体で表示してあります。

frontend/pages/profile/_key.vue

```
<template>
  <section>
    <character-profile
      mode="profile"
      :eno="eno"
      :name="name"
      :tags="tags"
      :profile="profile"
      :profileImage="profileImage"
      :icons="icons"
      :status="status"
      :isFaved="isFaved"
      :isBlocked="isBlocked"
      :isMuted="isMuted"
      @relation="relation"/>
  </section>
</template>

<script>
import CharacterProfile from '~/components/CharacterProfile.vue'
```

```
export default {
  components: {
    CharacterProfile
  },
  validate({params}) {
    return /^[1-9][0-9]*$/.test(params.key) || params.key == "main";
  },
  head() {
    return {
      title: `ENo.${this.eno} ${this.name}`
    };
  },
  data() {
    return {
      waitingResponse: false
    }
  },
  asyncData: async function(context) {
    const response = await context.$axios.get(`/api/characters/${context.params.key}`);
    return {
      eno:          response.data.eno,
      name:         response.data.name,
      tags:         response.data.tags,
      profile:      response.data.profile,
      profileImage: response.data.profileImage,
      icons:        response.data.icons,
      status:       response.data.status,
      isFaved:      response.data.isFaved,
      isBlocked:    response.data.isBlocked,
      isMuted:      response.data.isMuted
    }
  },
  methods: {
    relation: async function(action) {
      if (this.waitingResponse) {
        // 接続待ち状態であればアラートを表示し return、以降の処理を行わない
        return alert('しばらくお待ち下さい');
      }

      // 接続待ち状態を ON に
      this.waitingResponse = true;

      // 指定の内容に応じた API へ処理を渡す
      if (action == 'fav') {
        await this.$axios.post(`/api/characters/${this.$route.params.key}/faved`, {
          csrf: this.$store.getters['auth/loginCharacter'].csrf
        });
      }
      if (action == 'block') {
        await this.$axios.post(`/api/characters/${this.$route.params.key}/blocked`, {
          csrf: this.$store.getters['auth/loginCharacter'].csrf
        });
      }
      if (action == 'mute') {
        await this.$axios.post(`/api/characters/${this.$route.params.key}/muted`, {
          csrf: this.$store.getters['auth/loginCharacter'].csrf
```

```
        });
      }
      if (action == 'unfav') {
        await this.$axios.delete(`/api/characters/${this.$route.params.key}/faved`, {
          data: {
            csrf: this.$store.getters['auth/loginCharacter'].csrf
          }
        });
      }
      if (action == 'unblock') {
        await this.$axios.delete(`/api/characters/${this.$route.params.key}/blocked`, {
          data: {
            csrf: this.$store.getters['auth/loginCharacter'].csrf
          }
        });
      }
      if (action == 'unmute') {
        await this.$axios.delete(`/api/characters/${this.$route.params.key}/muted`, {
          data: {
            csrf: this.$store.getters['auth/loginCharacter'].csrf
          }
        });
      }

      // リクエストを行ったらお気に入り、ブロック、ミュート情報を更新
      const response = await this.$axios.get(`/api/characters/${this.$route.params.key}`);
      this.isFaved   = response.data.isFaved;
      this.isBlocked = response.data.isBlocked;
      this.isMuted   = response.data.isMuted;

      // 接続待ち状態をOFFに
      this.waitingResponse = false;
    }
  }
}
</script>
```

「isFaved」「isBlocked」「isMuted」をAPIから受け取ってCharacterProfileに表示するようにしている他、CharacterProfileからrelationイベントを受け取り、relationメソッドを呼び出すようにしています。relationメソッドでは渡されたactionの値に応じたAPIに処理を渡しています。ここで注意するべきなのはDELETE時の送信内容です。AxiosのPOSTやPUTでは送るデータの内容は単純に第2引数に設定するだけでよかったのですが、DELETE時は第2引数のdataプロパティの中に送るデータの内容を書くようにしなければなりません。アクセスが終わったらもう一度リクエストを行い、「isFaved」「isBlocked」「isMuted」の情報を更新しています。これによってページをリロードしなくても即座にそれらの関連情報が更新されるようになります。

実際にページにアクセスすると以下のようになります。1キャラクターしか登録していない場合は他のキャラクターの登録を行い、他キャラクターページから実際にそれぞれのボタンを押してみてください。変更内容が反映されるようになっているはずです。

Teiki Adventure

ENo.2 テスト2

お気に入り中 ブロックする ミュートする

4.9 トークルーム

4.9.1 トークルームのスキーマ設計

トークルームを作る前に要求されるトークルームの仕様について考えましょう。今回は次のようなトークルームを作ります。

トークルームの仕様

・全員が自由に会話できるパブリックトークルームとユーザーが立てることのできる個室トークルームが存在する。
・トークルーム内では自身および他のキャラクターの発言をリアルタイムに確認することができる。
・他の人の発言に対して返信をすることができる。
・返信には返信先キャラクターが記載されており、新たなキャラクターが返信による会話に参加していくことで返信先キャラクターは増えていく。
・トークルームの発言は以下のように絞り込み表示ができる。

全体	そのトークルームの発言全て
関連	自身の発言および自身への返信
お気に入り	自身およびお気に入りしているキャラクターの発言
自分	自身の発言
返信ツリー	ある返信に関して、その返信ツリーの全体を確認する

それでは以上の仕様を持ったトークルームを実現するアルゴリズムとデータ構造について考えてみましょう。まずは発言のデータをどのようにデータベースを保存するかについて考えます。最初に思いつくのはトークルームごとにドキュメントを用意し、その中に発言の配列を用意することです。しかし、これはうまく動作しません。MongoDBには1ドキュメントあたり16MBまでという制約があります。発言がどんどん増えていくとこの制約を越えるようになってしまい発言を書き込めなくなる恐れがあるためです。また、配列に入れ込んでしまうとMongoDBの豊富な検索機能も使いづらくなってしまいます。そういった観点から見てもトークルームドキュメントにログを保存してしまうのは好ましくありません。

そうなるとトークルームドキュメントとは別に発言のドキュメントを用意し、1つ1つの発言に対してドキュメントを保存していく形になるでしょう。これを発言ログと呼ぶことにします。ここで考えるべきは「部屋ごとにコレクションを用意しそこに発言ログを保存する」と「全ての部屋の発言ログを1つのコレクションにまとめる」のどちらの方式を採用するかです。これについてはどちらの構成もありえます。前者のような方式を採用しているプロジェクトもありますし後者のような方式を採用しているプロジェクトもあります。しかし前者のような方式はソースコードや管理が複雑になりがちなため今回は後者の方式を採用します。

さて、どのように発言ログを保存するか決まったところで実際にデータ構造を考えていきましょう。「発言者」「発言時のアイコン」「発言の内容」「返信先のキャラクター」「発言時刻」「削除フラグ」は当然必要になってくるでしょう。「返信先発言ログのID」もあったほうが好ましいかもしれません。また全ての発言ログを1つのコレクションにまとめるので「発言したトークルーム」の情報も必要になります。ここまでをスキーマにまとめると以下のようになるでし

ょう。

```
const MessageSchema = new Schema({
  room:      {type: Schema.Types.ObjectId,   ref: 'Room'},      // 発言されたトークルーム
  character: {type: Schema.Types.ObjectId,   ref: 'Character'}, // 発言したキャラクター
  refer:     {type: Schema.Types.ObjectId,   ref: 'Message'},   // 返信先発言ログのID
  to:        {type: [Schema.Types.ObjectId], ref: 'Character'}, // 返信先キャラクター
  deleted:   Boolean, // 削除フラグ
  timestamp: Date,    // 発言時刻
  name:      String,  // 発言名
  message:   String,  // 発言内容
  icon:      String   // 発言時のアイコン
});
```

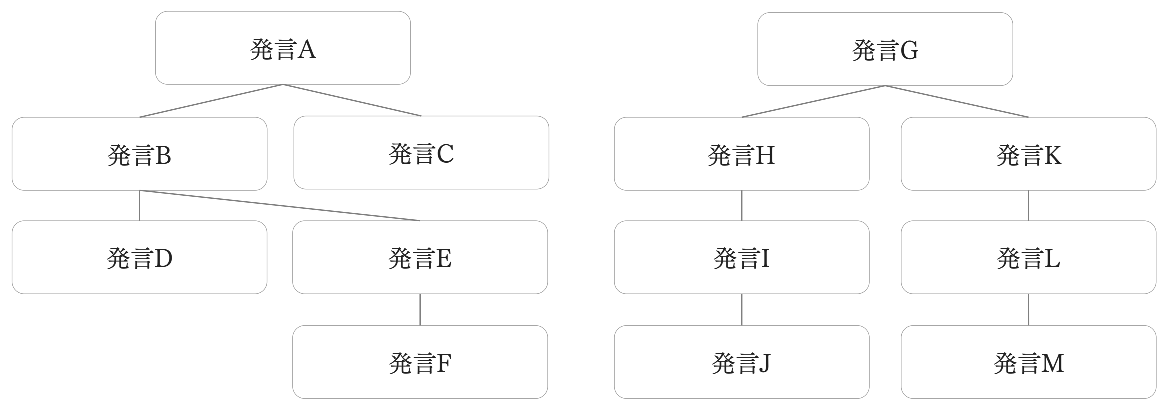
ここからログを検索することについて考えましょう。roomで検索することで発言した部屋については簡単に絞り込むことができます。ここから「全体」「自分」「お気に入り」「関連」「返信ツリー」の5つの絞り込み方法を作ることで要求仕様を満たすトークルームを作ることができます。

「全体」「自分」「お気に入り」については簡単です。「全体」についてはこれ以上絞り込まなければ、「自分」については発言者が自身である発言ログを絞り込めば、「お気に入り」については発言者がお気に入り＋閲覧者のキャラクター配列に適合するものを絞り込めばそれぞれ実現することができます。

「関連」についてはどうでしょうか。「関連」は自身が発言者、もしくは返信先キャラクターに自身が含まれているログを検索します。単純に「自身が発言者 OR 返信先に自身が含まれる」としてもいいのですが、これだと検索条件がやや複雑になってしまいMongoDBの速度低下につながってしまいます。

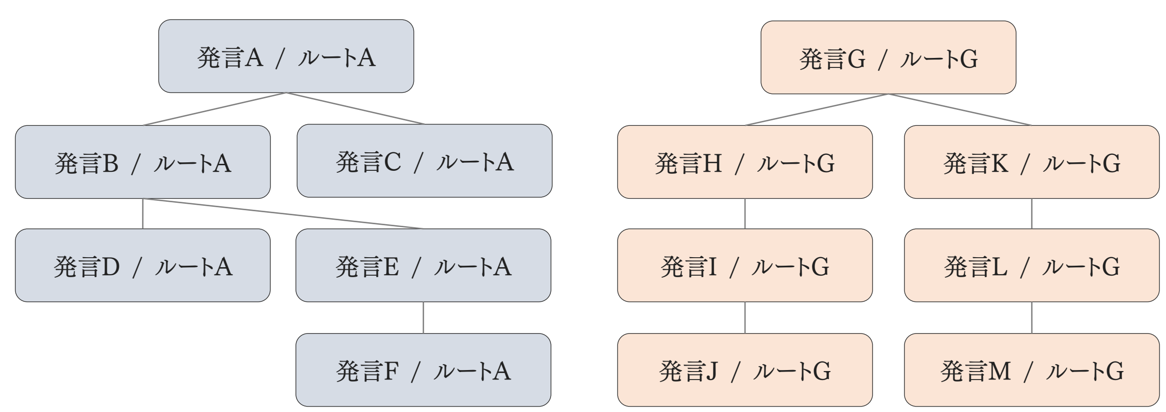
ここで発想を逆転しましょう。検索条件をORで連結するのではなく、検索されるフィールドの内容を繋げるのです。つまりフィールドを「返信先キャラクター」とするのではなく「関連するキャラクター」、すなわち「発言者＋返信先キャラクター」というようにします。このようにすることで自身が発言者の発言ログも返信先に自身が含まれる発言ログも「関連するキャラクターに自身が含まれている」という条件で検索できるようになります。返信先を確認したい場合は「関連するキャラクター」から「発言者」を取り除いたものが返信先になります。

最後に「返信ツリー」です。返信ツリーでは返信に関連したログの内容を確認できるようにします。しかし返信ツリーはこれまでのデータ構造だけでは実現することができません。どうやって表現するかを考えましょう。返信ツリーについて考えるに当たって、どのような返信ツリーの構造が考えられるか図式化してみましょう。以下のようなツリーが考えられます。



　多種多様なツリー構造が考えられますが、どの返信構造にも共通する性質が1つあります。それは「返信を元へ元へと辿っていくと最終的には1つの発言に行き着く」ということです。返信ツリーの実現にはこの性質を利用します。この返信を最初まで辿っていった先の発言ログのことをルート発言ログと呼ぶことにしましょう。ある返信ツリーについてその全体を確認したい場合、「ルート発言ログが共通する発言」を検索することで実現することができます。

　次はどのようにルート発言ログの情報をもたせるかについて考えます。これは返信先からルート発言ログのIDを引き継げば実現できます。図式化すると以下のようになります。



　具体的にどうするかというと、返信ではない発言は自身の発言IDをルート発言ログIDとして保持します。それに対して返信があった場合返信先のルート発言ログIDを引き継ぎます。さらに返信があった場合ルート発言ログIDを引き継ぎ……というようにします。アルゴリズムを簡潔に記述するなら以下のようになるでしょう。

「返信ではない発言」… 自身の発言IDをルート発言ログIDとする。
「返信」… ルート発言ログIDを返信先からコピーする。

　これで「ルート発言ログIDが共通する発言」を検索することで返信ツリーが実現できるようになります。（なお、これは返信ツリーに関連するログをまとめて表示しているだけなので正確には「ツリー」ではありません。しかし本当にツリー構造を実現しようとするとより複雑なアルゴリズムを組まなければならないためここでは扱いません。）

さて、これまでの内容をまとめて具体的にスキーマを組んでいきましょう。まずは設定ファイルにトークルームタイトルの最大の長さを設定しておきます。「backend/config/default.json」を開き、以下の設定を追記して保存してください。

backend/config/default.json

```
  "roomTitleMaxLength": 16
```

設定を保存したら「backend/server/model.js」を開きCharacterSchema宣言の次に以下のようにしてスキーマを宣言します。

backend/server/model.js

```
  （省略）

const RoomSchema = new Schema({
  rno:          {type: Number, sparse: true},    // RNo
  title:        {type: String, required: true, index: false, maxlength: config.roomTitleMaxLength}
, // タイトル
  character:    {type: Schema.Types.ObjectId, ref: 'Character', required: true}, // トークルーム主
  deleted:      {type: Boolean, default: false, index: true}, // 削除フラグ
  creationTime: {type: Date,    default: Date.now},           // 作成日時
  lastUpdate:   {type: Date,    default: Date.now},           // 更新日時
  messageCount: {type: Number,  default: 0}, // 発言数
  tags:         [String], // タグ
  summary:      String,    // トークルームリストで表示される短い説明文
  description:  String     // 説明文
});

const MessageSchema = new Schema({
  room:      {type: Schema.Types.ObjectId,   ref: 'Room',      required: true, index: true}, // 発
言されたトークルーム
  character: {type: Schema.Types.ObjectId,   ref: 'Character', required: true, index: true}, // 発
言したキャラクター
  related:   {type: [Schema.Types.ObjectId], ref: 'Character', index: true},                 // 関
連（発言したキャラクター＋返信先）
  referRoot: {type: Schema.Types.ObjectId,   ref: 'Message',   default: function() {         // ル
ート発言ログ ID
    return this._id; // デフォルト値は自身の発言ログ ID
  }},
  refer:     {type: Schema.Types.ObjectId,   ref: 'Message'}, // 返信先
  deleted:   {type: Boolean, default: false, index: true},    // 削除フラグ
  timestamp: {type: Date, default: Date.now},                 // 発言時刻
  name:      String, // 発言名
  message:   String, // 発言内容
  icon:      String  // 発言時のアイコン
});

  （省略）
```

スキーマをモデル化して外部から参照できるようにするのも忘れないようにしましょう。以下のように変更します。追記された箇所は斜体で表示してあります。

```
backend/server/model.js
  (省略)

const Character = mongoose.model('Character', CharacterSchema);
const Room      = mongoose.model('Room',      RoomSchema      );
const Message   = mongoose.model('Message',   MessageSchema   );

mongoose.connect(`mongodb://${config.dbUser}:${encodeURIComponent(config.dbPassword)}@127.0.0.1:${
config.dbPort}/${config.dbName}`);

module.exports = {
  Character: Character,
  Room:      Room,
  Message:   Message
};
```

4.9.2　初期化処理

パブリックなトークルームにもRNoやトークルーム主になるキャラクターが必要です。そこでENo.0とRNo.0をそれぞれ管理者キャラクター、パブリックトークルームとして作成するようにします。これらはデータを初期化したときに自動で作成された方が便利なので初期化処理を組んでいきましょう。

まずは「backend/config/default.json」に設定を書いていきます。administratorPasswordが管理者キャラクターのパスワードになります。ここは独自のものを使用してください。また、パスワードのハッシュ化も行うためsaltとstretchも設定しておきます。これはフロントエンド側と同じ値を使用してください。同じ値を使用しないとハッシュ化の結果が変わり管理者キャラクターでログインできなくなってしまいます。

```
backend/config/default.json
  "administratorPassword": "8YXdPv*O(_tn",
  "administratorName": "管理者",
  "administratorNickname": "管理者",
  "publicRoomTitle": "ロビー",
  "salt": "Xln&F0DZpqo1",
  "stretch": 1000
```

それでは初期化処理を書いていきましょう。「backend/server/」内に「initialize.js」を作成します。ファイル構成は以下のようになります。

```
server
  api.js
  index.js
  initialize.js
  model.js
  passport.js
  styling.js
```

作成したら以下の内容を記入して保存します。

backend/server/initialize.js

```
const config   = require('config');
const jsSHA    = require('jssha');
const redis    = require('redis').createClient();
const readline = require('readline');
const model    = require('./model.js');

const Character = model.Character;
const Room      = model.Room;
const Message   = model.Message;

const initialize = async() => {
  // 初期化処理
  console.log('初期化処理を行います。');

  // キャラクター、トークルーム、メッセージ情報を全て削除
  await Character.deleteMany({});
  await Room.deleteMany({});
  await Message.deleteMany({});

  // Redisに保存されたセッション情報の削除
  redis.keys(`${config.redisSessionPrefix}*`, (err, sessions) => {
    for (let i = 0; i < sessions.length; i++) {
      redis.del(sessions[i]);
    }
  });

  // 管理者パスワードをハッシュ化
  let s = config.administratorPassword + config.salt;
  for (let i = 0; i < config.stretch; i++) {
    const shaObj = new jsSHA("SHA-256", "TEXT");
    shaObj.update(s);
    s = shaObj.getHash("HEX");
  }

  // 管理者キャラクターをENo.0で作成して保存
  const administrator = new Character({
    eno:      0,
    name:     config.administratorName,
    nickname: config.administratorNickname,
    password: s
  });
  await administrator.save();

  // パブリックトークルームをRNo.0で作成して保存
  const publicRoom = new Room({
    rno:       0,
    title:     config.publicRoomTitle,
    character: administrator._id
  });
  await publicRoom.save();

  console.log('初期化処理が完了しました。');
```

```
}

// コンソールからの入力を受け付けるためのインターフェース
const readlineInterface = readline.createInterface({
  input:  process.stdin,
  output: process.stdout
});


// 本当に初期化するかの確認を行う処理
const initCheck = () => {
  readlineInterface.question('本当に初期化を行いますか？(yesと入力することで初期化)', async (answer) => {
    if (answer == 'yes') {
      await initialize();  // yesと入力されたら初期化処理を行う
    }
    process.exit(0); // 処理が終わるかyes以外が入力されたら実行を終了
  });
};

// 実行してすぐはいろいろ表示が出るので確認を1秒待つ
setTimeout(initCheck, 1000);
```

このファイルを実行すると1秒後に「本当に初期化を行いますか?(yesと入力することで初期化)」という確認が行われ、「yes」と入力されれば初期化処理が行われます。これにはNode.jsがデフォルトで提供しているreadlineというライブラリを使用しています。

初期化処理ではデータベースのコレクションの内容全てを削除している他、Redisに保存された接続情報も全て削除しています。データが完全に初期化されているのに初期化以前のデータにログインできてしまうとバグなどの原因になるためです。

データの削除が終わると管理者キャラクターがENo.0、指定のパスワードで作成され管理者がトークルーム主のパブリックトークルーム「ロビー」がRNo.0に作成されます。ここまでの処理が終わったら「初期化処理が完了しました。」と表示されてプログラムが終了します。

それではこれを「npm run init」から実行できるようにしましょう。「backend/package.json」を開き"scripts"を以下のように書き換えます。

backend/package.json

```
 (省略)

 "scripts": {
   "dev": "nodemon server/index.js --watch config --watch server",
   "start": "node server/index.js",
   "init": "node server/initialize.js"
 },

 (省略)
```

これで「npm run init」から初期化処理を実行できるようになります。それでは初期化を行いましょう。バックエンド側のプログラムをCtrl+Cから実行停止し「npm run init」を実行します。1秒待つと「本当に初期化を行いま

すか？(yesと入力することで初期化)」と確認されるので「yes」と入力しましょう。これでデータベースの内容や接続情報が初期化され管理者キャラクターやパブリックトークルームが作成されます。

MongoDB Compassから確認してみましょう。再度接続し「characters」「messages」の内容を確認してみます。以下のような感じで管理者とパブリックトークルームが作成されているはずです。

```
  _id: ObjectId("5de4f531dd5f3b6070a33346")
> status: Object
> declare: Object
  deleted: false
> tags: Array
> profileImages: Array
  ap: 0
  np: 20
> skill: Array
> story: Array
> explore: Array
> fav: Array
> block: Array
> blocked: Array
> mute: Array
  eno: 0
  name: "管理者"
  nickname: "管理者"
  password: "c4e8bde82affc5e306873bbb81c44192038da5cc76c6b820c310e3e2d5572793"
  csrf: "472bc63f2b90f5ad863bad0e9164b18463305815990f800fcafcecd0ac85b169"
  registrationTime: 2019-12-02T11:27:45.454+00:00
> icons: Array
  __v: 0
```

```
  _id: ObjectId("5de4f531dd5f3b6070a33347")
  deleted: false
  messageCount: 0
> tags: Array
  rno: 0
  title: "ロビー"
  character: ObjectId("5de4f531dd5f3b6070a33346")
  creationTime: 2019-12-02T11:27:45.468+00:00
  lastUpdate: 2019-12-02T11:27:45.468+00:00
  __v: 0
```

初期化が終わったらバックエンド側のコンソールで「npm run dev」を実行してバックエンド側のプログラムを再開しておきましょう。

4.9.3　ENo設定処理の変更

ENo.0に管理者キャラクターが作成されたことで登録時のENo設定処理で設定されるENoがずれてしまっています。なのでENo設定処理を少し変更しましょう。「backend/server/api.js」の「router.post('/characters' …」のtry/catch部分を以下のように書き換えて保存します。

backend/server/api.js

```
（省略）

try {
  // キャラクターを登録
  await character.save();
  const allCharacters = await Character.find({}, {_id: 1});

  // 全てのキャラクターから登録したキャラクターが何番目に該当するか検索
  // それによりENoをセットし、登録されたENoを返す
  // 初期化処理でENo.0が管理者になっているのでi == enoになる
```

```
    for (let i = allCharacters.length - 1; 0 <= i; i--) {
      if (allCharacters[i]._id.toString() == character._id.toString()) {
        await character.update({eno: i});
        return res.status(200).send({eno: i});
      }
    }

    return res.status(500).send();
  } catch (e) {
    return res.status(500).send();
  }

  （省略）
```

これで登録処理はOKです。初期化処理でテスト用に作っていたキャラクターやログイン情報は消えてしまっているので、新規登録ページからまた新規登録しておいてください。

4.9.4　トークルーム作成APIの作成

「/rooms」にPOSTリクエストをすることでトークルームの作成ができるようにしていきます。まずは「backend/server/api.js」からRoomとMessageモデルを呼べるようにしましょう。「backend/server/api.js」を開き、冒頭部分を以下のように書き換えてください。

backend/server/api.js

```
const express   = require('express');
const config    = require('config');
const passport  = require('passport');
const styling   = require('./styling.js');
const Character = require('./model.js').Character;
const Room      = require('./model.js').Room;
const Message   = require('./model.js').Message;
const router    = express.Router();

  （省略）
```

書き換え終わったら他のAPI同様に以下のAPIを追記して保存してください。

backend/server/api.js

```
router.post('/rooms', checkAuthentication, checkCsrf, async (req, res) => {
  try {
    const room = new Room({
      character:   req.user._id,
      title:       req.body.title,
      tags:        req.body.tags,
      summary:     req.body.summary,
      description: styling.profile(req.body.description)
    });

    // トークルームを登録
    await room.save();
    const allRooms = await Room.find({}, {_id: 1});
```

```
    // 全てのトークルームから登録したトークルームが何番目に該当するか検索
    // それによりRNoをセットし、登録されたRNoを返す
    // 初期化処理でRNo.0がパブリックルームになっているのでi == rnoになる
    for (let i = allRooms.length - 1; 0 <= i; i--) {
      if (allRooms[i]._id.toString() == room._id.toString()) {
        await room.update({rno: i});
        return res.status(200).send({rno: i});
      }
    }

    return res.status(500).send();
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

トークルームのタイトル・タグ・概要・説明文を受け取り、接続しているユーザーをトークルーム主としてトークルームを作成しています。また、トークルームの説明文(description)ではプロフィールと同じ装飾処理を行っています。キャラクター登録APIやキャラクター設定APIとほぼ同じなので特に目新しい部分はないかと思います。忘れずにCSRF対策を行うようにしましょう。

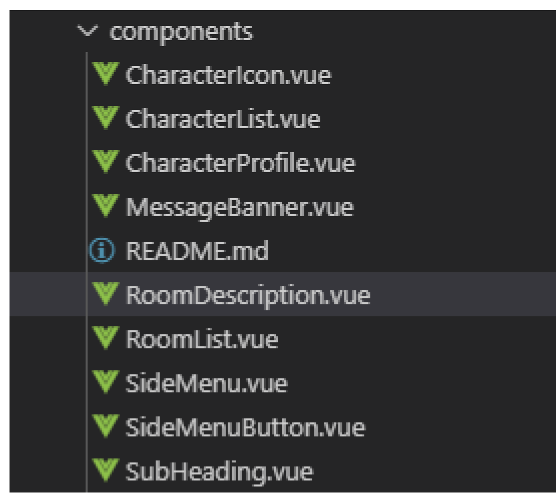
4.9.5 トークルーム作成ページの作成

それではトークルームを作成するページを作りましょう。まずは「nuxt.config.js」にトークルームのタイトルの最大の長さの設定を記述しておきます。「frontend/nuxt.config.js」を開き「env」内に以下の設定を追記して保存してください。この設定の数値はバックエンド側のものと合わせてください。

frontend/nuxt.config.js

```
    roomTitleMaxLength: 16
```

次ですが、プレビューなどのためにトークルームリストやトークルームの説明部分のコンポーネントをこの時点で作っておきます。「frontend/components」内に「RoomList.vue」と「RoomDescription.vue」を作成します。ファイル構成は以下のようになります。



それぞれ以下の内容を記入して保存してください。受け取った値を表示するだけのコンポーネントなので特筆するべき内容はあまりないです。filterを使って日付を見やすい形式に整形していたり「RoomDescription.vue」はプレビュー時の表示モード(preview)、パブリックトークルーム時の表示モード(public)、個室の表示モード(unofficial)の3つの表示モードがあったりというぐらいでしょうか。

frontend/components/RoomList.vue

```
<template>
  <ul class="room-list">
    <li v-for="room in rooms" :key="room.rno">
      <div class="info">
        <nuxt-link class="room-link" :to="`/talk/${room.rno}`">
          <span class="title">{{ room.title }}</span>
          <span class="rno">&lt; RNo.{{ room.rno }} &gt;</span>
        </nuxt-link>
      </div>
      <div class="tags">
        <span v-for="tag in room.tags" :key="tag">
          {{ tag }}
        </span>
      </div>
      <div class="summary">
        {{ room.summary }}
      </div>
      <div class="meta">
        発言数:{{ room.messageCount }} / 最終更新:{{ room.lastUpdate | date }}
      </div>
    </li>
  </ul>
</template>

<script>
import CharacterIcon from '~/components/CharacterIcon.vue'

export default {
  components: {
    CharacterIcon
  },
```

```
  props: ['rooms'],
  filters: {
    date: (str) => {
      const date = new Date(str);
        return (
          ('0' + (date.getMonth() + 1)).slice(-2) + '/' +
          ('0' + (date.getDate()     )).slice(-2) + '(' +
          ['日','月','火','水','木','金','土'][date.getDay()] + ') ' +
          ('0' + (date.getHours()    )).slice(-2) + ':' +
          ('0' + (date.getMinutes()  )).slice(-2)
        );
    }
  }
}
</script>

<style lang="scss" scoped>
.room-list {
  list-style: none;
  border-top: 1px solid lightgray;
  margin: 20px;

  li {
    border-bottom: 1px solid lightgray;
    margin: 0;
    padding: 10px;

    .room-link {
      text-decoration: none;
    }

    .title {
      font-size: 16px;
      font-weight: bold;
      color: #222222;
    }

    .rno {
      font-size: 13px;
      margin-left: 3px;
      color: gray;
    }

    .tags {
      font-size: 14px;
      text-decoration: none;
      color: gray;
    }

    .meta {
      text-align: right;
      line-height: 1.1;
      font-size: 13px;
      color: #AAA;
    }
  }
}
```

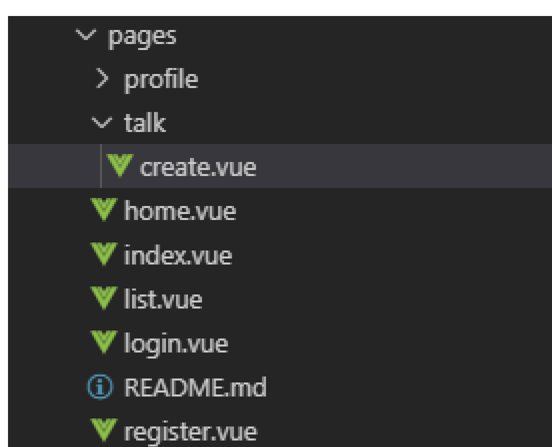
```
</style>
```

frontend/components/RoomDescription.vue

```
<template>
  <section>
    <h2 class="title">
        <span v-if="mode == 'preview'">RNo.---</span>
        <span v-else-if="mode == 'unofficial'">RNo.{{ rno }}</span>
        {{ title }}
    </h2>
    <div class="meta-info" v-if="mode != 'public'">
      &lt; ENo.{{ eno }} {{ nickname }} &gt;  最終更新: {{ lastUpdate | date }}
    </div>
    <div class="button-wrapper">
      <nuxt-link
        v-if="mode == 'public'"
        to="/talk">
        <button class="button">トークルームリスト</button>
      </nuxt-link>
      <nuxt-link
        v-if="(mode == 'unofficial') && (eno == $store.getters['auth/loginCharacter'].eno)"
        :to="`${$route.path}/edit`">
        <button class="button">設定変更</button>
      </nuxt-link>
    </div>
    <section class="description" v-html="description"></section>
  </section>
</template>

<script>
import SubHeading    from '~/components/SubHeading.vue'

export default {
  props: {
    mode:        String, // 表示モード public = パブリックトークルーム、preview = プレビュー、unofficial = 個室
    rno:         Number,
    title:       String,
    eno:         Number,
    nickname:    String,
    lastUpdate:  String,
    description: String
  },
  filters: {
    date: (str) => {
      const date = new Date(str);
        return (
          ('0' + (date.getMonth() + 1)).slice(-2) + '/' +
          ('0' + (date.getDate()     )).slice(-2) + '(' +
          ['日','月','火','水','木','金','土'][date.getDay()] + ') ' +
          ('0' + (date.getHours()    )).slice(-2) + ':' +
          ('0' + (date.getMinutes()  )).slice(-2)
        );
    }
  }
}
</script>
```

```
<style lang="scss" scoped>
$font: 'BIZ UDPGothic', 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ', Meiryo, 'ＭＳ Ｐゴシック
', sans-serif;

.title {
  display: block;
  background: #444;
  color: #EEE;
  margin: 10px 0;
  padding: 10px;
  font-size: 18px;
  font-family: $font;
  letter-spacing: 0;
}

.meta-info {
  color: #666;
  margin: 0 20px;
  font-size: 12px;
  text-align: right;
}

.description {
  margin: 0 20px;
}
</style>
```

コンポーネントを作成したら「/talk/create」からトークルームを作成できるようにします。「frontend/pages」内に「talk」ディレクトリを作成しその中に「create.vue」を作成してください。ファイル構成は以下のようになります。



作成したら以下の内容を記述して保存します。

frontend/pages/talk/create.vue

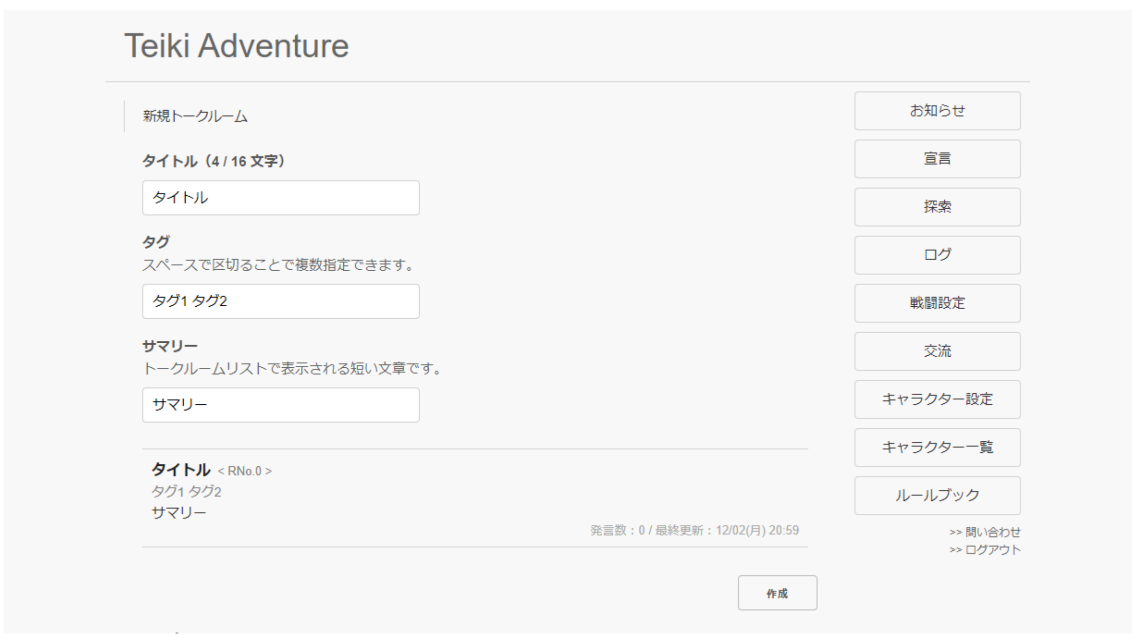
```
<template>
  <section>
    <sub-heading>新規トークルーム</sub-heading>
```

```
    <section class="form">
      <div class="form-title">タイトル ({{ title.length }} / {{ roomTitleMaxLength }} 文字) </div>
      <input class="form-input" type="text" v-model="title" placeholder="タイトル">
    </section>
    <section class="form">
      <div class="form-title">タグ</div>
      <div class="form-description">
        スペースで区切ることで複数指定できます。
      </div>
      <input class="form-input" type="text" v-model="joinedTags" placeholder="タグ">
    </section>
    <section class="form">
      <div class="form-title">サマリー</div>
      <div class="form-description">
        トークルームリストで表示される短い文章です。
      </div>
      <input class="form-input" type="text" v-model="summary" placeholder="サマリー">
    </section>
    <room-list :rooms="[{
      title:        title,
      summary:      summary,
      rno:          0,
      tags:         joinedTags ? joinedTags.split(/\s+/) : [],
      lastUpdate:   (new Date()).toString(),
      messageCount: 0
    }]"/>
    <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
    <div class="button-wrapper">
      <button class="button" @click="create">作成</button>
    </div>
    <section class="form">
      <div class="form-title">説明文</div>
      <div class="form-description">
        説明文はプロフィールと同様の書式で装飾することができます。<br>
        詳しくはルールブックを確認してください。
      </div>
      <textarea class="form-textarea" type="text" v-model="description" placeholder="説明文"></text
area>
    </section>
    <room-description
      mode="preview"
      :title="title"
      :description="description | styling"
      :eno="0"
      :lastUpdate="(new Date()).toString()" />
    <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
    <div class="button-wrapper">
      <button class="button" @click="create">作成</button>
    </div>
  </section>
</template>

<script>
import SubHeading      from '~/components/SubHeading.vue'
import RoomList        from '~/components/RoomList.vue'
import RoomDescription from '~/components/RoomDescription.vue'
import MessageBanner   from '~/components/MessageBanner.vue'
```

```
export default {
  components: {
    SubHeading,
    RoomList,
    RoomDescription,
    MessageBanner
  },
  middleware: 'authenticated',
  head() {
    return {
      title: '新規トークルーム'
    };
  },
  data() {
    return {
      roomTitleMaxLength: process.env.roomTitleMaxLength,

      title:           '',
      joinedTags:      '',
      summary:         '',
      description:     '',
      errorMessage:    '',
      waitingResponse: false
    }
  },
  filters: {
    styling: function(description) {
      if (!description) { // 変換するものがなければ空文字列を返す
        return '';
      }

      return (
        description
        // サニタイズ
          .replace(/&/g, '&amp;')
          .replace(/</g, '&lt;')
          .replace(/>/g, '&gt;')
          .replace(/"/g, '&quot;')
          .replace(/'/g, '&#39;')
        // タグ置換
        // サニタイズしてあるのでその前提で置換する
          .replace(/&lt;s&gt;(.+)&lt;\/s&gt;/sig , '<span class="small">$1</span>')
          .replace(/&lt;l&gt;(.+)&lt;\/l&gt;/sig , '<span class="large">$1</span>')
          .replace(/&lt;b&gt;(.+)&lt;\/b&gt;/sig , '<span class="bold">$1</span>')
        // 改行置換
          .replace(/\n/g, '<br>')
      );
    }
  },
  methods: {
    create: async function() {
      if (this.waitingResponse) {
        // 接続待ち状態であればアラートを表示しreturn、以降の処理を行わない
        return alert('しばらくお待ち下さい');
      }
```

```
      // 入力内容の検証を行う、問題があればエラーメッセージを表示、returnして処理を中断
      if (!this.title) {
        return this.errorMessage = 'タイトルが入力されていません';
      }
      if (this.roomTitleMaxLength < this.title.length) {
        return this.errorMessage = 'タイトルが長すぎます';
      }

      // 問題がなければハッシュ化と接続に入る、接続待ち状態をON(true)に
      this.waitingResponse = true;

      try {
        // 新しいトークルームを登録し、結果を受け取る
        const response = await this.$axios.post('/api/rooms', {
          csrf:        this.$store.getters['auth/loginCharacter'].csrf,
          title:       this.title,
          summary:     this.summary,
          description: this.description,
          tags:        this.joinedTags ? this.joinedTags.split(/\s+/) : []
        });

        // 作成したらそのトークルームのページへリダイレクト
        this.$router.push(`/talk/${response.data.rno}`);
      } catch (e) {
        // エラーが発生した場合エラーメッセージを表示して接続待ち状態をOFF(false)に
        this.errorMessage = '登録中にエラーが発生しました';
        this.waitingResponse = false;
      }
    }
  }
}
</script>
```

ここについても今までやったこととほぼ同じなので特筆するべき内容はあまりありません。キャラクター登録や設定ページとほぼ同じです。CSRFトークンを送るのを忘れないようにしましょう。実際に「http://dev.siroisakana.com/ta/talk/create」にアクセスすると以下のような感じになっています(長いので一部のみ)。



なお、作成が完了すると作成したトークルームにリダイレクトするようになっているのですがページ表示処理を作っていないので404ページへ飛ばされます。リダイレクト先のページはこれから作っていきましょう。

4.9.6　トークルームAPIの作成

それではトークルームにアクセスできるようにします。まずは必要な情報を返すAPIを作りましょう。ここでは「/rooms/(key)」でトークルームAPIにアクセスできるようにします。keyに数値を指定することで指定のRNoのトークルーム、publicを指定することでパブリックトークルームを取得できることにします。また、以下のURLパラメーターもつけることとします。

パラメーター	意味	補足
v	表示モード	favでお気に入り、ownで自分、relで関連ログ表示モード　それ以外は全体
rt	返信ツリー	指定されたルート発言ログIDを持つ返信ツリーを表示
page	ページ	表示するページ　0から始まり指定がなければ0扱い

まずは1ページあたりに表示する発言数を設定ファイルに記述します。「backend/config/default.json」に以下の設定を書き加えて保存してください。

backend/config/default.json
```
"messagesPerPage": 12
```

次に「backend/server/api.js」に以下のAPIを追記して保存してください。

backend/server/api.js
```
router.get('/rooms/:key(\\d+|public\)', checkAuthentication, async (req, res) => {
  try {
```

```
const rno = Number(req.params.key) || 0;
const room = await Room.findOne({rno: rno, deleted: false}, {
  rno:         1,
  title:       1,
  tags:        1,
  lastUpdate:  1,
  character:   1,
  description: 1
}).populate({
  path: 'character',
  select: {
    eno:      1,
    nickname: 1
  }
});

if (!room) { // 指定のトークルームが見つからなければ404を返し中断
  return res.status(400).send();
}

const user = await Character.findById(req.user._id, {
  fav:     1,
  block:   1,
  blocked: 1,
  mute:    1,
  icons:   1
});

if (user.block.concat(user.blocked).includes(room.character._id)) {
  // トークルーム主をブロックしているかブロックされている場合403を返し中断
  return res.status(403).send();
}

const query = { // 最低限のログ検索条件を準備
  room:    room._id,
  deleted: false,
  $and:    [] // 追加の検索条件はこの$andに入れていく
};

if (req.query.rt) { // ログに関連した返信ツリー
  query.$and.push({referRoot: req.query.rt});
}

if (req.query.v) {
  if (req.query.v == 'fav') { // お気に入りログモード
    query.$and.push({character: {$in: user.fav.concat(req.user)}});
  } else if (req.query.v == 'own') { // 自分ログモード
    query.$and.push({character: user._id});
  } else if (req.query.v == 'rel') { // 関連ログモード
    query.$and.push({related: {$in: user._id}});
  }
}

const dismissList = user.block.concat(user.blocked);
if (dismissList.length) {
  // ブロックもしくは被ブロックのキャラクターが存在する場合
  // ブロックか被ブロックのキャラクターの発言もしくは
```

```
      // 返信先に含む発言を非表示にする
      query.$and.push({related: {$nin: dismissList}});
    }

    if (user.mute.length) {
      // ミュートしているキャラクターが存在する場合
      // ミュートがしているキャラクターの発言を非表示にする
      query.$and.push({character: {$nin: user.mute}});
    }

    if (!query.$and.length) { // $andに指定されたものがなければ削除
      delete query.$and;
    }

    const currentPage = Number(req.query.page);

    let messages = await Message.find(query, {
      name:      1,
      character: 1,
      refer:     1,
      referRoot: 1,
      related:   1,
      message:   1,
      timestamp: 1,
      referRoot: 1
    }).sort({timestamp: -1}).skip(currentPage * config.messagesPerPage).limit(config.messagesPerPage + 1).popu
late({
      path: 'character',
      model: 'Character',
      select: {
        _id: 0,
        eno: 1
      }
    }).populate({
      path: 'related',
      model: 'Character',
      select: {
        _id:      0,
        eno:      1,
        nickname: 1
      }
    }).exec();

    let isContinuing = false;
    if (config.messagesPerPage < messages.length) {
      isContinuing = true;
      messages = messages.slice(0, config.messagesPerPage);
    }

    const formattedIcons = []; // iconsに_idが含まれているのでなくしておく
    for (let i = 0; i < user.icons.length ;i++) {
      formattedIcons.push({
        name: user.icons[i].name,
        url:  user.icons[i].url
      });
    }
```

```
    return res.status(200).send({
      room: {
        character: {
          eno:      room.character.eno,
          nickname: room.character.nickname,
        },
        description: room.description,
        lastUpdate:  room.lastUpdate,
        rno:         room.rno,
        tags:        room.tags,
        title:       room.title
      },
      user: {
        icons: formattedIcons
      },
      messages:     messages,
      isContinuing: isContinuing
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

長くて複雑ですが少しずつ読み解いていきましょう。まずはreq.paramsから取得するべきトークルームのRNoを取得し、表示に必要な情報を取得しています。また、トークルーム主のENoや短縮名情報もあったほうがいいのでpopulateで取得しています。このとき指定のトークルームが削除されたりそもそもそのRNoのトークルームが存在していなかったりなどでトークルームが取得できないことがあるので、そういう場合は404を返して処理を中断します。

次に部屋にアクセスしているユーザーのお気に入り・ブロック・被ブロック・ミュート・アイコンの情報を取得します。このとき、部屋主をブロックしているもしくはブロックされているときは403を返して処理を中断します。

特に問題がなければ発言の取得処理に移ります。「指定の部屋での発言かつ削除されていない」という最低限の検索条件を用意しておき、それに条件を追加していく形で検索条件を組み上げていきます。URLパラメーターなどによってそれぞれ以下のような条件が追加されています。

rt=(ルート発言ログID)	ログに関連した返信ツリー
v=fav	アクセスしているユーザーのお気に入りキャラ＋自身の発言
v=own	アクセスしているユーザー自身の発言
v=rel	アクセスしているユーザーへの返信＋自身の発言
ブロックされているもしくは ブロックしているキャラクターがいる	ブロック/被ブロックキャラクターの発言および そのキャラクターへの返信を検索対象外に
ミュートしているキャラクターがいる	ミュートしているキャラクターの発言を検索対象外に

条件が組み上がったら検索に入ります。検索条件と取り出したいフィールドを指定し、投稿時間降順で並び替えます。また、pageの指定により取得するべき発言ドキュメントの範囲のみを取得している他、発言者のENoおよび返信先キャラクターのENoと短縮名を取得しています。

ここでは「取得するべき発言ドキュメントの件数+1」件を取得しています。なぜかというと12件だけ表示したい場合に12件だけ取得した場合、「12件だけ表示されているがまだ検索結果は続く」のか「この12件で終わりでありもう検索結果に続きはない」のか区別がつかなくなってしまいます。ここで13件取得しておくことで13件取得できていれば「検索結果はまだ続く」、13件に届いていない場合「検索結果はここで終わり」なのかを見分けることができます。ここでは検索結果が続くかどうかの情報はisContinuingに格納してあり、検索結果は取得するべきドキュメントの件数まで削ってあります。

最後に返すデータを整形し、データを返しています。以上がトークルームおよびトークルームの発言内容を返す処理になります。

4.9.7　ダイス機能

トークルームへのメッセージを書き込むとき装飾の他にもダイスが振れるようになっていると便利です。ということで装飾システムにダイス機能を組み込みましょう。ここでは「<1d100>」「<1d6>」を入力するとそれぞれ1d100、1d6の結果をそれぞれ表示するものとします。「backend/server/styling.js」を以下のように書き換えて保存します。

backend/server/styling.js

```
const sanitize = (str) => {
  // 受け取った文字列をサニタイズする
  if (!str) { // 値がない場合そのまま空文字列を返す
    return '';
  }

  return (
    str
      .replace(/&/g, '&amp;')
      .replace(/</g, '&lt;')
      .replace(/>/g, '&gt;')
      .replace(/"/g, '&quot;')
      .replace(/'/g, '&#39;')
  );
};

const profile = (str) => {
  // 受け取った文字列をサニタイズして指定のタグで装飾し、改行させる
  // <s> ～ </s> 文字を小さくする
  // <l> ～ </l> 文字を大きくする
  // <b> ～ </b> 文字を太くする
  return (
    sanitize(str)
      .replace(/&lt;s&gt;(.+)&lt;\/s&gt;/sig , '<span class="small">$1</span>')
      .replace(/&lt;l&gt;(.+)&lt;\/l&gt;/sig , '<span class="large">$1</span>')
      .replace(/&lt;b&gt;(.+)&lt;\/b&gt;/sig , '<span class="bold">$1</span>')
      .replace(/\n/g, '<br>')
  );
};

const dice = (max) => {
  // 指定の面数のダイス結果を生成する
  const dice = Math.floor(Math.random() * max) + 1;
  return `【1d${max}: ${dice}】`;
```

```
};

const message = (str) => {
  // 受け取った文字列にプロフィールと同じ装飾をした上でさらにダイス機能をつける
  // <1d100> 1d100の結果を生成
  // <1d6>   1d6の結果を生成
  let s = profile(str);
  while (true) {
    if (s.match(/&lt;1d100&gt;/i)) { // <1d100>があれば1d100の結果に置換
      s = s.replace(/&lt;1d100&gt;/i, `<span class="dice">${dice(100)}</span>`);
    } else if (s.match(/&lt;1d6&gt;/i)) { // <1d6>があれば1d6の結果に置換
      s = s.replace(/&lt;1d6&gt;/i, `<span class="dice">${dice(6)}</span>`);
    } else { // どちらもなければ完了
      break;
    }
  }
  return s;
};

module.exports = {
  sanitize: sanitize,
  profile:  profile,
  message:  message
};
```

処理を追っていきましょう。message関数を呼び出すとまずprofile(str)が呼び出されサニタイズされた上でプロフィールと同じ装飾がほどこされます。「<1d100>」「<1d6>」はこれによってそれぞれ「<1d100>」「<1d6>」に変換されています。

次にwhile文で変換処理を繰り返し行います。「<1d100>」や「<1d6>」が見つかればそれぞれダイス結果に1つずつ変換されます。一気に変換してしまうと複数「<1d100>」などが記入されていた場合にも全て同じ結果になってしまうので1つずつ変換しています。「<1d100>」や「<1d6>」が見つからなくなれば変換完了として結果の文字列を返します。

4.9.8　発言APIの作成

次にトークルームへの書き込みAPIを作成します。「/rooms/(key)」にPOSTすることで書き込みを行うようにしましょう。「backend/server/api.js」に以下のAPIを書き加えて保存してください。

backend/server/api.js

```
router.post('/rooms/:key(\\d+|public\)', checkAuthentication, checkCsrf, async (req, res) => {
  try {
    const rno = Number(req.params.key) || 0;
    const room = await Room.findOne({rno: rno, deleted: false}, {character: 1});

    if (!room) { // 指定のトークルームが見つからなければ404を返し中断
      return res.status(404).send();
    }

    const user = await Character.findById(req.user._id, {
      nickname: 1,
      block:    1,
```

```
      blocked:  1
    }); // ユーザーのブロック関連リストを取得

    if (user.block.concat(user.blocked).includes(room.character)) {
      // トークルーム主をブロックしているかブロックされている場合403を返し中断
      return res.status(403).send();
    }

    const doc = {
      room:      room._id,
      character: req.user._id,
      name:      req.body.name || user.nickname, // 名前指定がなければ短縮名を設定
      icon:      req.body.icon,
      message:   styling.message(req.body.message)
    };

    if (req.body.refer) { // 返信だった場合
      const refer = await Message.findById(req.body.refer, {
        related:   1,
        referRoot: 1
      });

      // 返信先が存在しなければ400を返して中断
      if (!refer) {
        return res.status(400).send();
      }

      doc.refer     = refer._id;
      doc.referRoot = refer.referRoot;
      doc.related   = refer.related.includes(req.user._id) ? refer.related : refer.related.concat(
req.user._id);
      // 関連キャラクターに自身が含まれていない場合追加
    } else {
      doc.related = [req.user._id];
    }

    const message = new Message(doc);
    await message.save();

    res.status(200).send(); // とりあえずレスポンスを返しておいてから

    return room.update({$inc: {messageCount: 1}, lastUpdate: Date.now()});
    // 発言数を1増やし更新日時をアップデート
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

処理の流れを追っていきましょう。トークルームを取得し発言するユーザーの情報を取得、トークルームが存在しなかったりトークルーム主をブロックしていたりブロックされていたりした場合はエラーコードを返して処理を中断する、という処理はトークルーム取得時とほぼ同じです。

問題がなければ発言内容のドキュメントを作っていきます。まず基本的な内容として発言先トークルーム、発言

キャラクター、名前、発言アイコン、メッセージを設定します。メッセージからはstyling.messageが呼び出されているのでこれによって装飾されたりダイスが振られたりします。

次に返信かどうかで処理を分けます。返信である場合は返信先を取得し返信先IDを設定します。また、ルート発言ログIDを返信先から引き継ぎ、関連キャラクターに自身が含まれていない場合追加します。返信でない場合単に関連キャラクターに自身のIDを設定します。

ドキュメント内容を作り終えたら保存し200を返します。その後、トークルームの発言数を1増やし最終更新日時を今の時刻にアップデートします。この処理については発言に必要な内容と一切関わりがないためとりあえずレスポンスを返しておいてから行っています。これにより若干サーバーの応答速度がよくなります。

4.9.9　発言削除APIの作成

最後に発言削除APIを作ります。「/rooms/(key)/(logid)」にDELETEリクエストを送ることで指定の発言が削除されるようにしましょう。「backend/server/api.js」に他API同様に以下のAPIを追記してください。

backend/server/api.js

```
router.delete('/rooms/:key(\\d+|public\)/:logid', checkAuthentication, checkCsrf, async (req, res) => {
  try {
    const rno = Number(req.params.key) || 0;

    const message = await Message.findById(req.params.logid, {
      character: 1
    }).populate({
      path: 'room',
      model: 'Room',
      select: {
        _id: 0,
        rno: 1
      }
    });

    if (!message) { // 指定のログIDの発言が見つからなければ404を返して中断
      return res.status(404).send();
    }

    if (message.room.rno != rno) { // rnoの指定がおかしければ400を返して中断
      return res.status(400).send();
    }

    // 削除しようとしている発言の発言者がアクセスしているユーザーと違う場合は403を返して中断
    if (message.character.toString() != req.user._id.toString()) {
      return res.status(403).send();
    }

    // 削除フラグをONに
    await message.update({deleted: true});
    res.status(200).send(); // とりあえずレスポンスを返しておいてから

    return Room.findOneAndUpdate({rno: rno}, {$inc: {messageCount: -1}}); // 指定のトークルームの発言数を1減らす
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
```

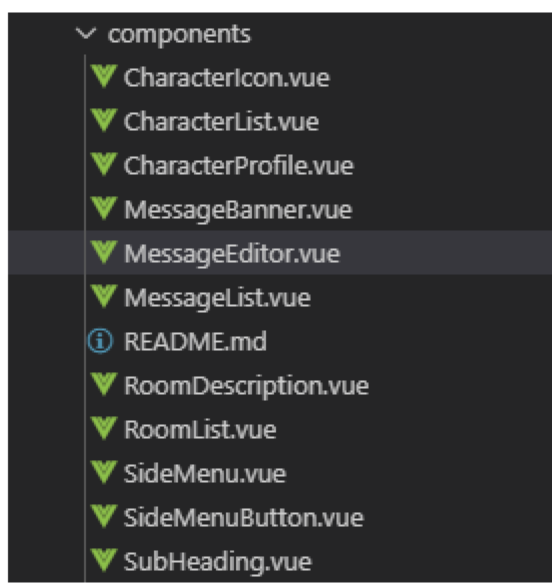
```
});
```

ログIDで発言を検索し発言が見つからなかったりログとトークルームの指定がミスマッチしていたり発言者と違うユーザーが発言を削除しようとしていたりした場合はエラーコードを返して処理を中断するようにしています。問題がなければ発言の削除フラグをONにしてひとまずレスポンスを返します。そしてトークルームの発言数を1減らしています。

これでトークルーム関連のAPIの作成は完了です。

4.9.10 トークルームページの作成

それではAPIにアクセスしてページを表示しましょう。まずは発言リストと発言エディターのコンポーネントを用意しておきます。「frontend/components」内に「MassageList.vue」と「MessageEditor.vue」を作成します。ファイル構成は以下のようになります。



作成したらそれぞれ次の内容を記述して保存してください。

frontend/components/MessageList.vue

```
<template>
  <section :class="1 < messages.length ? 'message-list' : 'message-list-oneitem'">
    <div class="message-wrapper" v-for="message in messages" :key="message._id">
      <div class="icon">
        <character-icon :src="message.icon"/>
      </div>
      <div class="message">
        <div v-if="message.refer" class="refers-wrapper">
          <nuxt-link class="refers" :to="`${$route.path}?rt=${message.referRoot}`">
            &gt;&gt;
            <span v-for="related in message.related" :key="related.eno">
              <span v-if="(message.related.length == 1) || (related.eno != message.character.eno)">
                {{ related.nickname }}({{ related.eno }})
              </span>
```

```
            </span>
          </nuxt-link>
        </div>
        <div class="info">
          {{ message.name }} <span class="eno">(ENo.{{message.character.eno ? message.character.eno : "---"}})
</span>
        </div>
        <div class="body" v-html="message.message"></div>
        <div class="actions">
          <div
            v-if="message.character.eno == $store.getters['auth/loginCharacter'].eno"
            class="action"
            @click="deleteMessage(message)">
            削除
          </div>
          <div
            class="action"
            @click="replyMessage(message)">
            返信
          </div>
        </div>
      </div>
    </div>
  </section>
</template>

<script>
import CharacterIcon from '~/components/CharacterIcon.vue'

export default {
  components: {
    CharacterIcon
  },
  props: ['messages'],
  methods: {
    deleteMessage: function (message) {
      this.$emit('delete-click', message);
    },
    replyMessage: function (message) {
      this.$emit('reply-click', message);
    }
  }
}
</script>

<style lang="scss" scoped>
.message-list {
  column-count: 2;
  column-gap: 20px;
  column-rule: 1px solid lightgray;

  &-oneitem {
    column-count: 2;
    column-gap: 20px;
  }
}
```

```
.message-wrapper {
  border: 1px solid lightgray;
  padding: 8px;
  border-radius: 4px;
  break-inside: avoid-column;
  margin-bottom: 10px;
  display: flex;
}

.message {
  box-sizing: border-box;
  margin: 0 8px;
  width: 100%;
  word-break: break-all;


  .info {
    font-weight: bold;
    color: #666;

    .eno {
      font-weight: normal;
      font-size: 12px;
      color: #CCC;
    }
  }

  .refers-wrapper {
    line-height: 1.1;

    .refers {
      text-decoration: none;
      color: #AAA;
      font-size: 12px;
    }
  }
}

.actions {
  width: 100%;
  display: flex;
  justify-content: flex-end;

  .action {
    display: flex;
    justify-content: center;
    padding: 2px;
    margin-left: 4px;
    min-width: 40px;
    border: 1px solid lightgray;
    font-size: 12px;
    border-radius: 4px;
    cursor: pointer;
  }
}
</style>
```

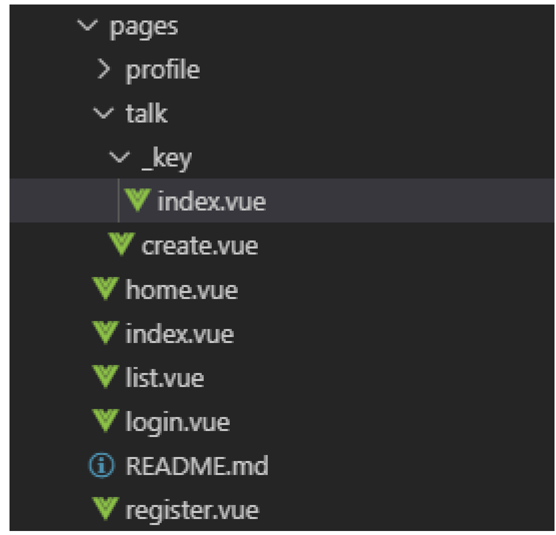
frontend/components/MessageEditor.vue

```
<template>
  <section>
    <div class="editor-wrapper">
      <div v-if="reply" @click="cancelReply">
        返信先（クリックでキャンセル）
        <div class="reply">
          <message-list
            :messages="[reply]"/>
        </div>
      </div>
      名前
      <input type="text" class="name-input" v-model="name">
      アイコン
      <select v-model="icon">
        <option :value="null">
          -- アイコンを選択 --
        </option>
        <option v-for="(iconOption, index) in icons" :value="iconOption.url" :key="index">
          {{ iconOption.name }}
        </option>
      </select>
      <textarea class="editor" v-model="message" placeholder="メッセージ"></textarea>
      <div class="button-wrapper">
        <button class="button" @click="sendClick">送信</button>
      </div>
    </div>
  </section>
</template>

<script>
import MessageList from '~/components/MessageList.vue'

export default {
  components: {
    MessageList
  },
  props: {
    icons: Array,
    reply: Object
  },
  data() {
    return {
      name:    '',
      icon:    null,
      message: ''
    }
  },
  methods: {
    reset: function() {
      this.message = '';
    },
    cancelReply: function() {
      this.$emit('cancel-reply');
    },
    sendClick: function() {
```

```
      this.$emit('send-click', {
        name:    this.name,
        icon:    this.icon,
        message: this.message
      });
    }
  }
}
</script>

<style lang="scss" scoped>
.editor-wrapper {
  margin: 0 20px;
}

.reply {
  margin: 10px 0 20px 10px;
}

.editor {
  width: 100%;
  height: 120px;
  resize: none;
  margin: 0;
}

.name-input {
  margin-right: 10px;
}

.button {
  margin-bottom: 0;
}
</style>
```

それぞれのコンポーネントにカスタムイベントがいろいろと指定されていますがこれは後々解説します。

さて、それではページを作っていきます。「/talk/(key)」で内容を表示します。「/talk/(key)」内にはさらに別のページも用意したいので、「frontend/pages」内の「/talk/_key/index.vue」にトークルームページを作成します。「frontend/pages/talk」内に新たに「_key」ディレクトリを作成し、その中に「index.vue」を作成してください。ファイル構成は以下のようになります。



作成したら以下の内容を記述して保存します。

frontend/pages/talk/_key/index.vue

```
<template>
  <section>
    <room-description
      :mode="$route.params.key == 'public' ? 'public' : 'unofficial'"
      :rno="room.rno"
      :title="room.title"
      :eno="room.character.eno"
      :nickname="room.character.nickname"
      :lastUpdate="room.lastUpdate"
      :description="room.description"/>
    <hr>
    <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
    <message-editor
      ref="messageEditor"
      :reply="reply"
      :icons="user.icons"
      @cancel-reply="cancelReply"
      @send-click="send"/>
    <hr>
    <div class="modelink-wrapper">
      <nuxt-link :class="['modelink', {current: mode == 'all'}]" :to="`${$route.path}?v=all`">全体</nuxt-link>
      <nuxt-link :class="['modelink', {current: mode == 'rel'}]" :to="`${$route.path}?v=rel`">関連</nuxt-link>
      <nuxt-link :class="['modelink', {current: mode == 'own'}]" :to="`${$route.path}?v=own`">自分</nuxt-link>
      <nuxt-link :class="['modelink', {current: mode == 'fav'}]" :to="`${$route.path}?v=fav`">お気に入り</nuxt-
link>
    </div>
    <message-list
      :messages="messages"
      @delete-click="deleteMessage"
      @reply-click="replyMessage"/>
    <div class="pagelink-wrapper">
      <nuxt-link
        class="pagelink"
        v-if="Number($route.query.page)"
        :to="`${$route.path}${queryWithoutPage}&page=${Number($route.query.page) - 1}`">
```

```
        前のページへ
      </nuxt-link>
      <nuxt-link
        class="pagelink"
        v-if="isContinuing"
        :to="`${$route.path}${queryWithoutPage}&page=${Number($route.query.page || 0) + 1}`">
        次のページへ
      </nuxt-link>
    </div>
  </section>
</template>

<script>
import RoomDescription from '~/components/RoomDescription.vue'
import MessageBanner   from '~/components/MessageBanner.vue'
import MessageEditor   from '~/components/MessageEditor.vue'
import MessageList     from '~/components/MessageList.vue'

export default {
  components: {
    RoomDescription,
    MessageBanner,
    MessageEditor,
    MessageList
  },
  middleware: 'authenticated',
  watchQuery: true,
  validate({params}) {
    return /^[1-9][0-9]*$/.test(params.key) || params.key == "public";
  },
  head() {
    return {
      title: this.room.rno ? `RNo.${this.room.rno} ${this.room.title}` : this.room.title
    };
  },
  data() {
    return {
      reply:           null,
      errorMessage:    '',
      waitingResponse: false
    }
  },
  asyncData: async function(context) {
    const queries  = context.route.fullPath.slice(context.route.path.length);
    const response = await context.$axios.get(`/api/rooms/${context.params.key}${queries}`);
    return {
      room: {
        character: {
          eno:       response.data.room.character.eno,
          nickname:  response.data.room.character.nickname
        },
        description: response.data.room.description,
        lastUpdate:  response.data.room.lastUpdate, // 通信によりDate型ではなくString型になっているので注意
        rno:         response.data.room.rno,
        tags:        response.data.room.tags,
        title:       response.data.room.title
      },
```

```
      user: {
        icons:       response.data.user.icons
      },
      messages:      response.data.messages,
      isContinuing:  response.data.isContinuing
    };
  },
  computed: {
    mode: {
      get() {
        if (
          this.$route.query.v == 'fav' ||
          this.$route.query.v == 'own' ||
          this.$route.query.v == 'rel'
        ) {
          return this.$route.query.v;
        } else {
          return 'all';
        }
      }
    },
    queryWithoutPage: {
      get() {
        const queries = [];
        for (const prop in this.$route.query) {
          if (prop != 'page') {
            queries.push(`${prop}=${this.$route.query[prop]}`);
          }
        }
        return '?' + queries.join('&');
      }
    }
  },
  methods: {
    replyMessage: function(message) {
      this.reply = message;
    },
    cancelReply: function() {
      this.reply = null;
    },
    deleteMessage: async function(message) {
      // 接続に入る、接続待ち状態をON(true)に
      this.waitingResponse = true;

      try {
        // 新しいメッセージを送信
        await this.$axios.delete(`/api/rooms/${this.$route.params.key}/${message._id}`, {
          data: {
            csrf: this.$store.getters['auth/loginCharacter'].csrf
          }
        });

        // メッセージ情報を再取得して情報を更新
        const queries  = this.$route.fullPath.slice(this.$route.path.length);
        const response = await this.$axios.get(`/api/rooms/${this.$route.params.key}${queries}`);
        this.room.lastUpdate = response.data.room.lastUpdate;
        this.messages        = response.data.messages;
```

```
        this.isContinuing    = response.data.isContinuing;

        // 入力内容をリセットし接続待ち状態をOFFに
        this.waitingResponse = false;
      } catch (e) {
        // エラーが発生した場合エラーメッセージを表示して接続待ち状態をOFF(false)に
        this.errorMessage = 'メッセージ削除中にエラーが発生しました';
        this.waitingResponse = false;
      }
    },
    send: async function(data) {
      if (this.waitingResponse) {
        // 接続待ち状態であればアラートを表示しreturn、以降の処理を行わない
        return alert('しばらくお待ち下さい');
      }

      // 入力内容の検証を行う、問題があればエラーメッセージを表示、returnして処理を中断
      if (!data.message) {
        return this.errorMessage = 'メッセージが入力されていません';
      }

      // 問題がなければ接続に入る、エラーメッセージを消去し接続待ち状態をON(true)に
      this.errorMessage = '';
      this.waitingResponse = true;

      try {
        // 新しいメッセージを送信
        await this.$axios.post(`/api/rooms/${this.$route.params.key}`, {
          csrf:    this.$store.getters['auth/loginCharacter'].csrf,
          name:    data.name,
          icon:    data.icon,
          message: data.message,
          refer:   this.reply ? this.reply._id : undefined
        });

        // メッセージ情報を再取得して情報を更新
        const queries  = this.$route.fullPath.slice(this.$route.path.length);
        const response = await this.$axios.get(`/api/rooms/${this.$route.params.key}${queries}`);
        this.room.lastUpdate = response.data.room.lastUpdate;
        this.messages        = response.data.messages;
        this.isContinuing    = response.data.isContinuing;

        // 入力内容をリセットし接続待ち状態をOFFに
        this.$refs.messageEditor.reset();
        this.waitingResponse = false;
      } catch (e) {
        // エラーが発生した場合エラーメッセージを表示して接続待ち状態をOFF(false)に
        this.errorMessage = '登録中にエラーが発生しました';
        this.waitingResponse = false;
      }
    }
  }
}
</script>

<style lang="scss" scoped>
.modelink-wrapper {
```

```
  padding-bottom: 20px;
  display: flex;
  justify-content: flex-end;

  .modelink {
    display: inline-flex;
    justify-content: center;
    min-width: 50px;
    padding: 4px 12px;
    margin: 0 10px;
    border: 1px solid #333;
    border-radius: 4px;
    text-decoration: none;
    font-weight: bold;
    color: #666;
  }

  .current {
    background-color: #666;
    color: #EEE;
  }
}

.pagelink-wrapper {
  padding: 20px;
  display: flex;
  justify-content: space-around;

  .pagelink {
    display: inline-flex;
    justify-content: center;
    min-width: 120px;
    padding: 4px 12px;
    border: 1px solid #333;
    border-radius: 4px;
    text-decoration: none;
    font-weight: bold;
    color: #666;
  }
}
</style>
```

「http://dev.siroisakana.com/ta/talk/create」からトークルームを作成すると以下のような感じになっているはずです。

いろいろ操作を行ってみてください。きちんと発言を送信できたりできるようになっていたり、発言内容を絞り込めたり、返信や発言の削除も行えるようになっているはずです。それではどういう仕組みになっているかを見ていきましょう。

まずは発言の送信です。これは主にMessageEditorコンポーネントが関わります。MessageEditorには使うことができるアイコンの情報が渡されており、その情報を元にして入力欄などを生成しています。内部的な値は発言名である「name」、発言時のアイコンのURLを示す「icon」、発言内容の「message」の3つで、これらの値は送信ボタンを押したときに「send-click」カスタムイベントから送られます。「index.vue」ではそのカスタムイベントと値を受け取ってAPIに値を送信しています。

次に発言の削除です。これは主にMessageListコンポーネントが関わります。MessageListでは自分のキャラクターの発言には「削除」というボタンがつくようになっています。このボタンをクリックすると削除するメッセージの値とともに「delete-click」カスタムイベントが発生するようになっており、「index.vue」ではこの情報を元にして発言削除APIにアクセスしています。

次は返信です。MessageListの「返信」ボタンをクリックすると「reply-click」カスタムイベントが返信先の情報とともに送られます。「index.vue」ではこれをreplyという値に入れており、これはMessageEditorにv-bind(:)で渡されているためMessageEditorに返信先情報が表示されます。replyに値が入っている状態、つまり返信が有効な状態で発言を行おうとする場合は発言時にAPIに返信先情報を渡すようにしてあります。なお、MessageEditorの返信先情報をクリックすると「cancel-reply」カスタムイベントが発生するようになっており、これによって「index.vue」はreplyの情報をリセットし返信をとりやめます。

MassageListでは返信の発言は関連キャラクター情報を元に返信先のENoと短縮名の一覧を表示するようになっているのですが、「関連キャラクターが1キャラのみ」もしくは「関連キャラクターのうち対象が発言者自身ではない」という条件で関連キャラクターから返信先を絞り込むようにしてあります。これはなぜかというと、関連キャラクターが1キャラクターのみ、すなわち返信先が自身である場合は自己レスとみなせるので返信先に発言者自身を表示したほうがわかりやすいのですが、そうではない場合、すなわち「関連キャラクターが2キャラクター以上」の場

合は「自身以外のキャラクターに返信を行っている」とみなすことができ、こういう場合は返信先には発言者自身は表示されないほうがわかりやすいです。そのためこのような条件になっています。

ページング処理は「今が1ページ目以降か？」「isContinuingがtrueか？」という情報により表示を行っており、前者の条件が満たされていれば「前のページへ」、後者の条件が満たされていれば「次のページへ」を表示するようになっています。今回はpage以外にもURLパラメーターが存在しうるため、算出プロパティを使ってpage以外のURLパラメーターを分けてpageの値のみを変更したリンクを作るようにしてあります。

「全体」「関連」「自分」「お気に入り」のところはそれぞれへのリンクになっている他、modeという算出プロパティでそれぞれ該当のモードであれば"all", "rel", "own", "fav"の値が返るようになっており、その内容に応じて色を変えるようにしています。なおページングとモード切り替えはURLパラメーターによって行われているため「index.vue」で「watchquery: true」を有効にしてあります。

またRoomDescriptionでは表示モードが"public"であればトークルームリストへのリンクが、表示モードが"unofficial"かつそれが自分の立てた個室トークルームであれば設定ページへのリンクがそれぞれ表示されるようになっています。まだリンク先は未作成ですがこれらのリンクをクリックすることでそれらのページへ飛べるようにしてあります。

また、細かい所として「index.vue」ではRNoが0のときはRNoをタイトルに表示しないようにしたりしてあります。

これが大まかなトークルームページの仕組みです。いろいろなコンポーネントや処理が絡み合って複雑ですが、処理の1つ1つの流れを丁寧に読み取っていけば全体がどのようになっているのか把握できるはずです。

4.9.11 トークルームリストAPIの作成

次はトークルームリストを表示できるようにしましょう。まずはAPIを作っていきます。APIを作る前に設定を「backend/config/default.json」に記入しておきます。ファイルを開いて以下の設定を追記して保存してください。ここではトークルームリスト1ページあたりに表示する件数を指定してあります。

backend/config/default.json

```
"roomListItemsPerPage": 20
```

設定を保存したらAPIを作りましょう。トークルームリストではURLパラメーターによって検索機能が使えるようにします。以下のようなURLパラメーターを使えるようにします。

パラメーター	意味	補足
tags	タグ	タグで検索　タグは空白で区切ることで複数指定可能
title	タイトル	タイトルで検索
fav	お気に入りキャラがトークルーム主	fav=tでお気に入りキャラがトークルーム主のものを検索

それではAPIを組んでいきます。「backend/server/api.js」を開き以下のAPIを記述して保存してください。

backend/server/api.js

```
router.get('/rooms', checkAuthentication, async (req, res) => {
  try {
```

```
const character = await Character.findById(req.user._id, {
  fav:     1,
  block:   1,
  blocked: 1
});

const query = { // 最低限のトークルーム検索条件を準備
  deleted: false,
  rno:     {$gte: 1},
  $and:    [] // 追加の検索条件はこの$andに入れていく
};

if (req.query.tags) {
  req.query.tags.split(/\s+/).forEach((tag) => {
    query.$and.push({tags: tag});
  });
}

if (req.query.title) {
  query.$and.push({title: new RegExp(req.query.title)});
}

if (req.query.fav == 't') {
  query.$and.push({character: {$in: character.fav}});
}

const dismissList = character.block.concat(character.blocked);
if (dismissList.length) {
  query.$and.push({character: {$nin: dismissList}});
}

if (!query.$and.length) { // $andに指定されたものがなければ削除
  delete query.$and;
}

const currentPage = Number(req.query.page) || 0;

const rooms = (
  await Room.find(query)
    .sort({lastUpdate: -1})
    .skip(config.roomListItemsPerPage * currentPage)
    .limit(config.roomListItemsPerPage + 1) // 必要な件数より1件多く取得する
    .select({
      _id:          0,
      rno:          1,
      title:        1,
      summary:      1,
      tags:         1,
      lastUpdate:   1,
      messageCount: 1
    })
    .exec()
);

const isContinuing = config.roomListItemsPerPage < rooms.length;
// 必要な件数+1取得できていたらまだ検索結果には続きがある
```

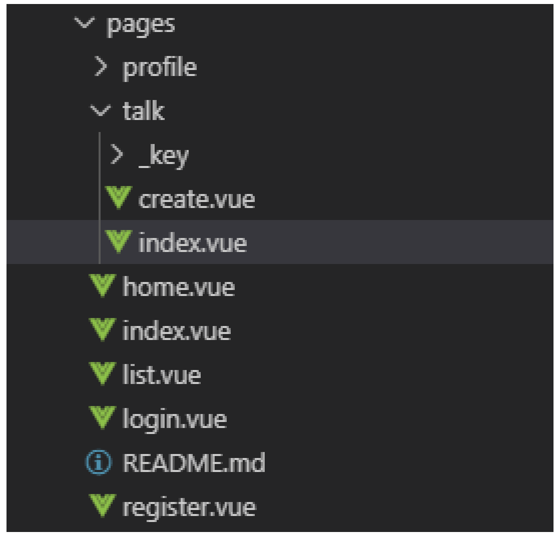
```
    return res.status(200).send({
      list:         rooms.slice(0, config.roomListItemsPerPage), // 必要な件数以下になるように検索結
果を削る
      isContinuing: isContinuing
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

トークルームでメッセージを取得したときとほぼ同じような感じです。「削除されておらずRNoが1以上のもの」を基本的な検索条件に設定しておりURLパラメーターの内容で検索条件を追加しています。なおtagsは複数指定可能なので空白で区切ったタグの個数分$andに検索条件を追加していっています。また、ブロック/被ブロック状況に応じてそのキャラクターが部屋主のトークルームを表示しないようにしてあります。

検索条件が決まったらその条件で検索し最終更新日時降順で並び替え、指定のページの内容+1件のみの結果を取得しています。これにより+1件が取得できているかどうかでまだ検索結果は続くのかどうか、という情報を返せるようにしてあり、ここまでの処理が終わったらトークルームリストと結果が続くのかどうかの情報を返しています。

4.9.12 トークルームリストページの作成

APIを作ったらトークルームリストページを作りましょう。今回は「http://dev.siroisakana.com/ta/talk」というURLでアクセスできるようにします。すでに「talk」というディレクトリがあるはずなので今回はその中に「index.vue」というファイルを作成します。ファイル構成は以下のようになります。



作成したら以下の内容を入力して保存してください。

frontend/pages/talk/index.vue

```
<template>
  <section>
    <section>
      <sub-heading>検索</sub-heading>
```

```
      <div class="search">
        <div class="search-form">
          <div class="search-target">
            部屋名
          </div>
          <input type="text" v-model="form.title">
        </div>
        <div class="search-form">
          <div class="search-target">
            タグ
          </div>
          <input type="text" v-model="form.tags">
        </div>
        <div class="search-form">
          <label>お気に入り <input type="checkbox" v-model="form.fav"></label>
        </div>
      </div>
      <div class="button-wrapper">
        <nuxt-link :to="form | queries"><button class="button">検索</button></nuxt-link>
      </div>
    </section>
    <section>
      <sub-heading>トークルームリスト</sub-heading>
      <div class="button-wrapper">
        <nuxt-link to="/talk/create">
          <button class="button">トークルーム作成</button>
        </nuxt-link>
      </div>
      <div v-if="list.length">
        <room-list :rooms="list"/>
        <div class="pagelink-wrapper">
          <nuxt-link
            class="pagelink"
            v-if="Number($route.query.page)"
            :to="`${$route.path}${queryWithoutPage}&page=${Number($route.query.page) - 1}`">
            前のページへ
          </nuxt-link>
          <nuxt-link
            class="pagelink"
            v-if="isContinuing"
            :to="`${$route.path}${queryWithoutPage}&page=${Number($route.query.page || 0) + 1}`">
            次のページへ
          </nuxt-link>
        </div>
      </div>
      <div v-else>
        検索結果はありません。
      </div>
    </section>
  </section>
</template>

<script>
import SubHeading from '~/components/SubHeading.vue'
import RoomList   from '~/components/RoomList.vue'

export default {
```

```
  components: {
    SubHeading,
    RoomList
  },
  middleware: 'authenticated',
  watchQuery: true,
  head() {
    return {
      title: 'トークルームリスト'
    }
  },
  data() {
    return {
      form: {
        title: this.$route.query.title,
        tags:  this.$route.query.tags,
        fav:   this.$route.query.fav == 't'
      }
    };
  },
  asyncData: async function(context) {
    const queries  = context.route.fullPath.slice(context.route.path.length);
    const response = await context.$axios.get(`/api/rooms${queries}`);

    return {
      list:         response.data.list,
      isContinuing: response.data.isContinuing
    }
  },
  filters: {
    queries: (form) => {
      const queries = [];
      if (form.title) queries.push(`title=${encodeURIComponent(form.title)}`);
      if (form.tags)  queries.push(`tags=${encodeURIComponent(form.tags)}`);
      if (form.fav)   queries.push('fav=t');
      return queries.length ? '?' + queries.join('&') : '';
    }
  },
  computed: {
    queryWithoutPage: {
      get() {
        const queries = [];
        for (const prop in this.$route.query) {
          if (prop != 'page') {
            queries.push(`${prop}=${this.$route.query[prop]}`);
          }
        }
        return '?' + queries.join('&');
      }
    }
  }
};
</script>

<style lang="scss" scoped>
.search {
  display: flex;
```

```
  align-items: flex-end;

  .search-form {
    margin: 0 10px;
  }
}

.pagelink-wrapper {
  padding: 20px;
  display: flex;
  justify-content: space-around;

  .pagelink {
    display: inline-flex;
    justify-content: center;
    min-width: 120px;
    padding: 4px 12px;
    border: 1px solid #333;
    border-radius: 4px;
    text-decoration: none;
    font-weight: bold;
    color: #666;
  }
}
</style>
```

「http://dev.siroisakana.com/ta/talk」にアクセスすると以下のような感じになっているはずです。

このページではフォームに検索内容を入力できるようになっています。フォームはformという値からform.title、form.tagsなどのようにv-modelで紐付けられており、それらをfilterして検索ボタンのリンク先URLが変更されるようになっています。これにより検索ボタンから適切なURLパラメーターによるアクセスが行えるようになっています。

また、今のページが0ページ目ではなかったりトークルームリストが1ページあたりの最大数を超えていたりすると「前のページへ」「次のページへ」というボタンがそれぞれ出るようになっています。これはそれぞれ今のページが0

ページ目ではない、APIから帰ってきたisContinuingがtrueという条件で表示されるようになっており、算出プロパティにより作られたpage以外のURLパラメーターの内容と連結してリンク先が作られています。

4.9.13 トークルーム編集APIの作成

　次はトークルームのタイトルや説明文などを後から編集できるようにしましょう。ということでAPIを作っていきます。「/rooms/(key)/setting」にGETでアクセスすることで編集ページに必要な値を取得することができ、同じURLにPUTでアクセスすることで情報を更新できるようにします。「backend/server/api.js」を開き以下のAPIを追記して保存してください。

backend/server/api.js

```
router.get('/rooms/:key(\\d+|public\)/setting', checkAuthentication, async (req, res) => {
  try {
    const rno = Number(req.params.key) || 0;
    const room = await Room.findOne({rno: rno, deleted: false}, {
      _id:         0,
      title:       1,
      tags:        1,
      summary:     1,
      description: 1,
      character:   1
    });

    if (!room) { // 指定のトークルームが見つからなければ404を返し中断
      return res.status(400).send();
    } else if (room.character.toString() != req.user._id.toString()) { // トークルーム主ではない場合403を返し中断
      return res.status(403).send();
    }

    return res.status(200).send({
      title:       room.title,
      tags:        room.tags,
      summary:     room.summary,
      description: room.description
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});

router.put('/rooms/:key(\\d+|public\)/setting', checkAuthentication, checkCsrf, async (req, res) => {
  try {
    const rno = Number(req.params.key) || 0;
    const room = await Room.findOne({rno: rno, deleted: false}, {character: 1});

    if (!room) { // 指定のトークルームが見つからなければ404を返し中断
      return res.status(400).send();
    } else if (room.character.toString() != req.user._id.toString()) { // トークルーム主ではない場合403を返し中断
      return res.status(403).send();
    }
```

```
    await room.update({
      title:       req.body.title,
      tags:        req.body.tags,
      summary:     req.body.summary,
      description: styling.profile(req.body.description)
    });

    return res.status(200).send(room);
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

トークルーム主であるかなどの確認が行われていたりしますが基本的には必要な情報をデータベースから受け取って送信 / 受信してデータベースを更新するだけのシンプルなAPIです。

4.9.14 トークルーム削除APIの作成

トークルームを削除するAPIも必要になるでしょう。「/rooms/(key)」からトークルームを削除できるようにします。「backend/server/api.js」に以下のAPIを追記して保存してください。なお、パブリックトークルームを削除することはないはずなので他のトークルーム関連のAPIとは違いpublicはキーに含めないようにしてあります。

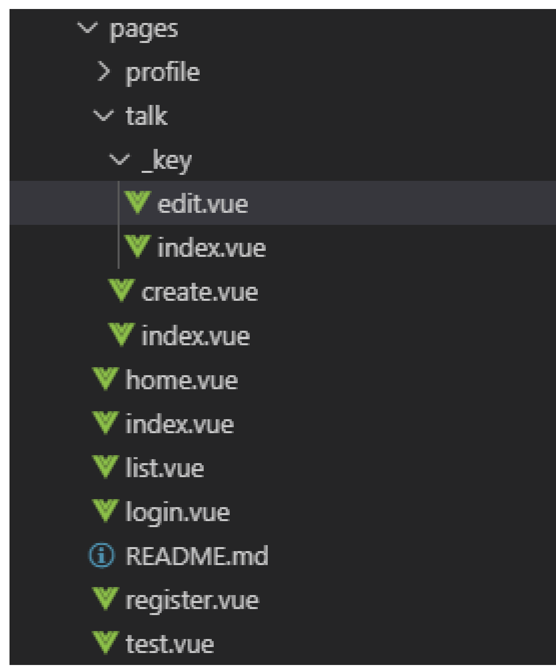
backend/server/api.js

```
router.delete('/rooms/:key(\\d+\)', checkAuthentication, checkCsrf, async (req, res) => {
  try {
    const rno = Number(req.params.key);

    if (!rno) { // rnoが無効であれば400を返し中断
      return res.status(400).send();
    }

    const room = await Room.findOne({rno: rno, deleted: false}, {character: 1});

    if (!room) { // 指定のトークルームが見つからなければ404を返し中断
      return res.status(400).send();
    } else if (room.character.toString() != req.user._id.toString()) { // トークルーム主ではない場合403を返し中断
      return res.status(403).send();
    }

    await room.update({deleted: true}); // 削除フラグをONに
    return res.status(200).send();
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

実行している内容はシンプルで、要求の内容に問題がなければ削除フラグをONにするだけのAPIになっています。

4.9.15 トークルーム編集ページ

次にトークルーム編集ページを作ります。「frontend/pages/talk/_key」内に「edit.vue」を作成してください。ファイル構成は以下のようになります。



作成したら以下の内容を記入して保存してください。当然トークルームの内容を編集できるようになっている他、このページからトークルームの削除を行えるようになっています。

frontend/pages/talk/_key/edit.vue

```
<template>
  <section>
    <sub-heading>トークルーム設定変更</sub-heading>
    <section class="form">
      <div class="form-title">タイトル ({{ title.length }} / {{ roomTitleMaxLength }} 文字) </div>
      <input class="form-input" type="text" v-model="title" placeholder="タイトル">
    </section>
    <section class="form">
      <div class="form-title">タグ</div>
      <div class="form-description">
        スペースで区切ることで複数指定できます。
      </div>
      <input class="form-input" type="text" v-model="joinedTags" placeholder="タグ">
    </section>
    <section class="form">
      <div class="form-title">サマリー</div>
      <div class="form-description">
        トークルームリストで表示される短い文章です。
      </div>
      <input class="form-input" type="text" v-model="summary" placeholder="サマリー">
    </section>
    <room-list :rooms="[{
      title:        title,
```

```
      summary:      summary,
      rno:          0,
      messageCount: 0,
      lastUpdate:   Date.now(),
      tags:         joinedTags ? joinedTags.split(/\s+/) : []
    }]"/>
    <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
    <div class="button-wrapper">
      <button class="button" @click="update">更新</button>
    </div>
    <section class="form">
      <div class="form-title">説明文</div>
      <div class="form-description">
        説明文はプロフィールと同様の書式で装飾することができます。<br>
        詳しくはルールブックを確認してください。
      </div>
      <textarea class="form-textarea" type="text" v-model="description" placeholder="説明文
"></textarea>
    </section>
    <section class="form">
      <div class="form-title">トークルーム削除</div>
      <div class="form-description">
        トークルームを削除する場合、下の入力欄に「delete」と入力してください。
      </div>
      <input class="form-input" type="text" v-model="deleteCheck">
    </section>
    <room-description
      mode="preview"
      :title="title"
      :description="description | styling" />
    <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
    <div class="button-wrapper">
      <button class="button" @click="update">更新</button>
    </div>
  </section>
</template>

<script>
import SubHeading      from '~/components/SubHeading.vue'
import RoomList        from '~/components/RoomList.vue'
import RoomDescription from '~/components/RoomDescription.vue'
import MessageBanner   from '~/components/MessageBanner.vue'

export default {
  components: {
    SubHeading,
    RoomList,
    RoomDescription,
    MessageBanner
  },
  middleware: 'authenticated',
  validate({params}) {
    return /^[1-9][0-9]*$/.test(params.key) || params.key == "public";
  },
  head() {
    return {
      title: 'トークルーム設定変更'
```

```
    };
  },
  data() {
    return {
      roomTitleMaxLength: process.env.roomTitleMaxLength,

      deleteCheck :    '',
      errorMessage:    '',
      waitingResponse: false
    }
  },
  asyncData: async function(context) {
    const response = await context.$axios.get(`/api/rooms/${context.params.key}/setting`);

    return {
      title:       response.data.title,
      summary:     response.data.summary,
      description: response.data.description,
      joinedTags:  response.data.tags.join(' ')
    }
  },
  filters: {
    styling: function(description) {
      if (!description) { // 変換するものがなければ空文字列を返す
        return '';
      }

      return (
        description
        // サニタイズ
          .replace(/&/g, '&amp;')
          .replace(/</g, '&lt;')
          .replace(/>/g, '&gt;')
          .replace(/"/g, '&quot;')
          .replace(/'/g, '&#39;')
        // タグ置換
        // サニタイズしてあるのでその前提で置換する
          .replace(/&lt;s&gt;(.+)&lt;\/s&gt;/sig , '<span class="small">$1</span>')
          .replace(/&lt;l&gt;(.+)&lt;\/l&gt;/sig , '<span class="large">$1</span>')
          .replace(/&lt;b&gt;(.+)&lt;\/b&gt;/sig , '<span class="bold">$1</span>')
        // 改行置換
          .replace(/\n/g, '<br>')
      );
    }
  },
  methods: {
    update: async function() {
      if (this.waitingResponse) {
        // 接続待ち状態であればアラートを表示しreturn、以降の処理を行わない
        return alert('しばらくお待ち下さい');
      }

      if (this.deleteCheck != "delete") {
        // トークルーム削除をしない場合
        // 入力内容の検証を行う、問題があればエラーメッセージを表示、returnして処理を中断
        if (!this.title) {
          return this.errorMessage = 'タイトルが入力されていません';
```

```
        }
        if (this.roomTitleMaxLength < this.title.length) {
          return this.errorMessage = 'タイトルが長すぎます';
        }

        // 問題がなければハッシュ化と接続に入る、接続待ち状態をON(true)に
        this.waitingResponse = true;

        try {
          // 更新を行う
          await this.$axios.put(`/api/rooms/${this.$route.params.key}/setting`, {
            csrf:        this.$store.getters['auth/loginCharacter'].csrf,
            title:       this.title,
            summary:     this.summary,
            description: this.description,
            tags:        this.joinedTags ? this.joinedTags.split(/\s+/) : []
          });

          // 更新したらそのトークルームのページへリダイレクト
          this.$router.push(`/talk/${this.$route.params.key}`);
        } catch (e) {
          // エラーが発生した場合エラーメッセージを表示して接続待ち状態をOFF(false)に
          this.errorMessage = '更新中にエラーが発生しました';
          this.waitingResponse = false;
        }
      } else {
        // トークルーム削除
        // 接続待ち状態をON(true)に
        this.waitingResponse = true;

        try {
          // 更新を行う
          await this.$axios.delete(`/api/rooms/${this.$route.params.key}`, {
            data: {
              csrf: this.$store.getters['auth/loginCharacter'].csrf,
            }
          });

          // 更新したらトークルームリストへリダイレクト
          this.$router.push(`/talk`);
        } catch (e) {
          // エラーが発生した場合エラーメッセージを表示して接続待ち状態をOFF(false)に
          this.errorMessage = '削除中にエラーが発生しました';
          this.waitingResponse = false;
        }
      }
    }
  }
}
</script>
```

処理の内容はほとんど他のAPIやページで見たようなものとほぼ一緒ですが、トークルーム削除(deleteCheck)に入力された値が「delete」であった場合トークルーム削除APIに、そうでなかった場合トークルーム編集APIにアクセスするようにしています。これ以外は今までのAPIやページで見てきた処理ばかりなので説明は割愛しま

す。

　これでトークルーム関連処理は終わりです。「トークルーム」「戦闘」という定期・APゲーム開発におけるアルゴリズムの難所の1つがこれで終わりました。お疲れさまです。

4.10 設定ページ

4.10.1 キャラクター削除APIの作成

次に設定ページを作ります。とはいえ今回開発する定期・APゲームでは特に環境設定などを作る予定はないので内容的にはキャラクター削除だけが行えるページになります。今回は「/characters/main」にDELETEリクエストを行うことで削除できるようにします。「backend/server/api.js」に以下のAPIを追記して保存しましょう。

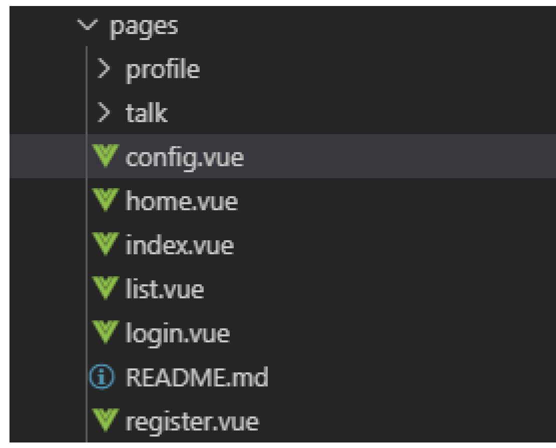
backend/server/api.js

```
router.delete('/characters/main', checkAuthentication, checkCsrf, async (req, res) => {
  try {
    await Character.findByIdAndUpdate(req.user._id, {
      deleted: true, // 削除フラグをONにし他キャラクターへの参照情報を全て削除
      fav:     [],
      block:   [],
      blocked: [],
      mute:    []
    });
    await Character.updateMany({}, {
      $pull: { // 他キャラクターのこのキャラクターへの参照情報を全て削除
        fav:     req.user._id,
        block:   req.user._id,
        blocked: req.user._id,
        mute:    req.user._id
      }
    });
    req.logout(); // ログアウトさせ
    return res.status(200).send(); // 200を返す
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

アクセスしたキャラクターの削除フラグをONにした上でそのキャラクターに関係する全てのお気に入り、ブロック、被ブロック、ミュートの情報を削除しています。削除されているはずのキャラクターへの参照情報があるとバグの原因になるためです。削除したのにログインし続けられるというのもバグの原因になるため、削除が完了したらログアウトさせています。

4.10.2 設定ページの作成

それでは設定ページもといキャラクター削除ページを作りましょう。「frontend/pages」内に「config.vue」ファイルを作成します。ファイル構成は以下のようになります。



作成したら以下の内容を記述して保存してください。

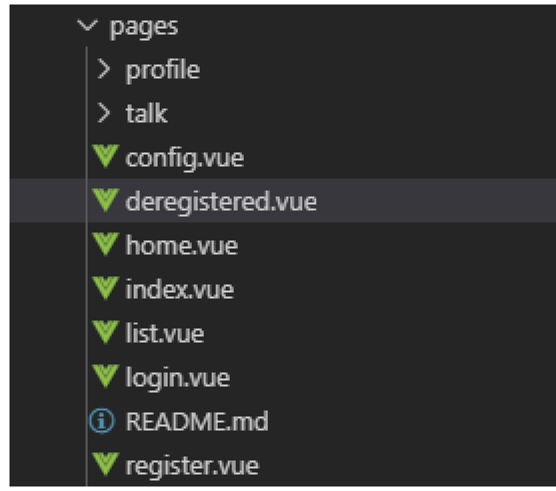
frontend/pages/config.vue

```
<template>
  <section>
    <sub-heading>その他設定</sub-heading>
    <section class="form">
      <div class="form-title">キャラクター削除</div>
      <div class="form-description">
        キャラクターを削除する場合、下の入力欄に「delete」と入力してください。
      </div>
      <input class="form-input" type="text" v-model="deleteCheck">
    </section>
    <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
    <div class="button-wrapper">
      <button class="button" @click="update">更新</button>
    </div>
  </section>
</template>

<script>
import SubHeading    from '~/components/SubHeading.vue'
import MessageBanner from '~/components/MessageBanner.vue'

export default {
  components: {
    SubHeading,
    MessageBanner
  },
  middleware: 'authenticated',
  head() {
    return {
      title: 'その他設定'
    };
  },
  data() {
    return {
      deleteCheck :    '',
      errorMessage:    '',
      waitingResponse: false
```

```
    }
  },
  methods: {
    update: async function() {
      if (this.waitingResponse) {
        // 接続待ち状態であればアラートを表示しreturn、以降の処理を行わない
        return alert('しばらくお待ち下さい');
      }

      if (this.deleteCheck == "delete") {
        // "delete"が入力されていたらキャラクター削除へ
        // 接続待ち状態をON(true)に
        this.waitingResponse = true;

        try {
          // キャラクター削除を行う
          // ここでサーバー側はログアウト状態になる
          await this.$axios.delete('/api/characters/main', {
            data: {
              csrf: this.$store.getters['auth/loginCharacter'].csrf,
            }
          });

          // ログイン状態を問い合わせ
          // ログアウトしているはずなのでこれでクライアント側もログアウト状態になる
          await this.$store.dispatch('auth/login');

          // 削除後、削除完了ページへリダイレクト
          this.$router.push(`/deregistered`);
        } catch (e) {
          // エラーが発生した場合エラーメッセージを表示して接続待ち状態をOFF(false)に
          this.errorMessage = '削除中にエラーが発生しました';
          this.waitingResponse = false;
        }
      }
    }
  }
}
</script>
```

「delete」が入力されていたら削除APIにアクセスしています。API側の処理によって削除時にログアウトされるようになっているのでVuexを呼び出してクライアント側にもログアウト状態を反映させています。削除時にただトップページに飛ばされるだけなのも味気ないので、ここでは「/deregistered」にリダイレクトするようにしてあります。

ということで次は削除完了ページを作りましょう。「frontend/pages」内に「deregistered.vue」を作成してください。ファイル構成は以下のようになります。



作成したら以下の内容を入力して保存してください。メッセージを表示しているだけのページになります。

frontend/pages/deregistered.vue

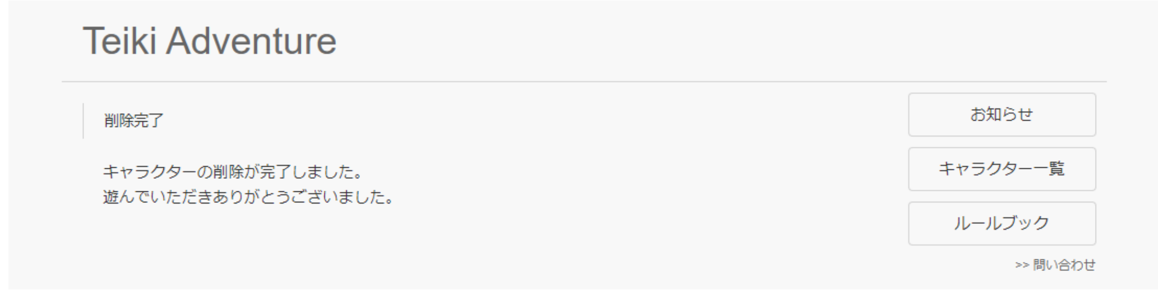
```
<template>
  <section>
    <sub-heading>削除完了</sub-heading>
    <p>
      キャラクターの削除が完了しました。<br>
      遊んでいただきありがとうございました。
    </p>
  </section>
</template>

<script>
import SubHeading from '~/components/SubHeading.vue'

export default {
  components: {
    SubHeading,
  }
}
</script>

<style lang="scss" scoped>
p {
  margin: 0 20px;
}
</style>
```

最後に右メニューにその他設定へのリンクを作っておきましょう。「frontend/components/SideMenu.vue」を開き、mini-link-wrapperの部分を以下のように書き換え保存してください。ログイン中は「>> その他設定」というメニューが小さく出るようになります。

```
frontend/components/SideMenu.vue
（省略）

  <div class="mini-link-wrapper">
    <nuxt-link v-if="auth" class="mini-link" to="/config">&gt;&gt; その他設定</nuxt-link>
    <nuxt-link class="mini-link" to="/inquiry">&gt;&gt; 問い合わせ</nuxt-link>
    <a v-if="auth" class="mini-link" :href="`${$router.options.base}api/logout`">&gt;&gt; ログアウト</a>
  </div>

（省略）
```

それでは実際にキャラクター削除をテストしましょう。削除するために適当なキャラクターを登録してください。「キャラクター一覧」などでそのキャラクターが登録されていることも一応確認しおきましょう。確認したら「その他設定」を開いてください。以下のようなページが表示されるので「delete」と入力して「更新」を押します。

そうするとログアウトされて以下のようなページに移動します。これでキャラクターが削除されました。



「キャラクター一覧」から消えておりログインもできなくなっていることを確認しておきましょう。

4.11　戦闘・基礎編

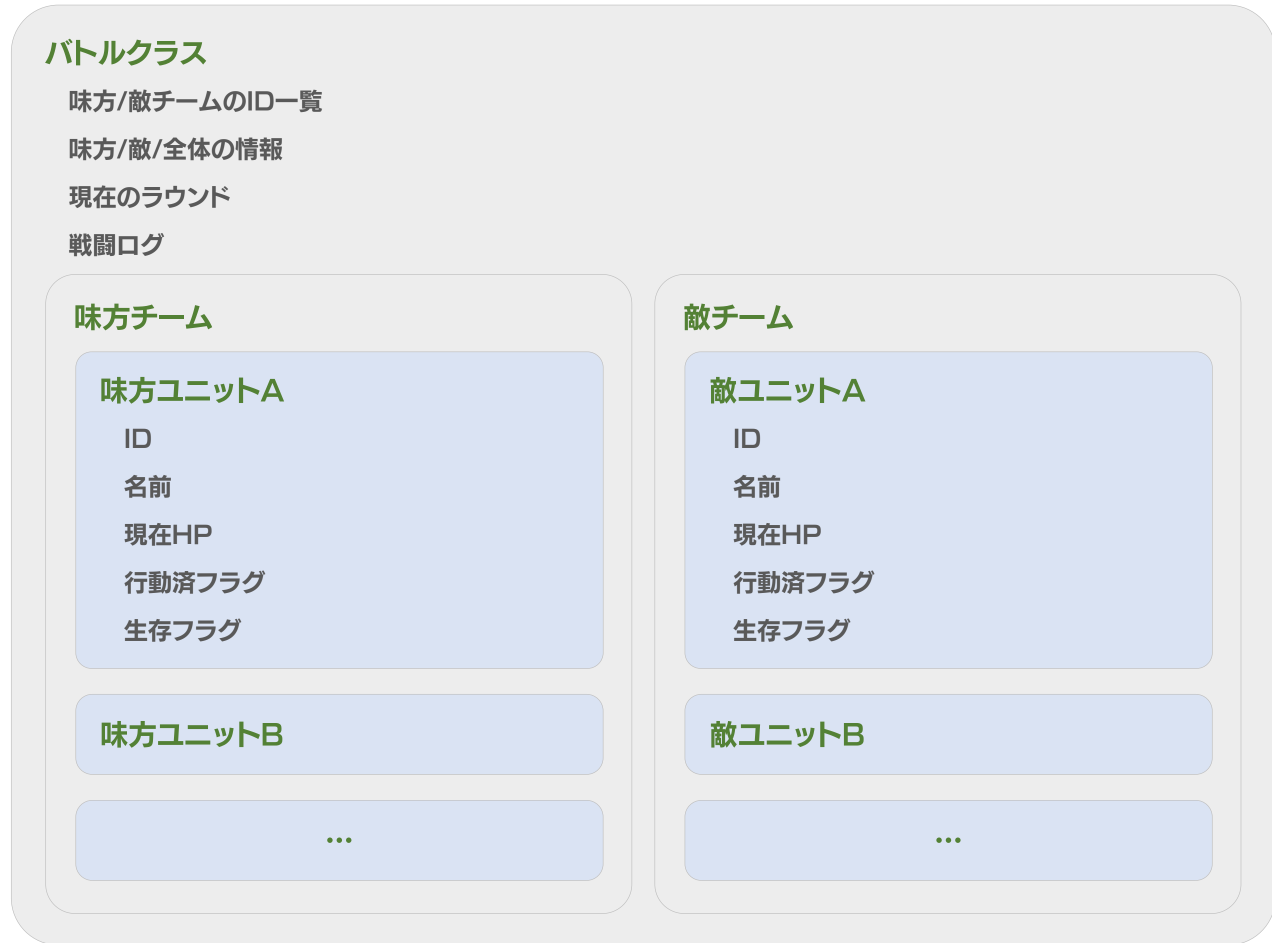
4.11.1 戦闘の仕様の決定

それでは次は戦闘システムを作っていきましょう。戦闘システムは複雑なのでまずは通常攻撃だけが行えるシンプルなシステムを作ってから最終的にいろいろなスキルのある戦闘システムへと徐々にステップアップしていきます。というわけで『戦闘・基礎編』では通常攻撃だけができる戦闘システムを作っていきます。具体的には以下のような仕様になります。

・味方チームと敵チームの2つに分かれて戦闘を行う。
・それぞれのキャラクターには「名前」があり、「HP」というステータスを持っている。
・ラウンドと呼ばれる区切りごとにキャラクターが1回ずつ行動を行う。これをターンと呼び、各キャラクターはターンごとに通常攻撃のみを行う。
・通常攻撃が行われると生きている敵のうちランダムな1体に10〜20点のダメージを与えHPを減らす。
・ラウンド終了時にHPが0以下になっているキャラクターは戦闘不能になり以降の戦闘に参加できなくなる。
・最終的に味方チームだけが生き残れば勝利、敵チームだけが生き残れば敗北。最大ラウンドが経過したときに決着がついていない場合は引き分けとなる。

4.11.2 戦闘アルゴリズム・基礎編

戦闘アルゴリズムの作り方は十人十色でどれが正解、といったものはありません。これから解説する内容は戦闘を表現する方法の1つと思っておいてください。それではこの戦闘システムをどうやって作ったらいいかを考えましょう。今回はクラスベースで戦闘を表現していきます。1キャラクターごとにそれぞれクラスのインスタンスを作り、インスタンスのメッセージのやりとりで戦闘を表現していきます。キャラクタークラスの他にも戦闘を取りまとめるバトルクラスも必要になるでしょう。図式化すると以下のようになります。



この図ではそれぞれのクラスの構造と管理する情報を記載しており、それぞれのクラスの領分については色分けして記載してあります。それぞれ解説していきます。

ユニットクラス

このクラスではそれぞれのキャラクターに依存する情報を管理しています。「名前」「現在HP」の情報を持っている他に「すでにこのラウンド行動しているか?」「戦闘不能になっているか?」という情報を持っています。また、バトルクラスで管理しやすくするためにIDという他のキャラクターとかぶらない値を割り振られています。

バトルクラス

このクラスは特定のキャラクターに依存しない戦闘全体の情報を管理しています。「現在のラウンド」「戦闘ログ」がそれにあたります。またこのクラスではそれぞれのユニットクラスの管理を行います。それぞれのユニットクラスに一意のIDを割り振り、それぞれのユニットとIDの情報を保管しています。このクラスは戦闘の進行を担う役割も持ちます。現在のラウンド、それぞれのユニットの行動状況、生存状況などを判断してラウンドを進行させたりキャラクターに行動を行わせたりします。

戦闘は基本的にこの2つのクラスを組み合わせることで表現します。具体的な処理手順を書くと以下のようになるでしょう。

1. 初期化処理

味方・敵それぞれのユニットを受け取り、IDを割り振って初期化する。

2. 戦闘開始

戦闘開始。ラウンド開始へ。

2.1 ラウンド開始

現在ラウンド数を+1する。

2.1.1 行動可能キャラチェック

行動可能なキャラを列挙する。
行動可能なキャラがいれば『2.1.2 ターン開始』に進む。
行動可能なキャラがいなければラウンド終了、『2.2 ラウンド終了』へ。

2.1.2 ターン開始

行動可能なキャラのうち1体のターンを開始する。

2.1.3 行動

そのキャラクターを行動させる。まだ生きている敵を取得して1体を選びHPを10〜20減らす。
行動したキャラクターは行動済みフラグをONに。

2.1.4 ターン終了

キャラクターのターン終了。『2.1.1 行動可能なキャラクターのチェック』へ戻る。

2.2 ラウンド終了

HPが0以下になっているキャラクターを戦闘不能状態にする。

2.3 戦闘判定

戦闘結果の判定を行う。
味方だけが生き残っていれば「勝ち」。敵だけが生き残っていれば「負け」。両方生き残っているのに現在ラウンドが最大ラウンドに到達しているか、敵味方全員戦闘不能になっていれば「引き分け」となる。『3. 戦闘終了』へ進む。
そのどれでもない場合はまだ戦闘を継続する。『2.1 ラウンド開始』に戻る。

3. 戦闘終了

戦闘終了。戦闘結果としてログを返す。

4.11.3 戦闘プログラムの作成

それでは実際に戦闘プログラムを作っていきましょう。まずは設定ファイルに戦闘の最大ラウンドを記載しておきます。「backend/config/default.json」を開き以下の設定を追記して保存してください。

```
backend/config/default.json
"battleMaxRound": 10
```

設定を保存したら「backend/server」内に「battle.js」を作成してください。ファイル構成は以下のようになります。

```
server
  api.js
  battle.js
  index.js
  initialize.js
  model.js
  passport.js
  styling.js
```

作成したら以下の内容を記述して保存してください。デバッグをやりやすくするためにこのプログラムは単一でアクセスを受け付けるようにしてあります。実行すると「http://dev.siroisakana.com/ta/api/battle」からこのプログラムにアクセスできるようになります。

```
backend/server/battle.js
const express = require('express');
const config  = require('config');
const app     = express();
const port    = config.port;

/*---------------------------------------------
  クラス
---------------------------------------------*/

/*---------------------------------------------
  API宣言
---------------------------------------------*/

app.get(`${config.directoryUrl}battle`, (req, res) => {
  return res.status(200).send('Battle API');
});

app.listen(port);
```

このプログラムを「npm run battletest」から実行できるようにしましょう。「backend/package.json」を開き"scripts"内を以下のように書き換えて保存してください。

backend/package.json

```
（省略）

"scripts": {
  "dev": "nodemon server/index.js --watch config --watch server",
  "start": "node server/index.js",
  "init": "node server/initialize.js",
  "battletest": "nodemon server/battle.js --watch config --watch server"
},

（省略）
```

それでは実行してみます。バックエンド側のプログラムをCtrl+Cで終了させて「npm run battletest」を実行し、「http://dev.siroisakana.com/ta/api/battle」にアクセスしてみてください。以下のように表示されればOKです。

Battle API

4.11.4 ユニットクラスの作成

それではクラスを作っていきます。まずはユニットクラスを作りましょう。「backend/server/battle.js」のコメントでクラスと書いてある部分の次に以下の内容を記述して保存してください。

backend/server/battle.js

```
（省略）

/*-----------------------------------------------
  クラス
-----------------------------------------------*/

class Unit {
  constructor(name, hp) { // 初期化処理１ 名前とHPを受け取り、行動済ステータスと生存ステータスもセット
    this.name   = name;
    this.hp     = hp;
    this.acted  = false;
    this.living = true;
  }

  init(id, battle) { // 初期化処理２ ユニット生成時には受け取れない値をここでセット
    this.id     = id;
    this.battle = battle;
  }

  isLiving() { // 生きているかどうか
    return this.living;
  }

  isActable() { // 行動可能かどうか、生きていて行動済ではない
    return this.isLiving() && !this.acted;
  }
```

```
  gainAttack() { // 攻撃を受ける
    const damage = Math.floor(Math.random() * 11) + 10; // 10 ～ 20点のダメージを受ける
    this.hp -= damage;
    this.battle.log(`　${this.name}は${damage}点のダメージを受けた！(現在HP：${this.hp})`);
  }

  act() { // 通常攻撃を行う
    this.acted = true;
    this.battle.log(`${this.name}の攻撃！`); // メッセージを表示

    const targets = this.battle.getLivingEnemies(this.id); // 生きている（行動者にとっての）敵たちを取得
    const target  = targets[Math.floor(Math.random() * targets.length)] // その中から1人を選び
    target.gainAttack(); // 攻撃する（攻撃を受ける処理を呼び出す）
  }

  setup() { // セットアップ処理 行動済ステータスを未行動（false）に
    this.acted = false;
  }

  cleanup() {
    if (this.living && this.hp <= 0) { // クリーンアップ時生きていてHP0以下なら戦闘不能に
      this.living = false;
      this.battle.log(`　${this.name}は倒れた……`);
    }
  }
}

（省略）
```

処理を見ていきましょう。コンストラクタでキャラクターの名前とHPを受け取って設定している他、行動済ステータスと生存ステータスも初期化しています。また、initという初期化メソッドも用意しています。これはバトルクラスからIDとバトルクラス自身を受け取る処理になります。これによりバトルクラスへのアクセスが可能になります。

isLiving、isActableはそれぞれ生きているかどうか、行動可能かどうかをそれぞれ返すメソッドになっています。isActableは生きていて行動済みではなければtrue、そうでなければfalseが返るようになっています。この情報を元にしてバトルクラスは戦闘処理を行っていきます。

gainAttackは攻撃を受ける処理です。ダメージ量は10 ～ 20点になるようになっていて、ダメージ量に応じてHPが減るようになっています。また、どれぐらいのダメージを受けたかログを出力するようにしています。

actは行動処理です。行動済ステータスをtrueに設定し、「(名前)の攻撃！」というログを出力しています。その後バトルクラスからまだ生きている敵の一覧を受け取ってその中の一体を選び、gainAttackを呼び出します。

setupはラウンド開始の処理になります。行動済みステータスをfalseに設定し再び行動ができるようにしてあります。

cleanupはラウンド終了時の処理になります。ラウンド終了時に生存ステータスがtrueかつHPが0以下になっている場合、生存ステータスをfalseにして「(名前)は倒れた……」というログが出力されるようにしています。

4.11.5 バトルクラスの作成

次はバトルクラスを作りましょう。「backend/server/battle.js」のユニットクラスの次に以下の内容を追記して保

存してください。

```
backend/server/battle.js

  （省略）

class Battle {
  constructor(allies, enemies) {
    this.allies  = allies;
    this.enemies = enemies;
    this.units   = allies.concat(enemies); // 敵味方全体

    this.allyIds  = [];
    this.enemyIds = [];

    this.unitIdDealer = 0;
    for (let i = 0; i < allies.length; i++) { // 味方チームに一意のIDを割り振る
      this.allies[i].init(this.unitIdDealer, this);
      this.allyIds.push(this.unitIdDealer);
      this.unitIdDealer++;
    }
    for (let i = 0; i < enemies.length; i++) { // 敵チームに一意のIDを割り振る
      this.enemies[i].init(this.unitIdDealer, this);
      this.enemyIds.push(this.unitIdDealer);
      this.unitIdDealer++;
    }

    this.round     = 0;
    this.battleLog = '';
  }

  log(str) { // バトルログを追加
    this.battleLog += this.battleLog ? '<br>' + str : str;
  }

  extractLiving(units) {
    // 受け取ったユニット配列から生きているユニットだけを抽出して返す
    const livingUnits = [];

    for (let i = 0; i < units.length ;i++) {
      if (units[i].isLiving()) {
        livingUnits.push(units[i]);
      }
    }

    return livingUnits;
  }

  extractActable(units) {
    // 受け取ったユニット配列から行動可能なユニットだけを抽出して返す
    const actableUnits = [];

    for (let i = 0; i < units.length ;i++) {
      if (units[i].isActable()) {
        actableUnits.push(units[i]);
      }
```

```
    }

    return actableUnits;
  }

  getEnemies(unitId) {
    // IDのユニットから見た敵側チームを取得
    return this.allyIds.includes(unitId) ? this.enemies : this.allies;
  }

  getLivingEnemies(unitId) {
    // IDのユニットから見た生きている敵側チームを取得
    return this.extractLiving(this.getEnemies(unitId));
  }

  judge() {
    const livingAllies  = this.extractLiving(this.allies); // 生き残っている味方チームを取得
    const livingEnemies = this.extractLiving(this.enemies); // 生き残っている敵チームを取得

    if (livingAllies.length && !livingEnemies.length) {
      // 味方が生き残っていて敵が全員戦闘不能 → 勝利
      return 'win';
    } else if (!livingAllies.length && livingEnemies.length) {
      // 味方が全員戦闘不能で敵が生き残っている → 敗北
      return 'lose';
    } else if (!livingAllies.length && !livingEnemies.length) {
      // 味方も敵も全員戦闘不能 → 引き分け
      return 'even';
    } else if (config.battleMaxRound <= this.round) {
      // どちらも生き残っているが現在ラウンドが最大ラウンド以上 → 引き分け
      return 'even';
    } else {
      // そのどれでもない → 戦闘継続
      return 'continue';
    }
  }

  fight() {
    this.log('戦闘を開始した！');

    round_loop:
    while(true) {
      this.round++;
      this.log("------------------------------------");
      this.log("" + this.round + "ラウンド目！");

      this.log("");

      for (let i = 0; i < this.units.length; i++) { // セットアップ処理とステータス表示
        this.units[i].setup();
        this.log(
          (this.units[i].isLiving() ? '◆' : '◇') +
          this.units[i].name +
          '  HP:' +
          this.units[i].hp
        );
      }
```

```
      this.log("");

      turn_loop:
      while(true) {
        // 全ユニットから行動可能なユニットを抽出
        const actableUnits = this.extractActable(this.units);

        if (actableUnits.length) {       // 行動可能な者がいれば
          const actor = actableUnits[0]; // 行動可能な者たちの中から行動者を選び
          actor.act();                   // 行動実行
        } else {                         // 行動可能な者がいなければ
          break turn_loop;               // そのラウンドは終了処理へ
        }
      }

      this.log("");

      for (let i = 0; i < this.units.length; i++) { // クリーンアップ処理
        this.units[i].cleanup();
      }

      const judge = this.judge(); // 戦闘判定
      if (judge != 'continue') {
        // 判定結果が戦闘継続でない場合戦闘終了
        this.log("");
        this.log('戦闘終了！');

        if (judge == 'win') { // ジャッジ結果が勝利のとき
          this.log('勝利！');
        } else if (judge == 'lose') { // ジャッジ結果が敗北のとき
          this.log('敗北……');
        } else { // そのどちらでもない（引き分け）とき
          this.log('引き分け……');
        }

        break round_loop; // 戦闘ループから抜ける
      }
      // 判定結果が戦闘継続かつ最大ラウンドに到達していない場合次のラウンドへ
    }

    // 戦闘ログを返す
    return this.battleLog;
  }
}

（省略）
```

ちょっと複雑に見えますが分けてしまえば簡単です。少しずつ見ていきましょう。まずはコンストラクタの部分です。ここでは味方チームのユニットの配列と敵チームのユニットの配列を受け取っています。受け取った値はそのまま味方チーム/敵チームとして保持している他、this.unitsで敵味方全体も参照できるようにしてあります。

次にそれぞれのユニットへのIDの割り振りと初期化を行っています。割り振ったIDはthis.allyIds、this.enemyIdsにそれぞれ保管しています。また、初期化ではユニットクラスのinitからIDとバトルクラス自身を渡すようにしてあ

ります。これでユニットクラスの初期化が完了です。

その他にも現在ラウンドと戦闘ログを初期化しています。現在ラウンドは0、戦闘ログは""が初期値です。ラウンド開始時に現在ラウンドは+1されるため初期値は0にするようにしてあります。

次はメソッドを見ていきましょう。logは受け取った値をログとして蓄積するためのメソッドになります。それが最初の行であれば改行は行わず、2行目以降であれば改行をするようになっていますが基本的には受け取ったログを連結しながら蓄積していくだけのメソッドになります。

extractLiving、extractActableはユニット配列を受け取って生きているユニット、行動可能なユニットをそれぞれ抽出するためのメソッドです。簡単に絞り込めると便利なのでメソッド化してそれらの操作をやりやすくしています。

getEnemiesはIDのキャラクターから見た敵チームを取得するメソッドです。味方チームのキャラクターのIDでこれを呼び出すと敵チームが、敵チームのキャラクターのIDでこれを呼び出すと味方チームが帰ってきます。getLivingEnemiesはその中から生きている敵のみを絞り込んで返しています。内部的にはgetEnemiesを呼び出してからextractLivingで生きているもののみを抽出して返しています。

judgeは戦闘判定を行うための関数です。味方・敵チームの生存状況や最大ラウンドと現在ラウンドの兼ね合いを見て勝利(win)、敗北(lose)、引き分け(even)、戦闘継続(continue)の4つの値のうちどれかを返します。

これらは処理をやりやすくするためのユーティリティメソッドになります。実際の戦闘はfightメソッドから行います。戦闘を開始したら「戦闘を開始した！」というログを出力します。

戦闘開始したらラウンド処理のループに入っていきます。ラウンド処理ではまず現在ラウンドを+1して何ラウンド目かの表示を行い、それぞれのユニットのセットアップ処理を呼び出します。このときそのユニットが生きているかに応じて「◆」「◇」を出し分けつつ名前とHPをログに主力しています。

次にターン処理のループに入っていきます。extractActableを使って行動可能なキャラを取得し、行動可能なキャラがいればそのうち1体を選んでactを呼び出して行動を実行させます。行動が終わったらターン処理の最初に戻り、行動可能なキャラクターがいなくなるまでこれを繰り返します。

行動可能なキャラクターがいなくなればそのラウンドは終了になります。judgeを呼び出して戦闘判定を行い、戦闘継続(continue)が帰ってくれば次のラウンドへ、そうでなければ結果に応じたログを出力して戦闘を終了します。戦闘終了したらいままで蓄積してきたログを返して処理を終了します。

以上がバトルクラスの全てです。これからいろいろ要素が増えて少しずつ複雑になっていきますが、基本形はずっとこのままです。これから作っていく戦闘の土台になるものなので全体としてどうなっているのかしっかりと把握するようにしてください。

4.11.6 戦闘の実行

それでは戦闘を実行しましょう。ユニットクラスとバトルクラスをそれぞれ作り、戦闘を行わせてその結果をAPIから返すようにします。「backend/server/battle.js」のAPI宣言と書かれているところの下を以下のように書き換えて保存してください。ここではHPが100ある味方3体とHPが60ある敵4体をそれぞれユニットクラスから作成し、バトルクラスに渡してバトルクラスを作成しています。作成されたバトルクラスは内部的に初期化が終わっているのであとはbattle.fight();を呼び出して戦闘を行わせ、戦闘結果を受け取ってAPIから返しています。

```
backend/server/battle.js
/*---------------------------------------------
  API宣言
---------------------------------------------*/

app.get(`${config.directoryUrl}battle`, (req, res) => {
  const battle = new Battle([
    new Unit('味方A', 100),
    new Unit('味方B', 100),
    new Unit('味方C', 100)
  ], [
    new Unit('敵A', 60),
    new Unit('敵B', 60),
    new Unit('敵C', 60),
    new Unit('敵D', 60)
  ])

  const result = battle.fight();

  return res.status(200).send(result);
});

app.listen(port);
```

それでは「http://dev.siroisakana.com/ta/api/battle」にアクセスしてみてください。以下のように戦闘が行われているはずです(長いので一部のみ)。なおアクセスごとに戦闘を行っているのでリロードすると結果が変わります。

```
戦闘を開始した！
-------------------------------------
1ラウンド目！

◆味方A　HP:100
◆味方B　HP:100
◆味方C　HP:100
◆敵A　HP:60
◆敵B　HP:60
◆敵C　HP:60
◆敵D　HP:60

味方Aの攻撃！
  敵Aは12点のダメージを受けた！(現在HP：48)
味方Bの攻撃！
  敵Cは17点のダメージを受けた！(現在HP：43)
味方Cの攻撃！
  敵Cは16点のダメージを受けた！(現在HP：27)
```

4.12 戦闘・応用編

4.12.1 ATK・AGIの組み込み

基礎が終わったらいろいろなステータスやスキルのある実践的なシステムを作っていきましょう。ステータスがHPしかないというのは寂しいので、まずはATK・AGIのステータスを組み込んでいきます。それぞれ以下のような効果を持つことにします。

ATK	攻撃力。攻撃によって与えるダメージは10～10+ATK点。
AGI	素早さ。高いほど行動順が早くなる他、AGI%の確率で行動時にもう1回攻撃できる。

それでは効果を組み込んでいきましょう。「backend/server/battle.js」を開いてください。まずはユニットクラスを書き換えていきます。まずはコンストラクタで「名前」「HP」の他に「ATK」「AGI」を受け取るようにします。ユニットクラスのconstructorを以下のように書き換えてください。

```
backend/server/battle.js
（省略）

  constructor(name, hp, atk, agi) { // 初期化処理1 名前とステータスを受け取り、行動済ステータスと生存ステータスもセット
    this.name   = name;
    this.hp     = hp;
    this.atk    = atk;
    this.agi    = agi;
    this.acted  = false;
    this.living = true;
  }

（省略）
```

次にgainAttack、actの処理に手を入れます。ユニットクラスのgainAttack、actメソッドのところを以下の3つのメソッドに書き換えてください。actメソッドではAGIのシステムによりAGI%の確率で2回攻撃する可能性があるようになっており、それに伴い1回の行動を行う処理はactionメソッドに分割してあります。

gainAttackではダメージ量が10～10+攻撃者のATK点になるように変更しています。このときgainAttack側で攻撃者の情報が必要になるためgainAttackは攻撃者の情報を受け取るように変更しており、呼び出す側も自身を攻撃者として渡すように変更しています。

```
backend/server/battle.js
（省略）

  gainAttack(attacker) { // 攻撃を受ける
    const damage = Math.floor(Math.random() * (attacker.atk + 1)) + 10;
    // 10 ～ 攻撃者のATK+10点のダメージを受ける
    this.hp -= damage;
    this.battle.log(`　${this.name}は${damage}点のダメージを受けた！(現在HP：${this.hp})`);
```

```
  }

  action() { // 1回の行動を行う
    this.battle.log(`${this.name}の攻撃！`); // メッセージを表示
    const targets = this.battle.getLivingEnemies(this.id); // 生きている（行動者にとっての）敵たちを取得
    const target  = targets[Math.floor(Math.random() * targets.length)] // その中から1人を選び
    target.gainAttack(this); // 攻撃する（攻撃を受ける処理を呼び出す）
  }

  act() { // ターン行動を行う
    this.action();
    if (Math.random() * 100 < this.agi) { // AGI%の確率で連続行動
      this.battle.log(`連続行動！`);
      this.action();
    }
    this.acted = true; // 行動済みフラグをONに
  }

（省略）
```

次にバトルクラスを書き換えましょう。まずはバトルクラスに以下のメソッドを追加してください。適切な場所であれば追加する場所はどこでもいいです。extractActableの次のところなどに追加しておいてください。sortUnitsByAgiは受け取ったユニット配列をAGI降順に並べ替えて並べ替えた後の配列を返すメソッドになっています。

backend/server/battle.js

```
（省略）

  sortUnitsByAgi(units) {
    // 受け取ったユニット配列をAGI降順にソート
    units.sort((a, b) => {
      return b.agi - a.agi;
    });

    return units;
  }

（省略）
```

次にバトルクラスのturn_loop:となっているところを以下のように書き換えましょう。行動可能なユニットの配列を先程作ったsortUnitsByAgi関数で並び替え、もっとも速い者、つまり配列0番目のユニットを行動者としています。

backend/server/api.js

```
（省略）

    turn_loop:
    while(true) {
      // 全ユニットから行動可能なユニットを抽出
      const actableUnits = this.extractActable(this.units);

      if (actableUnits.length) { // 行動可能な者がいれば
```

```
        const actor = this.sortUnitsByAgi(actableUnits)[0];
        // それをAGI降順に並べ替え最もAGIが速いものを行動者とし
        actor.act();      // 行動実行
      } else {            // 行動可能な者がいなければ
        break turn_loop; // そのラウンドは終了処理へ
      }
    }

  (省略)
```

最後にAPI宣言部分を以下のように書き換えてください。ユニットクラスが要求する値が変わっているので味方や敵に新たなステータスが設定されています。

backend/server/battle.js

```
  (省略)

/*----------------------------------------------
  API宣言
----------------------------------------------*/

app.get(`${config.directoryUrl}battle`, (req, res) => {
  const battle = new Battle([
    new Unit('味方A', 250, 40, 10),
    new Unit('味方B', 250, 40, 10),
    new Unit('味方C', 250, 40, 10)
  ], [
    new Unit('速い敵A', 120, 20, 25),
    new Unit('速い敵B', 120, 20, 25),
    new Unit('力強い敵A', 150, 60, 0),
    new Unit('力強い敵B', 150, 60, 0)
  ])

  const result = battle.fight();

  return res.status(200).send(result);
});

app.listen(port);
```

ここまで記述したら保存して再度「http://dev.siroisakana.com/ta/api/battle」にアクセスしてみましょう。以下のような感じになるはずです(長いので一部のみ)。ダメージ量が変わっていたり行動順が「速い敵」「味方」「力強い敵」の順番になるようになっていたり連続行動が発生するようになっていたりしてATKやAGIのシステムが反映されているのがわかるはずです。

```
戦闘を開始した！
------------------------------------
1ラウンド目！

◆味方A　HP:250
◆味方B　HP:250
◆味方C　HP:250
◆速い敵A　HP:120
◆速い敵B　HP:120
◆力強い敵A　HP:150
◆力強い敵B　HP:150

速い敵Aの攻撃！
  味方Aは17点のダメージを受けた！(現在HP：233)
速い敵Bの攻撃！
  味方Bは30点のダメージを受けた！(現在HP：220)
連続行動！
速い敵Bの攻撃！
  味方Cは16点のダメージを受けた！(現在HP：234)
味方Aの攻撃！
  力強い敵Aは18点のダメージを受けた！(現在HP：132)
味方Bの攻撃！
  力強い敵Bは10点のダメージを受けた！(現在HP：140)
味方Cの攻撃！
  速い敵Aは28点のダメージを受けた！(現在HP：92)
力強い敵Aの攻撃！
  味方Cは42点のダメージを受けた！(現在HP：192)
力強い敵Bの攻撃！
  味方Bは14点のダメージを受けた！(現在HP：206)
```

4.12.2 ヘイトシステム

次はヘイトのシステムを作ります。ヘイトとはどの敵をどれだけ優先的に狙うかを数値化したシステムのことで、ここでは受けたダメージの分だけ攻撃してきた敵にヘイトを持つことにし、それぞれのユニットはそれぞれにとって最もヘイトが高い敵に攻撃をするようにします。ヘイトが同値の場合はランダムな敵を攻撃するようにします。

それでは実際に組んでいきましょう。ユニットクラスのconstructorを以下のように書き換えてください。ヘイト情報を持てるように初期化を行うようにしています。ヘイト情報は配列に持つことにし、配列のキーにヘイトを持つ対象のIDを使い、そのキーの値にヘイト量を設定するようにします。

backend/server/battle.js

```
（省略）

  constructor(name, hp, atk, agi) { // 初期化処理1 名前とステータスを受け取り、行動済ステータスと生存ステータスもセット
    this.name    = name;
    this.hp      = hp;
    this.atk     = atk;
    this.agi     = agi;

    this.acted  = false;
    this.living = true;
    this.hates  = []; // ヘイト量
  }
```

```
（省略）
```

それではヘイトシステムを組み込んでいきましょう。まずはキャラクタークラスの適当なところ（適切な場所であればどこでも構いません、isActableの次など）に以下の2つのメソッドを追加してください。hateはヘイト量とターゲットを受け取りターゲットへのヘイトを蓄積するメソッド、sortUnitsByHateはユニット配列を受け取ってヘイトの高い順に並び替えるメソッドです。this.hates[id]は初期値がundefinedであることに注意してください。

backend/server/battle.js

```
（省略）

hate(hate, target) { // 対象へのヘイトを蓄積する
  this.hates[target.id] = (this.hates[target.id] || 0) + hate;
}

sortUnitsByHate(units) {
  // 受け取ったユニット配列をヘイト値降順にソート
  units.sort((a, b) => {
    const hateA = this.hates[a.id] || 0; // undefinedだったら0扱い
    const hateB = this.hates[b.id] || 0;
    return hateB - hateA;
  });

  return units;
}

（省略）
```

次にバトルクラスの適当なところ（適当な場所であればどこでも構いません、sortUnitsByAgiの次など）に以下のメソッドを追加してください。ユニットクラスではなくバトルクラスに追加するので注意してください。shuffleUnitsは受け取ったユニット配列をランダムに並べ替えるメソッドです。（なお、ここではシャッフルアルゴリズムの内容については触れません。これがどういうものか気になる方はFisher-Yatesシャッフルで検索してみてください。）

backend/server/battle.js

```
（省略）

shuffleUnits(units) {
  // 受け取ったユニットをランダムに並べ替える
  // (Fisher-Yatesシャッフル)
  for (let i = units.length - 1; i > 0; i--){
    const target = Math.floor(Math.random() * (i + 1));
    [units[i], units[target]] = [units[target], units[i]];
  }

  return units;
}

（省略）
```

最後にユニットクラスに戻ってgainAttackとactionに手を加えましょう。それぞれのメソッドを以下のように書き換えて保存してください。gainAttackでは今までの処理に加えてhateメソッドを呼び出して受けたダメージの分ヘイトを蓄積するようになっています。actionではターゲットリストをまずシャッフルしてからヘイト順に並び替えています。これで基本的にはヘイト順に並びつつ同値であればランダムな順番になります。そして配列の0番目、最もヘイトの高い敵をターゲットとして選択してgainAttackを呼び出しています。なお、デバッグ用にヘイトがどういう状況になっているのかをログに出力するようにもしています。

backend/server/battle.js

```
(省略)

gainAttack(attacker) { // 攻撃を受ける
  const damage = Math.floor(Math.random() * (attacker.atk + 1)) + 10;
  // 10 ～ 攻撃者のATK+10点のダメージを受ける
  this.hp -= damage;
  this.battle.log(`  ${this.name}は${damage}点のダメージを受けた！(現在HP：${this.hp})`);
  this.hate(damage, attacker); // 受けたダメージの分ヘイトを蓄積する
}

action() { // 1回の行動を行う
  this.battle.log(`${this.name}の攻撃！`); // メッセージを表示
  const targets = this.battle.getLivingEnemies(this.id); // 生きている（行動者にとっての）敵たちを取得

  // 現在のヘイト量を表示（デバッグ用）
  for (let i = 0; i < targets.length; i++) {
    this.battle.log(`  ${targets[i].name}へのヘイト：${this.hates[targets[i].id]}`);
  }

  const target = this.sortUnitsByHate(this.battle.shuffleUnits(targets))[0];
  // その中から最もヘイトが高い敵をターゲットとして選択（ヘイト同値はランダム）

  // 選ばれたターゲットを表示（デバッグ用）
  this.battle.log(`    → ターゲット：${target.name}`);

  target.gainAttack(this); // 攻撃する（攻撃を受ける処理を呼び出す）
}

(省略)
```

実行すると以下のような感じになります(長いので一部のみ)。ヘイトが高い敵を攻撃しつつ、同値であればランダムな敵を攻撃するようになっていることが分かります。

```
味方Aの攻撃！
  力強い敵Aへのヘイト：30
  力強い敵Bへのヘイト：173
    → ターゲット：力強い敵B
  力強い敵Bは17点のダメージを受けた！(現在HP：36)
味方Bの攻撃！
  力強い敵Aへのヘイト：undefined
  力強い敵Bへのヘイト：undefined
    → ターゲット：力強い敵B
  力強い敵Bは14点のダメージを受けた！(現在HP：22)
味方Cの攻撃！
  力強い敵Aへのヘイト：125
  力強い敵Bへのヘイト：undefined
    → ターゲット：力強い敵A
  力強い敵Aは46点のダメージを受けた！(現在HP：72)
```

4.12.3 その他のステータスの組み込み

それではその他のステータスをシステムに組み込んでいきましょう。それぞれ以下のような効果を持つことにします。

DEX	DEX%の確率でクリティカルになりダメージが2倍になる。
DEF	与えるヘイトの量がDEF倍になる。
MND	回復力だが、まだ回復が実装されていないので今は特に効果はない。

最大HPもステータスから算出されることにします。最低保証HP＋それぞれのステータスに係数をかけ合わせたものが最大HPおよび初期HPとします。また、今はまだ使いませんがアクティブスキルの発動に使用するスキルポイント(SP)も同様の形式で算出して保持しておくことにします。その他に回避システムも作りましょう。基礎回避率＋被攻撃者のAGI%－攻撃者のDEX%の確率で攻撃を回避できることにします。

まずはHP・SPの係数や最低保証、基礎回避率を設定ファイルに記述しておきます。「backend/config/default.json」を開き、以下の設定を追記して保存してください。

backend/config/default.json

```
"basicDodgeRate": 20,
"hpRate": {
  "atk": 2,
  "dex": 2,
  "mnd": 1,
  "agi": 1,
  "def": 5
},
"guaranteedHp": 100,
"spRate": {
  "atk": 3,
  "dex": 3,
  "mnd": 5,
  "agi": 2,
```

```
    "def": 1
  },
  "guaranteedSp": 100
```

設定したら「backend/server/battle.js」に戻り、ユニットクラスのconstructorを以下のように書き換えてください。今までは1つのステータスを1つの引数で受け取っていましたがステータスが増えてきたのでatk, dex, mnd, agi, defプロパティを持ったオブジェクトにより一括で受け取るように、rawStatusに格納するようにしています。

最大HP(mhp)や最大SP(msp)はそれぞれのステータスに係数をかけ合わせて最低保証を足したものが設定されていて、その値に応じて現在HPや現在SPも設定されています。

backend/server/battle.js

```
（省略）

constructor(name, status) { // 初期化処理1
  this.name = name;

  this.rawStatus = { // 生ステータス
    atk: status.atk,
    dex: status.dex,
    mnd: status.mnd,
    agi: status.agi,
    def: status.def
  };

  this.mhp =
    this.atk * config.hpRate.atk +
    this.dex * config.hpRate.dex +
    this.mnd * config.hpRate.mnd +
    this.agi * config.hpRate.agi +
    this.def * config.hpRate.def +
    config.guaranteedHp; // 最大HP
  this.hp = this.mhp;     // 現在HP

  this.msp =
    this.atk * config.spRate.atk +
    this.dex * config.spRate.dex +
    this.mnd * config.spRate.mnd +
    this.agi * config.spRate.agi +
    this.def * config.spRate.def +
    config.guaranteedSp; // 最大SP
  this.sp = this.msp;     // 現在SP

  this.acted    = false; // 行動済ステータス（false: 未行動 true: 行動済み）
  this.living   = true;  // 生存ステータス（false: 戦闘不能 true: 生存）
  this.hates    = [];    // ヘイト量
}

（省略）
```

ステータスを受け取ってrawStatus内で管理するようにはしましたが、this.rawStatus.atkというようにアクセス

するのは若干手間なので今までの形式と同じくthis.atkというような感じでアクセスできるようにしましょう。(本当は別の理由もあるのですがそれはそのときになってから解説します。)これにはクラスのゲッター機能を使います。ユニットクラスの適当なところ(適当な場所であればどこでも構いません、最後尾など)に以下のゲッターを追加してください。

backend/server/battle.js

```
（省略）

get atk() {
  return this.rawStatus.atk;
}

get dex() {
  return this.rawStatus.dex;
}

get mnd() {
  return this.rawStatus.mnd;
}

get agi() {
  return this.rawStatus.agi;
}

get def() {
  return this.rawStatus.def;
}

（省略）
```

追加したらユニットクラスの適当なところ(適切な場所であればどこでも構いません、gainAttackの前など)に以下のメソッドを追加してください。tryDodgeは攻撃者を受け取り攻撃者のDEXと自身のAGIによって回避判定を行うメソッドになります。回避に成功したら「(名前)は攻撃を回避した!」と表示されtrueが、失敗したら特に何も表示されずfalseが帰ってくるメソッドになっています。

backend/server/battle.js

```
（省略）

tryDodge(attacker) {
  const dodgeRate = config.basicDodgeRate + this.agi - attacker.dex;
  // 回避率 基礎回避率 + 自身のAGI% - 攻撃者のDEX%
  if (Math.random() * 100 < dodgeRate) { // 回避成功時、メッセージを表示しtrueを返す
    this.battle.log(`  ${this.name}は攻撃を回避した！`);
    return true;
  } else { // 回避失敗時falseを返す
    return false;
  }
}

（省略）
```

次はgainAttackとactionに手を加えます。gainAttackではクリティカルやDEF倍のヘイト蓄積が行われるようになっており、actionでは攻撃前にtryDodgeを呼び出して対象が回避失敗した場合にのみ攻撃処理を行うように変更しています。

```
backend/server/battle.js
（省略）

  gainAttack(attacker) { // 攻撃を受ける
    let damage = Math.floor(Math.random() * (attacker.atk + 1)) + 10;
    // 10 ～ 攻撃者のATK+10点のダメージ

    if (Math.random() * 100 < attacker.dex) { // 攻撃者のDEX%の確率でクリティカル
      this.battle.log(`  クリティカル！`);
      damage *= 2; // クリティカル時はダメージを2倍に
    }

    this.hp -= damage;
    this.battle.log(`  ${this.name}は${damage}点のダメージを受けた！(現在HP：${this.hp})`);
    this.hate(damage * attacker.def, attacker); // 「受けたダメージ×攻撃者のDEF」分ヘイトを蓄積する
  }

  action() { // 1回の行動を行う
    this.battle.log(`${this.name}の攻撃！`); // メッセージを表示
    const targets = this.battle.getLivingEnemies(this.id); // 生きている（行動者にとっての）敵たちを取得

    // 現在のヘイト量を表示（デバッグ用）
    for (let i = 0; i < targets.length; i++) {
      this.battle.log(`  ${targets[i].name}へのヘイト：${this.hates[targets[i].id]}`);
    }

    const target = this.sortUnitsByHate(this.battle.shuffleUnits(targets))[0];
    // その中から最もヘイトが高い敵をターゲットとして選択（ヘイト同値はランダム）

    // 選ばれたターゲットを表示（デバッグ用）
    this.battle.log(`    → ターゲット：${target.name}`);

    if (!target.tryDodge(this)) { // ターゲットに回避を試みさせ、回避失敗したら
      target.gainAttack(this); // 攻撃する（攻撃を受ける処理を呼び出す）
    }
  }

（省略）
```

バトルクラスも変更しておきましょう。変更するとはいってもユニットにMHPやSP、MSPの概念が増えたのでそれをログに表示するようにするだけです。round_loop:内の「// セットアップ処理とステータス表示」と書かれているfor文のところを以下のように書き換えてください。

```
backend/server/battle.js
（省略）

    for (let i = 0; i < this.units.length; i++) { // セットアップ処理とステータス表示
      this.units[i].setup();
      this.log(
        (this.units[i].isLiving() ? '◆' : '◇') +
        this.units[i].name +
        '  HP:' +
        this.units[i].hp + '/' + this.units[i].mhp +
        '  SP:' +
        this.units[i].sp + '/' + this.units[i].msp
      );
    }

（省略）
```

これでステータスの組み込みは完了です。最後にAPI宣言部分からユニットクラスにそれぞれのステータスを渡すように変更しましょう。以下のように書き換えて保存してください。

```
backend/server/battle.js
（省略）

/*---------------------------------------------
  API宣言
---------------------------------------------*/

app.get(`${config.directoryUrl}battle`, (req, res) => {
  const battle = new Battle([
    new Unit('味方A', {
      atk: 20,
      dex: 20,
      mnd: 20,
      agi: 20,
      def: 20
    }),
    new Unit('味方B', {
      atk: 20,
      dex: 20,
      mnd: 20,
      agi: 20,
      def: 20
    }),
    new Unit('味方C', {
      atk: 20,
      dex: 20,
      mnd: 20,
      agi: 20,
      def: 20
    })
  ], [
    new Unit('速い敵A', {
      atk: 10,
      dex: 10,
```

```
      mnd: 10,
      agi: 30,
      def: 10
    }),
    new Unit('速い敵B', {
      atk: 10,
      dex: 10,
      mnd: 10,
      agi: 30,
      def: 10
    }),
    new Unit('力強い敵A', {
      atk: 20,
      dex: 10,
      mnd: 10,
      agi: 10,
      def: 20
    }),
    new Unit('力強い敵B', {
      atk: 20,
      dex: 10,
      mnd: 10,
      agi: 10,
      def: 20
    })
  ])

  const result = battle.fight();

  return res.status(200).send(result);
});

app.listen(port);
```

実際にアクセスすると以下のような感じになっています(長いので一部のみ)。しっかりとクリティカルや回避、ヘイト倍率が適用されていることが確認できます。ステータスのバランスについてはともかく、かなりゲームらしい戦闘システムになってきたのではないでしょうか。

味方Cの攻撃！
　速い敵Aへのヘイト：390
　速い敵Bへのヘイト：600
　力強い敵Aへのヘイト：340
　力強い敵Bへのヘイト：undefined
　　→ ターゲット：速い敵B
　速い敵Bは27点のダメージを受けた！(現在HP：91)
力強い敵Aの攻撃！
　味方Aへのヘイト：1300
　味方Bへのヘイト：undefined
　味方Cへのヘイト：undefined
　　→ ターゲット：味方A
　クリティカル！
　味方Aは48点のダメージを受けた！(現在HP：177)
力強い敵Bの攻撃！
　味方Aへのヘイト：undefined
　味方Bへのヘイト：2800
　味方Cへのヘイト：undefined
　　→ ターゲット：味方B
　味方Bは攻撃を回避した！

4.12.4 アクティブスキルの構造

ステータスなどの基礎システム周りはこのぐらいにして、次はスキルを組んでいきましょう。まずはアクティブスキルから作っていきます。アクティブスキルは行動の際に実行されるスキルです。今はactionでtryDodgeを呼び出したりgainAttackを呼び出したりといろいろなことをしていますが、これを「アクティブスキルを呼び出す」という感じに変更していきます。アクティブスキルは以下のような感じのものを作ります。(あくまで例であってこのスキルを実際に作るというわけではありません。)

パラダイスロスト
【50SP / 9行動毎】敵:攻撃&毒+SPが30%以上なら敵全:攻撃+自:HP回復

スキルはスキルごとにスキルクラスを作って作成していきます。とはいえスキル1つ1つにSPを消費する処理や今が〇行動目なのか確認する処理を書いていくのはあまりにもムダが大きいです。なので、スキルをパーツごとに分解してパーツごとにクラスを作りその組み合わせでスキルクラスを作っていくことにしましょう。上のようなスキルを以下のようなクラスで表現できるようにすることを目指します。(これはあくまでこういう感じで表現できるようにしようという例なので、実際に完成するものとほぼ同じとはいえ若干異なります。)

```
class ParadiseLost extends ActiveSkill {
  constructor() {
    this.name      = 'パラダイスロスト';
    this.cost      = 50;
    this.condition = new PerActions(9);
    this.skillEffectChain = [
      new SkillEffect(new EnemyTarget(), [new AttackElement(), new PoisonElement()]),
```

```
    new SkillEffect(new AllEnemyTarget(), [new AttackElement()], new SpRemainCondition(30)),
    new SkillEffect(new OwnTarget(), [new HealElement()])
    // 【SP50：通常時】敵：攻撃＋SPが30%以上なら敵全：攻撃＋自：HP回復
  ];
 }
}
```

具体的にはまずそれぞれのアクティブスキルクラスはスキル名と実行する際のSPコストを持っています。他にもconditionという値を持っており、これはスキル全体の発動条件になります。conditionのところには条件を処理するクラスであるConditionクラスおよびそれを継承したクラスが入ります。また、スキルエフェクトチェーン(skillEffectChain)という配列も持ちます。ここにはスキル効果を表すSkillEffectクラスの配列が入ります。

スキルエフェクトでは1つ1つのスキル効果を表現します。ここでスキルエフェクトと呼んでいるのは例でいうと＋で区切られている部分、つまり「敵:攻撃＆毒」「SPが30%以上なら敵全:攻撃」「自:HP回復」になります。ここからスキルエフェクトを構成するものとしてさらにスキルの対象範囲を示すTargetクラス、エフェクト要素を表すElementクラス、そのスキルエフェクトの発動条件を示すSkillEffectConditionクラスに分かれます。スキルエフェクトにはElementクラスが複数指定可能で、「攻撃」というエフェクト要素と「毒」というエフェクト要素を組み合わせて「攻撃＆毒」というスキルエフェクトの効果を表現することが出来ます。

4.12.5 スキルエフェクト

それでは実際に組んでいきましょう。まずはスキルエフェクトについて組んでいくことにします。スキルについてはファイルを分割することにします。「backend/server」内に「skill.js」を作成してください。ファイル構成は以下のようになります。

```
server
  api.js
  battle.js
  index.js
  initialize.js
  model.js
  passport.js
  skill.js
  styling.js
```

まずはスキルエフェクトのパーツを作っていきます。スキルパーツは以下のような役割と機能を持つことにします。

Target	ターゲットになるキャラクターの配列を返す。
SkillElement	キャラクター1体をターゲットとして受け取り、効果を発動させる。
SkillEffectCondition	発動条件を満たしていればtrue、満たしていなければfalseを返す。

「backend/server/skill.js」に以下の内容を入力してください。

backend/server/skill.js

```
/*---------------------------------------------
  ターゲットクラス
---------------------------------------------*/

class Target {
  constructor() {
    this.name = ''; // ターゲット名
  }

  init(unit, battle) { // 初期化処理 スキル使用者とバトル環境を受け取る
    this.unit   = unit;
    this.battle = battle;
  }

  resolve() { // ターゲットになるキャラクターの配列を返す
    return [];
  }
}

class SomeEnemyTarget extends Target {
  // ヘイトの高い順に敵を指定数ターゲットする、同値はランダム
  constructor(number) {
    super();

    this.name   = 1 < number ? `敵${number}` : '敵';
    this.number = number;
  }

  resolve() {
    const livingEnemies = this.battle.getLivingEnemies(this.unit.id); // まず生きている敵を取得し
    const shuffledLivingEnemies = this.battle.shuffleUnits(livingEnemies); // ランダムに並べ替え

    const livingEnemiesSortedByHate = shuffledLivingEnemies.sort((a, b) => {
      // 次にヘイトの高い順に並び替える
      const hateA = this.unit.hates[a.id] || 0; // undefinedだったら0扱い
      const hateB = this.unit.hates[b.id] || 0;
      return hateB - hateA;
    });

    return livingEnemiesSortedByHate.slice(0, this.number);
    // 指定数だけ切り出してターゲットとして返す
    // 生きている敵が指定数より少ないときには指定数よりターゲット数が少なくなる
  }
}

/*---------------------------------------------
  エレメントクラス
---------------------------------------------*/

class Element {
  constructor() {
    this.name = ''; // 効果要素名
  }

  init(unit, battle) { // 初期化処理 スキル使用者とバトル環境を受け取る
```

```
    this.unit   = unit;
    this.battle = battle;
  }

  resolve(target) { // ターゲットに対して効果を発動させる

  }
}

class AttackElement extends Element {
  // 対象に攻撃を行うスキルエレメント
  constructor(potency) {
    super();

    this.name    = '攻撃'; // 効果要素名
    this.potency = potency; // 威力を受け取りその威力を設定
  };

  resolve(target) { // 効果を発動
    target.gainAttack(this.potency, this.unit); // 威力と攻撃者でターゲットの被攻撃メソッドを呼び出す
  }
}

/*------------------------------------------------
  スキルエフェクト発動条件クラス
------------------------------------------------*/

class SkillEffectCondition {
  constructor() {
    this.name = ''; // スキルエフェクト発動条件名
  }

  init (unit, battle) {
    this.unit   = unit;
    this.battle = battle;
  }

  resolve() {
    return false;
  }
}

class SpRemain extends SkillEffectCondition {
  // SPがMSPの指定の%以上なら
  constructor(percentage) {
    super();

    this.name       = `自身のSPが${percentage}%以上なら`;
    this.percentage = percentage;
  }

  resolve() {
    return this.percentage <= (this.unit.sp / this.unit.msp) * 100;
  }
}
```

それぞれのクラスを見ていきましょう。Target、Element、SkillEffectConditionはそれぞれ継承して使うためのクラスになります。ここでinitメソッドを宣言しており初期化時はこれを呼び出してユニットやバトルクラスにアクセスできるようにしています。それぞれのクラスはresolveというメソッドを持っており、これが実行されるとそれぞれの処理の内容に応じた処理を実行したり、値を返したりします。

SomeEnemyTargetクラスはTargetクラスを継承したもので、ターゲット数になる数を受け取って生きている敵のうちヘイトの高い順から指定数だけユニットを返すようになっています。AttackElementクラスはElementクラスを継承したもので、威力を設定できます。resolveではターゲットを受け取り、ターゲットのgainAttackに威力と攻撃者を渡して呼び出しています。SpRemainクラスはSkillEffectConditionクラスを継承したもので、自身の残りSPがMSPの指定の％以上ならtrue、そうでなければfalseを返すようになっています。それぞれ処理の内容の割に見た目が長いですが、内容を見てみるとそんなに複雑なことはしていません。

それでは次はそれらを組み合わせるクラスであるSkillEffectクラスを作りましょう。先程入力したコードの後に続いて以下のコードを入力してください。

backend/server/skill.js

```
（省略）

/*----------------------------------------------
  スキルエフェクトクラス
----------------------------------------------*/

class SkillEffect {
  constructor(target, dodgeable, elements, condition) {
    this.target    = target;    // ターゲット情報
    this.dodgeable = dodgeable; // 回避可能かどうか（true: 回避可能, false: 回避不可）
    this.elements  = elements;  // スキルエフェクト要素
    this.condition = condition; // スキルエフェクト発動条件（指定しない場合特に発動条件はない）
  }

  init(unit, battle) { // 初期化処理 スキル使用者とバトル環境を受け取り
    this.unit   = unit;
    this.battle = battle;
    this.target.init(unit, battle); // ターゲットと

    if (this.condition) {
      this.condition.init(unit, battle); // （あれば）スキルエフェクト発動条件と
    }

    for (let i = 0; i < this.elements.length; i++) {
      this.elements[i].init(unit, battle); // エフェクト要素を初期化
    }
  }

  resolve() { // スキルエフェクト発動
    if (this.condition && !this.condition.resolve()) {
      // スキルエフェクト発動条件があり、発動条件を満たしていないなら
      // 特に何もせず処理を終える
      return;
    }

    const targets = this.target.resolve(); // ターゲットになるキャラクターの配列を取得
```

```
    for (let i = 0; i < targets.length; i++) { // ターゲットごとに処理を繰り返す
      if (this.dodgeable && targets[i].tryDodge(this.unit)) {
        // 回避可能なスキルエフェクトなら回避を試み、成功なら
        continue; // 処理をスキップ、次のターゲットへ
      }

      // 回避に失敗したらスキルエフェクト要素たちを発動させる
      for (let j = 0; j < this.elements.length; j++) {
        this.elements[j].resolve(targets[i]); // 実行
      }
    }
  }
}
```

SkillEffectクラスではTarget、Element、SkillEffectConditionのクラスを持っている他に回避可能かどうか？という情報も持つようにしています。これはなぜかというと、攻撃スキルは基本的に回避できたほうがいいですが回復スキルなどを回避するのは少し変になってしまうためスキルエフェクトが回避可能かどうかも設定できるようにしています。SkillEffectConditionについては指定しないことも可能とします。特にSkillEffectConditionを指定しない場合、条件なしで発動可能という扱いになります。

initではユニットとバトルクラスを受け取って自身を初期化している他に自身の持つTarget、Element、(あれば)SkillEffectConditionも初期化するようにしています。

resolveはスキルエフェクトを発動するメソッドです。まず発動条件チェックを行っています。SkillEffectConditionが設定されていてそれをresolveしたときにfalseが帰ってくればスキルエフェクトは発動されません。returnして処理を中断します。発動条件が満たされていればTargetをresolveしてターゲットになるユニットの配列を受け取ります。そして受け取ったユニットたちから1体1体を取り出して回避可能であれば回避を試みさせます。回避成功すれば次のユニットへ、失敗すればElementたちを発動させています。これでSkillEffectが組み上がりました。

4.12.6 アクティブスキル

アクティブスキル全体を作っていきます。まずはアクティブスキル全体の発動条件を組んでいきましょう。先程入力したコードの後に以下のコードを入力してください。

backend/server/skill.js

```
（省略）

/*-----------------------------------------------
  スキル発動条件
-----------------------------------------------*/

class Condition {
  constructor() {
    this.name = ''; // 発動条件名
  }

  init (unit, battle) {
    this.unit   = unit;
    this.battle = battle;
```

```
  }

  resolve() {
    return false;
  }
}

class Always extends Condition {
  constructor() {
    super();

    this.name = '通常時';
  }

  resolve() { // 常に発動可能
    return true;
  }
}

class PerActions extends Condition {
  constructor(perActions) {
    super();

    this.name       = `${perActions}行動毎`;
    this.perActions = perActions;
  }

  resolve() {
    return !(this.unit.actionCount % this.perActions); // 指定の行動ごと
  }
}
```

スキルエフェクト発動条件とやっていることはほぼ変わりません。指定の条件を満たしたらtrue、満たしていなければfalseが返るようになっています。Alwaysは常時実行可能、PerActionsは○行動毎に実行可能という条件になっています。PerActionsではユニットのactionCountを参照していますが、これはまだ作っていないので後で実装するようにしましょう。

ここまで組んだらスキルを組んでいきましょう。先程入力したコードの次に以下のコードを入力してください。

backend/server/skill.js

```
 （省略）

/*---------------------------------------------
  スキル
---------------------------------------------*/

class ActiveSkill {
  constructor() {
    this.name = ''; // スキル名
    this.cost = 0;  // 実行した際に消費するSP
    this.condition = null; // スキル発動条件を初期化
    this.skillEffectChain = null; // 実行されるスキルエフェクトチェーン
  }
```

```
  init(unit, battle) { // 初期化処理 スキル使用者とバトル環境を受け取る
    this.unit   = unit;
    this.battle = battle;

    this.condition.init(unit, battle); // スキル発動条件を初期化
    for (let i = 0; i < this.skillEffectChain.length; i++) {
      this.skillEffectChain[i].init(unit, battle); // スキルエフェクトをそれぞれ初期化
    }
  }

  isCostOk() { // SPが足りているかどうか
    return this.cost <= this.unit.sp;
  }

  isExecutable() { // 実行可能かどうか
    return this.condition.resolve();
  }

  execute() { // スキル実行
    for (let i = 0; i < this.skillEffectChain.length; i++) {
      this.skillEffectChain[i].resolve();
    }
  }

  afterExecute() { // 実行後処理、SPを消費する
    this.unit.consumeSp(this.cost);
  }
}

class NormalAttack extends ActiveSkill {
  constructor() {
    super();

    this.name = '通常攻撃';
    this.cost = 0;
    this.condition = new Always();
    this.skillEffectChain = [
      new SkillEffect(new SomeEnemyTarget(1), true, [new AttackElement(50)])
      // 【SP0：通常時】敵：攻撃（回避可能、攻撃の威力50）
    ];
  }

  isCostOk() { // SPが足りているかどうか、通常攻撃は必ず実行可能とする
    return true;
  }

  isExecutable() { // 実行条件、通常攻撃は必ず実行可能とする
    return true;
  }
}

class Burst extends ActiveSkill {
  constructor() {
    super();

    this.name = 'バースト';
```

```
    this.cost = 40;
    this.condition = new PerActions(4);
    this.skillEffectChain = [
      new SkillEffect(new SomeEnemyTarget(2), true, [new AttackElement(80)]),
      new SkillEffect(new SomeEnemyTarget(1), true, [new AttackElement(100)], new SpRemain(90))
      // 【SP40：4行動毎】敵2：攻撃（回避可能、攻撃の威力80）＋自身のSPが90%以上なら敵：攻撃（回避可能、攻撃の威力100）
    ];
  }
}
```

まずActiveSkillクラスから見ていきましょう。ActiveSkillはスキル名、実行した際に消費するSP、Condition、SkillEffectの配列（これを以降スキルエフェクトチェーンと呼びます）の4つの値を持っています。

次はそれぞれのメソッドを見ていきましょう。まずはinitです。initではユニット、バトルクラスを受け取って初期化している他、Conditionとスキルエフェクトチェーンについても初期化を行っています。

isCostOkとisExecutableは実行条件になっています。isCostOkはこのスキルを持つユニットが実行に必要なSPを持っているかの条件になっていて、isExecutableはConditionをresolveして発動条件を満たしているかどうかを返します。実際にActiveSkillを呼び出す際はこの2つのメソッドを呼び出して両方の条件を満たしていたら実行するという感じになるでしょう。

executeはスキルエフェクトチェーンを実行する処理です。実際にスキルエフェクトを発動する処理はSkillEffectのresolveに記述されているので、ここではスキルエフェクトの分だけ繰り返して実行するだけです。afterExecuteは実行後処理になります。ここでユニットのconsumeSpメソッドを呼び出してSPコスト分をユニットに消費させます。consumeSpメソッドはまだ実装されていませんが後で実装することにしましょう。

さて、NormalAttackとBurstはそれぞれ実際のアクティブスキルになります。今までのパーツを組み合わせるだけで作れるようになっています。なお、NormalAttackは他に実行できるアクティブスキルがないときに実行されるアクティブスキルにします。そのような性質の関係上、NormalAttackのisCostOkとisExecutableは必ずtrueが返るようにしてあります。

スキルを作り終えたら他ファイルから読み込めるようにしましょう。いちいちスキル名を設定して読み込むのは面倒な上に管理も複雑になってしまうので、ここではスキルごとにIDを割り振るようにし、読み込み時はスキルIDの配列を受け取ってスキルインスタンスの配列を返すようにします。先程入力したコードの次に以下のコードを入力してください。スキルIDの割り振りと読み込みには配列を使っており、配列のインデックス＝スキルIDになるようにしています。

backend/server/battle.js

```
  （省略）

/*----------------------------------------------
  読み込み
----------------------------------------------*/

const skillArray = [
  /*  0 */ NormalAttack,
  /*  1 */ Burst
];
```

```
const skillResolver = (skillIds) => {
  // スキル読み込み用の関数
  // スキルIDの配列を受けとってスキルのインスタンス配列を返す
  const skills = [];

  for (let i = 0; i < skillIds.length; i++) {
    const SkillClass = skillArray[skillIds[i]];
    skills.push(new SkillClass());
  }

  return skills;
};

module.exports = {
  skillResolver: skillResolver
};
```

ここまで設定したら今度は「backend/server/battle.js」の方も書き換えていきましょう。まずは「skill.js」を読み込むようにします。冒頭部分を以下のように書き換えてください。

backend/server/battle.js

```
const express = require('express');
const config  = require('config');
const skill   = require('./skill.js');
const app     = express();
const port    = config.port;

  (省略)
```

次にユニットクラスの初期化処理であるconstructorとinitを以下のように書き換えます。constructorではスキルID配列を受け取るようになっています。何も実行できるものがないときに通常攻撃が出るようにしたいため、このときにスキルID配列の末尾に0(通常攻撃のスキルID)を追加するようにしています、initではスキルIDからスキルを読み込んで初期化してactiveSkillsに格納するようにしています。また、constructor内で行動回数カウントを初期化するようにしています。

backend/server/battle.js

```
  (省略)

  constructor(name, status, skillIds) { // 初期化処理1
    this.name = name;

    this.rawStatus = { // 生ステータス
      atk: status.atk,
      dex: status.dex,
      mnd: status.mnd,
      agi: status.agi,
      def: status.def
    };

    this.mhp =
```

```
    this.atk * config.hpRate.atk +
    this.dex * config.hpRate.dex +
    this.mnd * config.hpRate.mnd +
    this.agi * config.hpRate.agi +
    this.def * config.hpRate.def +
    config.guaranteedHp; // 最大HP
  this.hp = this.mhp;     // 現在HP

  this.msp =
    this.atk * config.spRate.atk +
    this.dex * config.spRate.dex +
    this.mnd * config.spRate.mnd +
    this.agi * config.spRate.agi +
    this.def * config.spRate.def +
    config.guaranteedSp; // 最大SP
  this.sp = this.msp;     // 現在SP

  this.skillIds = skillIds; // スキルID配列
  this.skillIds.push(0);    // スキルID配列の末尾に0（通常攻撃）を追加

  this.acted       = false; // 行動済ステータス（false: 未行動 true: 行動済み）
  this.living      = true;  // 生存ステータス（false: 戦闘不能 true: 生存）
  this.actionCount = 0;     // 行動回数カウント
  this.hates       = [];    // ヘイト量
}

init(id, battle) { // 初期化処理2 ユニット生成時には受け取れない値をここでセット
  this.id     = id;
  this.battle = battle;

  const skills = skill.skillResolver(this.skillIds); // スキルをロード

  for (let i = 0; i < skills.length; i++) { // スキルを初期化
    skills[i].init(this, battle);
  }

  this.activeSkills = skills; // アクティブスキルとして受け取る
}

（省略）
```

次はSP消費処理を作ります。ユニットクラスの適当なところ（適切な場所であればどこでも構いません、tryDodgeの前など）に以下のメソッドを追加してください。

backend/server/battle.js

```
consumeSp(cost) { // SP消費処理
  this.sp -= cost;
}
```

次にgainAttackの処理を変更します。攻撃の威力という概念ができたので、威力と攻撃者を受け取ってそれに応じたダメージが出るようにしています。ここでは「攻撃者のATK+10」を基礎ダメージとし、「基礎ダメージ×威

力÷50×0.8 ～ 1.2」がダメージになるようにしています。

backend/server/battle.js

```
（省略）

gainAttack(potency, attacker) { // 威力と攻撃者を受け取る
  let basicDamage = 10 + attacker.atk; // 基礎ダメージ 攻撃者のATK+10
  let damage = Math.floor((basicDamage * potency / 50) * (0.8 + Math.random() * 0.4));
  // ダメージ (基礎ダメージ×威力÷50) × 0.8～1.2

  if (Math.random() * 100 < attacker.dex) { // 攻撃者のDEX%の確率でクリティカル
    this.battle.log(` クリティカル！`);
    damage *= 2; // クリティカル時はダメージを2倍に
  }

  this.hp -= damage;
  this.battle.log(` ${this.name}は${damage}点のダメージを受けた！(現在HP：${this.hp})`);
  this.hate(damage * attacker.def, attacker); // 「受けたダメージ×攻撃者のDEF」分ヘイトを蓄積する
}

（省略）
```

次にactionを変更します。アクティブスキルは配列の上から順番に実行できるかをチェックしていき、実行できるものがあればそれを実行するようにします。constructorの処理により配列の一番後ろに通常攻撃スキルがつくようになっているので、何も実行できるものがないときは通常攻撃が実行されるようになっています。また、発動するスキルの名前と現在の行動回数も表示するようにしています。

backend/server/battle.js

```
（省略）

action() { // 1回の行動を行う
  this.actionCount++; // 行動回数カウントを+1

  // アクティブスキル配列の上から順番に実行判断
  for (let i = 0; i < this.activeSkills.length; i++) {
    if (this.activeSkills[i].isCostOk() && this.activeSkills[i].isExecutable()) {
      // 実行コストが足りていて実行可能なら
      this.battle.log(`${this.name}の${this.activeSkills[i].name}！(${this.actionCount}行動目)`);
      // 名前、実行するスキル名、行動回数を表示し
      this.activeSkills[i].execute(); // 実行
      this.activeSkills[i].afterExecute();  // 実行後処理を行う
      return;
      // 通常攻撃は必ず末尾に登録されておりかつ実行可能になっているので
      // 他に実行できるスキルがなければ通常攻撃になる
    }
  }
}

（省略）
```

あとは必要なくなったメソッドを消しておきましょう。ユニットクラスのsortUnitsByHateの部分の役割はスキルパーツであるSomeEnemyTargetが担うようになり必要なくなったので削除しておきます。

最後にユニットクラスにスキルID配列を渡すようにしましょう。API宣言の部分を以下のように書き換えて保存してください。「味方」と「速い敵」が「バースト」のスキルを取得するようにしてあります。

backend/server/battle.js

```
（省略）

/*---------------------------------------------
  API宣言
---------------------------------------------*/

app.get(`${config.directoryUrl}battle`, (req, res) => {
  const battle = new Battle([
    new Unit('味方A', {
      atk: 20,
      dex: 20,
      mnd: 20,
      agi: 20,
      def: 20
    }, [1]),
    new Unit('味方B', {
      atk: 20,
      dex: 20,
      mnd: 20,
      agi: 20,
      def: 20
    }, [1]),
    new Unit('味方C', {
      atk: 20,
      dex: 20,
      mnd: 20,
      agi: 20,
      def: 20
    }, [1]),
  ], [
    new Unit('速い敵A', {
      atk: 10,
      dex: 10,
      mnd: 10,
      agi: 30,
      def: 10
    }, [1]),
    new Unit('速い敵B', {
      atk: 10,
      dex: 10,
      mnd: 10,
      agi: 30,
      def: 10
    }, [1]),
    new Unit('力強い敵A', {
      atk: 20,
      dex: 10,
      mnd: 10,
```

```
      agi: 10,
      def: 20
    }, []),
    new Unit('力強い敵B', {
      atk: 20,
      dex: 10,
      mnd: 10,
      agi: 10,
      def: 20
    }, [])
  ]);

  const result = battle.fight();

  return res.status(200).send(result);
});

app.listen(port);
```

それではアクセスしてみましょう。以下のような感じになっているはずです(長いので一部のみ)。「味方」と「速い敵」は4回行動するごとに「バースト」のスキルを発動するようになっています。バーストの2つ目のスキルエフェクトはSPが90%以上でないと発動しないようになっているので、8行動目以降のバーストは攻撃回数が減っていることも確認してみてください。

```
速い敵Aの通常攻撃！(2行動目)
  味方Aは18点のダメージを受けた！(現在HP：279)
速い敵Bの通常攻撃！(3行動目)
  味方Cは攻撃を回避した！
連続行動！
速い敵Bのバースト！(4行動目)
  味方Bは26点のダメージを受けた！(現在HP：294)
  味方Cは29点のダメージを受けた！(現在HP：264)
  味方Cは攻撃を回避した！
```

4.12.7 スキルの追加

既存のパーツを組み合わせるだけでもある程度いろいろなスキルは作れるようになっていますが、もっと幅広いスキルを実装していきましょう。まったく新しいスキルをこのシステムに実装しようとした場合、以下のような手順を踏むことになるでしょう。

1. 作りたい効果や条件などのパーツを作る
2. Elementなどで使う必要な処理をユニットクラスなどに実装する
3. パーツを組み合わせて新しいスキルを作る

例えばここからHPが低い味方がいるときに弱っている味方を回復するスキルを作りたい場合は以下のような手順になります。

1. Target「弱っている味方」、Element「HP回復」、Condition「弱っている味方がいる」をそれぞれ作る
2. バトルクラスにgetLivingAllies、ユニットクラスにgainHealを実装する
3. 作ったパーツを組み合わせてスキルを実装する

それでは作ってみましょう。「backend/server/skill.js」を開き、以下のパーツをそれぞれ追記してください。適切な場所であればどこでも構いません。SomeWeakenedAllyTargetはターゲットクラスの箇所の末尾に、HealElementはエレメントクラスの箇所の末尾に、WeakenedAllyExistsはスキル発動条件の箇所の末尾にそれぞれ追加するといいでしょう。

backend/server/skill.js

```
（省略）

class SomeWeakenedAllyTarget extends Target {
  constructor(number) {
    super();

    this.name   = 1 < number ? `弱味${number}` : '弱味';
    this.number = number;
  }

  resolve() {
    // MHPに対するHPの割合が低い順に味方を指定数ターゲットする、同値はランダム
    const livingAllies = this.battle.getLivingAllies(this.unit.id); // まず生きている味方を取得
    const shuffledLivingAllies = this.battle.shuffleUnits(livingAllies); // ランダムに並べ替え

    const livingAlliesSortedByHpRate = shuffledLivingAllies.sort((a, b) => {
      // 次にHP割合の低い順に並び替える
      const hpRateA = a.hp / a.mhp;
      const hpRateB = b.hp / b.mhp;
      return hpRateA - hpRateB;
    });

    return livingAlliesSortedByHpRate.slice(0, this.number);
    // 指定数だけ切り出してターゲットとして返す
    // 生きている味方が指定数より少ないときには指定数よりターゲット数が少なくなる
  }
}

（省略）

class HealElement extends Element {
  constructor(potency) {
    super();

    this.name    = 'HP回復';
    this.potency = potency;
  }

  resolve(target) {
    target.gainHeal(this.potency, this.unit); // 威力と回復者でターゲットの被回復メソッドを呼び出す
  }
```

```
}

  （省略）

class WeakenedAllyExists extends Condition {
  constructor() {
    super();

    this.name = '味方重傷';
  }

  resolve() { // HPが50%以下の味方がいるとき
    const livingAllies = this.battle.getLivingAllies(this.unit.id); // まず生きている味方を取得

    for (let i = 0; i < livingAllies.length; i++) {
      if (livingAllies[i].hp / livingAllies[i].mhp <= 0.5) {
        return true; // HP50%以下の味方がいるなら実行可能
      }
    }

    return false; // いなければ実行不可
  }
}
```

パーツ内でgetLivingAlliesやgainHealという新たなメソッドにアクセスするようになっているのでそれぞれのメソッドを追加していきます。「backend/server/battle.js」を開き、まずはバトルクラスの適当なところ(適切な場所であればどこでも構いません、getLivingEnemiesの後など)に以下のメソッドを追加してください。

backend/server/battle.js

```
  （省略）

  getAllies(unitId) {
    // IDのユニットから見た味方側チームを取得
    return this.allyIds.includes(unitId) ? this.allies : this.enemies;
  }

  getLivingAllies(unitId) {
    // IDのユニットから見た生きている味方側チームを取得
    return this.extractLiving(this.getAllies(unitId));
  }

  （省略）
```

次にgainHealを作ります。ユニットクラスの適当なところ(適切な場所であればどこでも構いません、gainAttackの後など)に以下のメソッドを追加してください。回復の威力と回復者を受け取って回復処理を行うようにしています。このとき、回復後のHPがMHPを超えないようになっています。

backend/server/battle.js

```
  （省略）
```

```
  gainHeal(potency, healer) {
    const basicHeal = (10 + healer.mnd); // 基礎回復量 回復者のMND+10
    const heal = Math.floor((basicHeal * potency / 50) * (0.8 + Math.random() * 0.4));
    // 回復量 (基礎回復量×威力÷50) × 0.8〜1.2

    const formarHp = this.hp; // 回復前のHP
    this.hp += heal; // 回復する
    if (this.mhp < this.hp) {
      this.hp = this.mhp; // MHPを超えたらMHPにする
    }
    const actualHealed = this.hp - formarHp; // 実際の回復量

    this.battle.log(`  ${this.name}はHPが${actualHealed}点回復した！(現在HP：${this.hp})`);
  }

 （省略）
```

それでは作ったパーツを組み合わせてスキルを作りましょう。「backend/server/skill.js」を開き適当なところ（適切な場所であればどこでも構いません、Burstクラスの次など）に以下のスキルを追加してください。最も弱っている味方1体に威力40で回復をするようにしています。また、回復を回避するのは少し変なので回避可能かどうかのステータスをfalseに設定します。

backend/server/skill.js

```
 （省略）

class Heal extends ActiveSkill {
  constructor() {
    super();

    this.name = 'ヒール';
    this.cost = 20;
    this.condition = new WeakenedAllyExists();
    this.skillEffectChain = [
      new SkillEffect(new SomeWeakenedAllyTarget(1), false, [new HealElement(40)])
      // 【SP20：味方重傷】弱味：HP回復
    ];
  }
}

 （省略）
```

あとはこのスキルを外部から読み込めるようにしましょう。読み込みと書かれたところの下のskillArrayを以下のように書き換えて保存します。

backend/server/skill.js

```
 （省略）

/*-----------------------------------------------
  読み込み
-----------------------------------------------*/
```

```
const skillArray = [
  /*  0 */ NormalAttack,
  /*  1 */ Burst,
  /*  2 */ Heal
];

  (省略)
```

それではこのスキルを使ってみましょう。「backend/server/battle.js」を開き、API宣言内の「味方C」を以下のように書き換えて保存してください。これで「味方C」がヒールを習得して条件を満たしたときに使うようになります。

```
backend/server/battle.js
  (省略)

    new Unit('味方C', {
      atk: 20,
      dex: 20,
      mnd: 20,
      agi: 20,
      def: 20
    }, [2, 1]),

  (省略)
```

それぞれ保存したらアクセスしてみましょう。以下のような感じになります(長いので一部のみ)。HPが50%未満の味方がいる場合、「味方C」が最もHP割合の低い味方に対してヒールを行うようになっています。

```
味方Cのヒール！(5行動目)
  味方BはHPが21点回復した！(現在HP：67)
```

4.12.8 状態異常

次は状態異常を追加してみましょう。今回は「毒」という状態異常を作ります。毒が付与されていると毎ラウンドの最後に5 ～ 15のHPダメージを受けるようになる他、効果中はATK、DEX、AGIが0.8倍になることにします。状態異常は残りターン数という概念を持ち、毎ラウンド最後に1ずつ減少して0になったら効果終了とします。

それでは組み込んでいきましょう。「backend/server/battle.js」を開きます。まずはユニットクラスのconstructorのthis.rawStatusを宣言しているところの次に以下のようにコードを追加してください。追加された部分は斜体で表示してあります。状態異常(effect)が初期化されるようになっています。

```
backend/server/battle.js
  (省略)

  constructor(name, status, skillIds) { // 初期化処理1
    this.name = name;
```

```
    this.rawStatus = { // 生ステータス
      atk: status.atk,
      dex: status.dex,
      mnd: status.mnd,
      agi: status.agi,
      def: status.def
    };

    this.effect = { // 状態異常の残りターン数
      poison: 0
    };

（省略）
```

次に毒を受ける処理を追加します。適当なところ（適切な場所であればどこでも構いません、gainHealの次など）に以下のメソッドを追加してください。毒の効果ターン数を受け取り、指定のターン数毒のターン数を延長してメッセージを表示しています。

backend/server/battle.js

```
  gainPoison(turn) {
    this.effect.poison += turn;
    this.battle.log(`　${this.name}は毒を${turn}ターン分受けた！`);
  }
```

次にcleanupメソッドを改変しましょう。まだ戦闘不能ではなく毒の効果中であれば毒によりダメージを受ける処理と残りターン数を-1する処理を戦闘不能判定前に行うようにしています。

backend/server/battle.js

```
（省略）

  cleanup() {
    if (this.living && this.effect.poison) {
      const poisonDamage = Math.floor(Math.random()* 11) + 5;
      // 毒はクリーンアップ時に5～15のダメージを受ける
      this.hp -= poisonDamage;
      this.effect.poison--; // ターン数を減少
      if (this.effect.poison) { // まだターン数が残っていれば
        this.battle.log(`　${this.name}は毒により${poisonDamage}のダメージを受けた！(残りターン：
${this.effect.poison})`);
      } else { // 治癒したら
        this.battle.log(`　${this.name}は毒により${poisonDamage}のダメージを受けた！`);
        this.battle.log(`　${this.name}の毒は治った！`);
      }
    }

    if (this.living && this.hp <= 0) { // クリーンアップ時生きていてHP0以下なら戦闘不能に
      this.living = false;
      this.battle.log(`　${this.name}は倒れた……`);
    }
  }
```

```
（省略）
```

最後に毒の効果時間中はATK、DEX、AGIが0.8倍になるようにします。これはユニットクラスのゲッターを書き換えることで実現します。該当の部分を以下のように書き換えてください。ステータスにゲッターを使っていたメインの理由はこの方がステータスに補正を入れて計算がしやすくなるから、というわけです。

backend/server/battle.js

```
（省略）

get atk() {
  return this.rawStatus.atk * (this.effect.poison ? 0.8 : 1); // 毒効果中は0.8倍になる
}

get dex() {
  return this.rawStatus.dex * (this.effect.poison ? 0.8 : 1); // 毒効果中は0.8倍になる
}

get mnd() {
  return this.rawStatus.mnd;
}

get agi() {
  return this.rawStatus.agi * (this.effect.poison ? 0.8 : 1); // 毒効果中は0.8倍になる
}

get def() {
  return this.rawStatus.def;
}

（省略）
```

次に毒を付与するスキルを作りましょう。まずは毒を付与するElementを作ります。「backend/server/skill.js」を開き、適当なところ（適切な場所であればどこでも構いません、エレメントクラス部分の末尾など）に以下のクラスを追加してください。ターン数を渡すと発動時にターゲットにそのターン数の毒を付与します。

backend/server/skill.js

```
class PoisonElement extends Element {
  constructor(turn) {
    super();

    this.name = '毒';
    this.turn = turn;
  }

  resolve(target) {
    target.gainPoison(this.turn);
  }
}
```

Elementができたらスキルを作ります。適当なところ(適切な場所であればどこでも構いません、Healクラスの次など)に以下のクラスを追加してください。4行動毎に最もヘイトの高い敵1体に対して攻撃し毒を付与します。

backend/server/skill.js

```
class PoisonAttack extends ActiveSkill {
  constructor() {
    super();

    this.name = 'ポイズン';
    this.cost = 30;
    this.condition = new PerActions(4);
    this.skillEffectChain = [
      new SkillEffect(new SomeEnemyTarget(1), true, [new AttackElement(30), new PoisonElement(3)])
      // 【SP30：4行動毎】敵：攻撃＆毒
    ];
  }
}
```

このスキルを外部から読み込めるようにしましょう。skillArrayの部分を以下のように書き換えてください。ID3がこのスキルのIDになります。

backend/server/skill.js

```
const skillArray = [
  /*  0 */ NormalAttack,
  /*  1 */ Burst,
  /*  2 */ Heal,
  /*  3 */ PoisonAttack
];
```

最後にこのスキルをキャラクターに習得させましょう。「backend/server/battle.js」のAPI宣言部分の「味方B」を作っている箇所を以下のように書き換えてください。「味方B」は「バースト」の代わりに「ポイズン」を使うようになります。

backend/server/battle.js

```
  (省略)

    new Unit('味方B', {
      atk: 20,
      dex: 20,
      mnd: 20,
      agi: 20,
      def: 20
    }, [3]),

  (省略)
```

編集が完了したらそれぞれ保存してアクセスしてみましょう。以下のような感じになっているはずです(長いので

一部のみ)。毒が反映されるようになっています。

```
味方Bのポイズン！(4行動目)
  クリティカル！
  力強い敵Aは36点のダメージを受けた！(現在HP：180)
  力強い敵Aは毒を3ターン分受けた！
味方Cのヒール！(3行動目)
  味方CはHPが22点回復した！(現在HP：159)
力強い敵Bの通常攻撃！(3行動目)
  味方Aは30点のダメージを受けた！(現在HP：161)
力強い敵Aの通常攻撃！(4行動目)
  味方Bは24点のダメージを受けた！(現在HP：186)

  力強い敵Aは毒により15のダメージを受けた！(残りターン：2)
```

4.12.9 パッシブスキル

次はパッシブスキルを作っていきましょう。パッシブスキルとはキャラクターの行動タイミングではなく「戦闘開始時」や「攻撃回避時」、「被攻撃時」など指定のタイミングで発動するスキルのことです。アクティブスキルが消費SPと実行条件を持つ代わりに、パッシブスキルは発動トリガーと発動率を持つことにしましょう。まずはパッシブスキルのクラスを作ります。「backend/server/skill.js」を開き、class ActiveSkillのところを以下の3つのクラスに書き換えてください。PassiveSkillが追加されている他、ActiveSkillとPassiveSkillに共通するものはSkillというクラスにまとめています。それに伴いActiveSkillクラスの内容がSkillクラスに一部移管されています。

backend/server/api.js

```
class Skill {
  constructor() {
    this.name = ''; // スキル名
    this.skillEffectChain = null; // 実行されるスキルエフェクトチェーン
  }

  init(unit, battle) { // 初期化処理 スキル使用者とバトル環境を受け取る
    this.unit   = unit;
    this.battle = battle;

    for (let i = 0; i < this.skillEffectChain.length; i++) {
      this.skillEffectChain[i].init(unit, battle); // スキルエフェクトをそれぞれ初期化
    }
  }

  execute() { // スキル実行
    for (let i = 0; i < this.skillEffectChain.length; i++) {
      this.skillEffectChain[i].resolve();
    }
  }
}

class ActiveSkill extends Skill {
  constructor() {
```

```
    super();

    this.name = ''; // スキル名
    this.cost = 0;  // 実行時に消費するSP
    this.condition = null; // スキル発動条件を初期化
    this.skillEffectChain = null; // 実行されるスキルエフェクトチェーン
  }

  init(unit, battle) { // 初期化処理 スキル使用者とバトル環境を受け取る
    super.init(unit, battle); // 親クラスのinitを呼び出してから
    this.condition.init(unit, battle); // スキル発動条件を初期化
  }

  isCostOk() { // SPが足りているかどうか
    return this.cost <= this.unit.sp;
  }

  isExecutable() { // 実行可能かどうか
    return this.condition.resolve();
  }

  afterExecute() { // 実行後処理、SPを消費する
    this.unit.consumeSp(this.cost);
  }
}

class PassiveSkill extends Skill {
  constructor() {
    super();

    this.name = '';      // スキル名
    this.rate = 0;       // スキル発動率
    this.trigger = null; // 発動トリガー
    this.skillEffectChain = null; // 実行されるスキルエフェクトチェーン
  }

  isRateCheckOk() { // 発動率チェック
    return Math.random() * 100 < this.rate;
  }

  getTriggerId() {
    return this.trigger.id;
  }
}
```

PassiveSkillはisRateCheckOkというメソッドを持っています。これは自身のスキル発動率の確率で判定を行い成功すればtrueが、失敗すればfalseが返るメソッドになっています。スキル実行前にこれを呼び出して失敗すれば処理をしない、という感じにすることでスキル発動率を表現します。

また、PassiveSkillでは発動トリガーをクラスで表現することにします。このクラスをTriggerとしましょう。PassiveSkillのgetTriggerId内ではTriggerのidをそのまま返しています。これによりスキルがどういうタイミングで発動するのかを呼び出している側は知ることができます。つまり、Triggerクラスはidという値を持っておけばいいだけです。以下のような感じになるでしょう。以下のコードを適当なところ（適切な場所であればどこでも構いません、スキ

ル発動条件のところの前など)に入力してください。

backend/server/skill.js

```
/*---------------------------------------------
  トリガークラス
---------------------------------------------*/

class Trigger {
  constructor() {
    this.name = ''; // 表示されるトリガー名
    this.id   = ''; // 内部的なトリガーID
  }
}

class Dodge extends Trigger { // 回避したとき
  constructor() {
    super();

    this.name = '回避時';
    this.id   = 'dodge';
  }
}
```

なんの機能ももたない、本当にIDとついでに表示名を保管しているだけのクラスです。これだけならクラス化せずともPassiveSkillがトリガーIDを文字列そのままで持っておけばいいのでは?という感じもありますし実際そういう部分はあるのですが、クラスで記述しておいたほうがスキルを組むときにミスタイプしていた場合エラーが表示されるようになってミスを減らすことができるのでここではクラス化しています。

ここまで組んだらパッシブスキルを作りましょう。以下のクラスを適当なところ(適切な場所であればどこでも構いません、PoisonAttackの後など)に追加してください。回避時に30%の確率で敵1体に攻撃を行うスキルになります。

backend/server/battle.js

```
class CounterAttack extends PassiveSkill {
  constructor() {
    super();

    this.name = '反撃';
    this.rate = 30;
    this.trigger = new Dodge();
    this.skillEffectChain = [
      new SkillEffect(new SomeEnemyTarget(1), true, [new AttackElement(50)])
      // 【回避時】敵:攻撃
    ]
  }
}
```

次は「読み込み」の部分を以下のように書き換えてください。先程作ったスキルにIDを割り振っている他、スキルがActiveSkillなのかPassiveSkillなのかを判定する関数を追加しています。これはユニットクラス側でActiveSkil

lとPassiveSkillを振り分けるのに使われます。

backend/server/skill.js

```
/*----------------------------------------------
  読み込み
----------------------------------------------*/

const skillArray = [
  /*  0 */ NormalAttack,
  /*  1 */ Burst,
  /*  2 */ Heal,
  /*  3 */ PoisonAttack,
  /*  4 */ CounterAttack
];

const skillResolver = (skillIds) => {
  // スキル読み込み用の関数
  // スキルIDの配列を受けとってスキルのインスタンス配列を返す
  const skills = [];

  for (let i = 0; i < skillIds.length; i++) {
    const SkillClass = skillArray[skillIds[i]];
    skills.push(new SkillClass());
  }

  return skills;
};

const isActiveSkill = (skillInstance) => {
  // アクティブスキルかどうかを判定する関数
  return skillInstance instanceof ActiveSkill;
};

const isPassiveSkill = (skillInstance) => {
  // パッシブスキルかどうかを判定する関数
  return skillInstance instanceof PassiveSkill;
};

module.exports = {
  skillResolver:  skillResolver,
  isActiveSkill:  isActiveSkill,
  isPassiveSkill: isPassiveSkill
};
```

　次は戦闘システムを変更していきましょう。「backend/server/battle.js」を開きます。まずはユニットクラスのinitを開き以下のように書き換えてください。パッシブスキルが追加されたことに伴い、習得しているスキルをアクティブスキルとパッシブスキルに振り分けるようになっています。

backend/server/battle.js

```
  init(id, battle) { // 初期化処理2
    this.id     = id;
    this.battle = battle;
```

```
  const skills = skill.skillResolver(this.skillIds); // スキルをロード

  this.activeSkills  = []; // スキルをアクティブスキルと
  this.passiveSkills = []; // パッシブスキルに振り分ける
  for (let i = 0; i < skills.length; i++) {
    skills[i].init(this, this.battle);              // スキルを初期化
    if (skill.isActiveSkill(skills[i])) {           // そのスキルがアクティブスキルなら
      this.activeSkills.push(skills[i]);            // アクティブスキル配列に追加
    } else if (skill.isPassiveSkill(skills[i])) { // そのスキルがパッシブスキルなら
      this.passiveSkills.push(skills[i]);           // パッシブスキル配列に追加
    }
  }
}
```

次はパッシブスキルを発動させる処理を実装します。ユニットクラスの適当なところ(適切な場所であればどこでも構いません、actionメソッドの前など)に以下のメソッドを追加してください。dispatchはトリガーIDを受け取り、トリガーIDが一致して発動率チェックもOKだったものを発動させています。

backend/server/battle.js

```
dispatch(triggerId) {
  for (let i = 0; i < this.passiveSkills.length; i++) {
    if (triggerId == this.passiveSkills[i].getTriggerId() && this.passiveSkills[i].isRateCheckOk()) {
      // パッシブスキル配列からトリガーにマッチするものを検索し発動率による条件を満たしたら
      this.battle.log(`${this.name}の${this.passiveSkills[i].name}！`); // 名前、実行するスキル名を表示し
      this.passiveSkills[i].execute(); // 実行
    }
  }
}
```

あとは指定のタイミングでdispatchを呼び出せばパッシブスキルを実行させることができます。ユニットクラスのtryDodgeを以下のように書き換えてください。回避に成功したときトリガーIDが'dodge'のパッシブスキルを発動させるようにしています。

backend/server/battle.js

```
tryDodge(attacker) {
  const dodgeRate = config.basicDodgeRate + this.agi - attacker.dex;
  // 回避率 基礎回避率 + 自身のAGI% - 攻撃者のDEX%
  if (Math.random() * 100 < dodgeRate) { // 回避成功時、メッセージを表示しtrueを返す
    this.battle.log(`  ${this.name}は攻撃を回避した！`);
    this.dispatch('dodge'); // 回避時スキルを発動させる
    return true;
  } else { // 回避失敗時falseを返す
    return false;
  }
}
```

回避時のようなキャラクターに依存したスキルはこんな感じでいいのですが、例えば戦闘開始時のように特定のユニットに依存しない発動条件までもユニットクラスに持たせてしまうと若干処理が不便です。ということで、バトル

クラスからも同じような感じでパッシブスキルを発動できるようにしましょう。バトルクラスの適当なところ(適切な場所であればどこでも構いません、judgeの前など)に以下の2つのメソッドを追加してください。

backend/server/battle.js

```
getLivingUnits() {
  // 生きているユニットを取得
  return this.extractLiving(this.units);
}

dispatch(triggerId) {
  const livingUnits = this.getLivingUnits();
  const livingUnitsSortByAgi = this.sortUnitsByAgi(livingUnits);
  // 生きているユニットをAGI降順に並べ替え

  for (let i = 0; i < livingUnitsSortByAgi.length; i++) {
    livingUnitsSortByAgi[i].dispatch(triggerId);
    // その順番でパッシブスキルを実行
  }
}
```

生きているユニットをAGI降順に並べ替えて順番にユニットのdispatchを呼び出すようにしています。これでバトルクラスの戦闘開始のところでthis.dispatch('battlestart');などのように呼び出すとAGIの高い順で戦闘開始時スキルが実行されるようになります。

4.12.10　ライブラリ化

デバッグのためにあえてそうしたのですが、現状このプログラムは「battle.js」と「skill.js」で完結してしまっており外部から結果を受け取ることができません。これを外部から読み出せるようにしましょう。「backend/server/battle.js」の冒頭部分を以下のように書き換えてください。必要のないライブラリを削っている他、「styling.js」を呼び出しています。

backend/server/battle.js

```
const config  = require('config');
const skill   = require('./skill.js');
const styling = require('./styling.js');

  (省略)
```

また、API宣言の部分を以下のように書き換えます。

backend/server/battle.js

```
  (省略)

/*-----------------------------------------------
  外部読み込み
-----------------------------------------------*/

const fight = (allies, enemies) => {
```

```
  // 味方ユニットの生成
  const allyUnits = [];
  for (let i = 0; allies.length; i++) {
    const ally = new Unit(
      styling.sanitize(allies[i].name),
      allies[i].status,
      allies[i].skillIds
    );
    allyUnits.push(ally);
  }

  // 敵ユニットの生成
  const enemyUnits = [];
  for (let i = 0; enemies.length; i++) {
    const enemy = new Unit(
      styling.sanitize(enemies[i].name),
      enemies[i].status,
      enemies[i].skillIds
    );
    enemyUnits.push(enemy);
  }

  // バトルの生成
  const battle = new Battle(allyUnits, enemyUnits);

  // 戦闘を行い、結果を返す
  return battle.fight();
};

module.exports = {
  fight: fight
};
```

ネットワークからアクセスを受け取って結果を返すのではなく呼び出し元から味方・敵チームの名前・ステータス・スキルIDを受け取るようになっています。このとき、忘れずに名前をサニタイズします。名前は特にサニタイズが行われておらずバトルログを構築している部分でも特にサニタイズは行われていないため、ここでサニタイズしておかないと戦闘ログにXSS脆弱性が発生してしまうためです。

（なお、それならデータベースに保管するときにサニタイズを行っておけばいいのでは？と思われるかもしれませんが、それだと再編集のときなどに管理が複雑になったりNuxt.jsでたくさんv-htmlを使わないといけなくなったりといろいろ面倒が起きるので原則としてサニタイズは出力側で行うようにしています。）

ここのサニタイズは忘れがちなので注意しましょう。この戦闘システムはアイコンやスキル発動時のセリフなどはありませんが、もしそれらを組み込むならそれにもサニタイズが必要です。

次は「backend/server/skill.js」に手を入れます。このファイルはもともとライブラリになっていますが、「battle.js」以外から参照されるにあたって「スキルID配列を受け取ってそれぞれのスキルの説明を返す処理」と「どのスキルが習得可能なのか？を返す処理」があると便利なのでそれらを作っていきます。

まずはスキルの説明を返す処理です。まずはSkillEffectクラスの適当なところ（適切な場所であればどこでも構いません、末尾など）に以下のメソッドを追加します。それぞれのスキルパーツに設定されたnameからスキル説明を構築して返します。

```
backend/server/skill.js
getDescription() { // スキルの説明を返すメソッド
  let s = '';

  if (this.condition) {
    s += this.condition.name;
  }

  s += this.target.name + ':';
  s += this.elements.map(el => el.name).join('＆');

  return s;
}
```

次にSkillクラスの適当なところ(適切な場所であればどこでも構いません、末尾など)に以下のメソッドを追加します。スキル説明を返すメソッドになっているのですが、これの実体はActiveSkillクラスやPassiveSkillクラスに実装します。なので処理の内容はありません。どんなメソッドになっていてどんなデータが返るのか(どんなメソッドを実装してどんなデータを返せばいいのか)を分かりやすくするためだけに置いてあります。

```
backend/server/skill.js
getDescription() {
  return {
    type: '',
    name: this.name,
    cond: '',
    desc: ''
  }
}
```

次はActiveSkillにスキル説明を返すメソッドの実体を作ります。適当なところ(適切な場所であればどこでも構いません、末尾など)に以下のメソッドを追加します。先程追加したSkillEffectクラスから説明を呼び出している他、スキル分類とスキル名やスキル発動条件、SPコストも返すようにしています。

```
backend/server/skill.js
getDescription() {
  return {
    type: 'Active',
    name: this.name,
    cond: `${this.cost}SP / ${this.condition.name}`,
    desc: this.skillEffectChain.map(se => se.getDescription()).join('＋')
  }
}
```

PassiveSkillクラスにも同様に実装します。ActiveSkillとは違ってcondに入るのは発動条件ではなく発動タイミングになります。

backend/server/skill.js

```
getDescription() {
  return {
    type: 'Passive',
    name: this.name,
    cond: this.trigger.name,
    desc: this.skillEffectChain.map(se => se.getDescription()).join('+')
  }
}
```

次は習得可能なスキルIDの配列を返す関数を作ります。それぞれのスキルは能力値によって習得可能かどうかが決まるようにしましょう。ここではこれを実装するためにそれぞれのクラスにisAvailableという静的メソッドを作ります。

これはatk, dex, mnd, agi, defのプロパティを持ったステータスオブジェクトを受け取って習得可能かどうかをtrue/falseで返す静的メソッドになります。まずはSkillクラスの適当なところ（適切な場所であればどこでも構いません、末尾など）に静的メソッドisAvailableを以下のように追加します。機能の実体はないですが実装するべきメソッドを表すために追加しています。

backend/server/skill.js

```
static isAvailable(status) {
  return false;
}
```

後は実際のクラスに判定を行う処理を書いていきます。それぞれのクラスを以下のように書き換えてください。バーストはATK20以上、ヒールはMND20以上、ポイズンはDEX20以上、反撃はAGI20以上で習得可能としてみましょう。以下のようになります。なお、通常攻撃はデフォルトで利用可能なだけで習得するものではないので常にfalseが返るようにしています。

backend/server/skill.js

```
class NormalAttack extends ActiveSkill {
  constructor() {
    super();

    this.name = '通常攻撃';
    this.cost = 0;
    this.condition = new Always();
    this.skillEffectChain = [
      new SkillEffect(new SomeEnemyTarget(1), true, [new AttackElement(50)])
      // 【SP0：通常時】敵：攻撃（回避可能、攻撃の威力50）
    ];
  }

  isCostOk() { // SPが足りているかどうか、通常攻撃は必ず実行可能とする
    return true;
  }
```

```
  isExecutable() { // 実行条件、通常攻撃は必ず実行可能とする
    return true;
  }

  static isAvailable(status) { // デフォルトで利用可能なだけで習得するものではないのでfalse
    return false;
  }
}

class Burst extends ActiveSkill {
  constructor() {
    super();

    this.name = 'バースト';
    this.cost = 40;
    this.condition = new PerActions(4);
    this.skillEffectChain = [
      new SkillEffect(new SomeEnemyTarget(2), true, [new AttackElement(80)]),
      new SkillEffect(new SomeEnemyTarget(1), true, [new AttackElement(100)], new SpRemain(90))
      // 【SP40：4行動毎】敵2：攻撃（回避可能、攻撃の威力80）＋自身のSPが90%以上なら敵：攻撃（回避可能、攻撃の威力100）
    ];
  }

  static isAvailable(status) {
    return 20 <= status.atk;
  }
}

class Heal extends ActiveSkill {
  constructor() {
    super();

    this.name = 'ヒール';
    this.cost = 20;
    this.condition = new WeakenedAllyExists();
    this.skillEffectChain = [
      new SkillEffect(new SomeWeakenedAllyTarget(1), false, [new HealElement(40)])
      // 【SP20：味方重傷】弱味：HP回復
    ];
  }

  static isAvailable(status) {
    return 20 <= status.mnd;
  }
}

class PoisonAttack extends ActiveSkill {
  constructor() {
    super();

    this.name = 'ポイズン';
    this.cost = 30;
    this.condition = new PerActions(4);
    this.skillEffectChain = [
      new SkillEffect(new SomeEnemyTarget(1), true, [new AttackElement(30), new PoisonElement(3)])
      // 【SP30：4行動毎】敵：攻撃＆毒
    ];
```

```
  }

  static isAvailable(status) {
    return 20 <= status.dex;
  }
}

class CounterAttack extends PassiveSkill {
  constructor() {
    super();

    this.name = '反撃';
    this.rate = 30;
    this.trigger = new Dodge();
    this.skillEffectChain = [
      new SkillEffect(new SomeEnemyTarget(1), true, [new AttackElement(50)])
      // 【回避時】敵：攻撃
    ]
  }

  static isAvailable(status) {
    return 20 <= status.agi;
  }
}
```

最後に追加していったメソッドを使って外部から参照できる関数を作ります。「読み込み」の部分を以下のように書き換えてください。「スキルID配列を受け取ってそれぞれのスキルの説明を返す処理」がgetDescriptions、「どのスキルが習得可能なのか?を返す処理」がgetAvailableSkillIdsになります。なお外部から呼び出すときにnull（スキル未設定）が含まれていた場合もそのまま処理できたほうが便利なのでskillResolverやgetDescriptionはそれに対応してあります。

backend/server/skill.js

```
/*----------------------------------------------
  読み込み
----------------------------------------------*/

const skillArray = [
  /*  0 */ NormalAttack,
  /*  1 */ Burst,
  /*  2 */ Heal,
  /*  3 */ PoisonAttack,
  /*  4 */ CounterAttack
];

const skillResolver = (skillIds) => {
  // スキル読み込み用の関数
  // スキルIDの配列を受けとってスキルのインスタンス配列を返す
  // 外部から呼び出すときnullをスキップできたほうが便利なのでスキップさせる
  const skills = [];

  for (let i = 0; i < skillIds.length; i++) {
    if (skillIds[i] != null) {
      const SkillClass = skillArray[skillIds[i]];
```

```
      skills.push(new SkillClass());
    }
  }

  return skills;
};

const isActiveSkill = (skillInstance) => {
  // アクティブスキルかどうかを判定する関数
  return skillInstance instanceof ActiveSkill;
};

const isPassiveSkill = (skillInstance) => {
  // パッシブスキルかどうかを判定する関数
  return skillInstance instanceof PassiveSkill;
};

const getDescriptions = (skillIds) => {
  // スキル説明文読み込み用の関数
  // スキルID配列を受け取ってスキル説明オブジェクトを返す
  const skillDescriptions = [];

  for (let i = 0; i < skillIds.length; i++) {
    if (skillIds[i] == null) {
      // スキルIDがnullであれば該当の説明オブジェクトはnullにする
      // （外部から参照するときに扱いやすくするため）
      skillDescriptions.push(null);
    } else {
      const SkillClass  = skillArray[skillIds[i]];
      const instance    = new SkillClass();
      const description = instance.getDescription();
      description.id    = skillIds[i]; // ID情報も付加しておく
      skillDescriptions.push(description);
    }
  }

  return skillDescriptions;
};

const getAvailableSkillIds = (status) => {
  // 習得可能なスキルを取得する関数
  // ステータスを受け取って習得可能なスキルのIDの配列を返す
  const skillIds = [];

  for (let i = 0; i < skillArray.length; i++) {
    const SkillClass = skillArray[i];
    if (SkillClass.isAvailable(status)) {
      skillIds.push(i);
    }
  }

  return skillIds;
};

module.exports = {
  skillResolver:        skillResolver,
  isActiveSkill:        isActiveSkill,
```

```
  isPassiveSkill:        isPassiveSkill,
  getDescriptions:       getDescriptions,
  getAvailableSkillIds:  getAvailableSkillIds
};
```

これでライブラリ化は完了です。これ以降「npm run battletest」を実行することはないので「backend/package.json」を開き、"scripts"の内容を以下のように書き換えて保存してください。

backend/package.json

```
（省略）

  "scripts": {
    "dev": "nodemon server/index.js --watch config --watch server",
    "start": "node server/index.js",
    "init": "node server/initialize.js"
  },

（省略）
```

保存したらバックエンド側のコンソールにCtrl+Cを入力してプログラムを停止し「npm run dev」を実行してバックエンドを実行しましょう。

4.12.11　battle.jsの最終形

あとはHATE増などスキルから呼び出すためのいろいろな処理を追加し、出力されるログの形式を「
」で区切っただけのテキストログではなくしっかりとHTML形式で出力できるようにして戦闘結果が勝利・敗北・引き分けのどれなのかも返すようにすれば「battle.js」は完成です。処理の追加については今まで解説した内容が全てですし、その他についても特に複雑なことはしていないので解説はしません。最終的な「battle.js」は以下のようになります。以下の内容で「backend/server/battle.js」を上書きして保存してください。

backend/server/battle.js

```
const config  = require('config');
const skill   = require('./skill.js');
const styling = require('./styling.js');

/*------------------------------------------------
  クラス
------------------------------------------------*/

class Unit {
  constructor(name, status, skillIds) { // 初期化処理1
    this.name = name;

    this.rawStatus = { // 生ステータス
      atk: status.atk,
      dex: status.dex,
      mnd: status.mnd,
      agi: status.agi,
      def: status.def
```

```
    };

    this.effect = { // 状態異常の残りターン数
      poison: 0
    };

    this.mhp =
      this.atk * config.hpRate.atk +
      this.dex * config.hpRate.dex +
      this.mnd * config.hpRate.mnd +
      this.agi * config.hpRate.agi +
      this.def * config.hpRate.def +
      config.guaranteedHp; // 最大HP
    this.hp = this.mhp;     // 現在HP

    this.msp =
      this.atk * config.spRate.atk +
      this.dex * config.spRate.dex +
      this.mnd * config.spRate.mnd +
      this.agi * config.spRate.agi +
      this.def * config.spRate.def +
      config.guaranteedSp; // 最大SP
    this.sp = this.msp;     // 現在SP

    this.skillIds = skillIds; // スキルID配列
    this.skillIds.push(0);    // スキルID配列の末尾に0（通常攻撃）を追加

    this.acted       = false; // 行動済ステータス（false: 未行動 true: 行動済み）
    this.living      = true;  // 生存ステータス（false: 戦闘不能 true: 生存）
    this.actionCount = 0;     // 行動回数カウント
    this.hates       = [];    // ヘイト量
  }

  init(id, battle) { // 初期化処理2
    this.id     = id;
    this.battle = battle;

    const skills = skill.skillResolver(this.skillIds); // スキルをロード

    this.activeSkills  = []; // スキルをアクティブスキルと
    this.passiveSkills = []; // パッシブスキルに振り分ける
    for (let i = 0; i < skills.length; i++) {
      skills[i].init(this, this.battle);            // スキルを初期化
      if (skill.isActiveSkill(skills[i])) {         // そのスキルがアクティブスキルなら
        this.activeSkills.push(skills[i]);          // アクティブスキル配列に追加
      } else if (skill.isPassiveSkill(skills[i])) { // そのスキルがパッシブスキルなら
        this.passiveSkills.push(skills[i]);         // パッシブスキル配列に追加
      }
    }
  }

  isLiving() { // 生きているかどうか
    return this.living;
  }

  isActable() { // 行動可能かどうか、生きていて行動済ではない
    return this.isLiving() && !this.acted;
```

```
  }

  hate(hate, target) { // 対象へのヘイト量を蓄積する
    this.hates[target.id] = (this.hates[target.id] || 0) + hate;
  }

  consumeSp(cost) { // SP消費処理
    this.sp -= cost;
  }

  tryDodge(attacker) {
    const dodgeRate = config.basicDodgeRate + this.agi - attacker.dex;
    // 回避率 基礎回避率 + 自身のAGI% - 攻撃者のDEX%
    if (Math.random() * 100 < dodgeRate) { // 回避成功時、メッセージを表示しtrueを返す
      this.battle.log(`<div class="result">${this.name}は攻撃を回避した！</div>`);
      this.dispatch('dodge'); // 回避時スキルを発動させる
      return true;
    } else { // 回避失敗時falseを返す
      return false;
    }
  }

  gainAttack(potency, attacker) { // 威力と攻撃者を受け取る
    let basicDamage = 10 + attacker.atk; // 基礎ダメージ 攻撃者のATK+10
    let damage = Math.floor((basicDamage * potency / 50) * (0.8 + Math.random() * 0.4));
    // ダメージ (基礎ダメージ×威力÷50) × 0.8～1.2

    this.battle.log('<div class="result">');
    if (Math.random() * 100 < attacker.dex) { // 攻撃者のDEX%の確率でクリティカル
      this.battle.log(`クリティカル！ `);
      damage *= 2; // クリティカル時はダメージを2倍に
    }

    let damageRank = '';
    if (100 <= damage) {
      damageRank = ' damage-gte100';
    } else if (50 <= damage) {
      damageRank = ' damage-gte50';
    }

    this.hp -= damage;
    this.battle.log(`${this.name}は<span class="damage${damageRank}">${damage}</span>点のダメージを
受けた！(現在HP：${this.hp})</div>`);
    this.hate(damage * attacker.def, attacker); // 「受けたダメージ×攻撃者のDEF」分ヘイトを蓄積する
    this.dispatch('attacked'); // 被攻撃時スキルを発動させる
  }

  gainHate(potency, attacker) {
    const basicHate = (10 + attacker.atk); // 基礎ヘイト 攻撃者のDEF+10
    const hate = Math.floor((basicHate * potency / 50) * (0.8 + Math.random() * 0.4)) * attacker.d
ef;
    // ヘイト (基礎ヘイト×威力÷50) × 0.8～1.2×攻撃者のDEF

    this.battle.log(`<div class="result">${this.name}は${attacker.name}へのヘイトが高まった！</div>
`);
    this.hate(hate, attacker); // ヘイトを蓄積する
  }
```

```
  gainHeal(potency, healer) {
    const basicHeal = (10 + healer.mnd); // 基礎回復量 回復者のMND+10
    const heal = Math.floor((basicHeal * potency / 50) * (0.8 + Math.random() * 0.4));
    // 回復量 (基礎回復量×威力÷50) × 0.8～1.2

    const formarHp = this.hp; // 回復前のHP
    this.hp += heal; // 回復する
    if (this.mhp < this.hp) {
      this.hp = this.mhp; // MHPを超えたらMHPにする
    }
    const actualHealed = this.hp - formarHp; // 実際の回復量

    this.battle.log(`<div class="result">${this.name}はHPが<span class="heal">${actualHealed}</spa
n>点回復した！(現在HP：${this.hp})</div>`);
  }

  gainAgi(potency) { // AGIを増やす効果
    this.battle.log(`<div class="result">${this.name}のAGIが増加した！</div>`);
    this.rawStatus.agi += potency;
  }

  gainPoison(turn) {
    this.effect.poison += turn;
    this.battle.log(`<div class="result">${this.name}は毒を${turn}ターン分受けた！</div>`);
  }

  dispatch(triggerId) {
    for (let i = 0; i < this.passiveSkills.length; i++) {
      if (triggerId == this.passiveSkills[i].getTriggerId() && this.passiveSkills[i].isRateCheckOk
()) {
        // パッシブスキル配列からトリガーにマッチするものを検索し発動率による条件を満たしたら
        this.battle.log('<section class="action">');
        this.battle.log(`${this.name}の<span class="skill-name">${this.passiveSkills[i].name}！</s
pan>`); // 名前、実行するスキル名を表示し
        this.passiveSkills[i].execute(); // 実行
        this.battle.log('</section>');
      }
    }
  }

  action() { // 1回の行動を行う
    this.actionCount++; // 行動回数カウントを+1

    // アクティブスキル配列の上から順番に実行判断
    for (let i = 0; i < this.activeSkills.length; i++) {
      if (this.activeSkills[i].isCostOk() && this.activeSkills[i].isExecutable()) {
        // 実行コストが足りていて実行可能なら
        this.battle.log('<section class="action">');
        this.battle.log(`${this.name}の<span class="skill-name">${this.activeSkills[i].name}！</sp
an>(${this.actionCount}行動目)`);
        // 名前、実行するスキル名、行動回数を表示し
        this.activeSkills[i].execute(); // 実行
        this.activeSkills[i].afterExecute();  // 実行後処理を行う
        this.battle.log('</section>');
        return;
        // 通常攻撃は必ず末尾に登録されておりかつ実行可能になっているので
```

```
        // 他に実行できるスキルがなければ通常攻撃になる
      }
    }
  }

  act() { // ターン行動を行う
    this.action();
    if (Math.random() * 100 < this.agi) { // AGI%の確率で連続行動
      this.battle.log('<div class="action-twice">連続行動！</div>');
      this.action();
    }
    this.acted = true; // 行動済みフラグをONに
  }

  setup() { // セットアップ処理 行動済ステータスを未行動（false）に
    this.acted = false;
  }

  cleanup() {
    if (this.living && this.effect.poison) {
      const poisonDamage = Math.floor(Math.random()* 11) + 5;
      // 毒はクリーンアップ時に5～15のダメージを受ける
      this.hp -= poisonDamage;
      this.effect.poison--; // ターン数を減少
      if (this.effect.poison) { // まだターン数が残っていれば
        this.battle.log(`<div class="result">${this.name}は毒により${poisonDamage}のダメージを受け
た！(残りターン：${this.effect.poison})</div>`);
      } else { // 治癒したら
        this.battle.log(`<div class="result">${this.name}は毒により${poisonDamage}のダメージを受け
た！</div>`);
        this.battle.log(`<div class="result">${this.name}の毒は治った！</div>`);
      }
    }

    if (this.living && this.hp <= 0) { // クリーンアップ時生きていてHP0以下なら戦闘不能に
      this.living = false;
      this.battle.log(`<div class="result">${this.name}は倒れた……</div>`);
    }
  }

  get atk() {
    return this.rawStatus.atk * (this.effect.poison ? 0.8 : 1); // 毒効果中は0.8倍になる
  }

  get dex() {
    return this.rawStatus.dex * (this.effect.poison ? 0.8 : 1); // 毒効果中は0.8倍になる
  }

  get mnd() {
    return this.rawStatus.mnd;
  }

  get agi() {
    return this.rawStatus.agi * (this.effect.poison ? 0.8 : 1); // 毒効果中は0.8倍になる
  }

  get def() {
```

```
    return this.rawStatus.def;
  }
}

class Battle {
  constructor(allies, enemies) {
    this.allies  = allies;
    this.enemies = enemies;
    this.units   = allies.concat(enemies); // 敵味方全体

    this.allyIds  = [];
    this.enemyIds = [];

    this.unitIdDealer = 0;
    for (let i = 0; i < allies.length; i++) { // 味方チームに一意のIDを割り振る
      this.allies[i].init(this.unitIdDealer, this);
      this.allyIds.push(this.unitIdDealer);
      this.unitIdDealer++;
    }
    for (let i = 0; i < enemies.length; i++) { // 敵チームに一意のIDを割り振る
      this.enemies[i].init(this.unitIdDealer, this);
      this.enemyIds.push(this.unitIdDealer);
      this.unitIdDealer++;
    }

    this.round     = 0;
    this.battleLog = '';
  }

  log(str) { // バトルログを追加
    this.battleLog += str;
  }

  extractLiving(units) {
    // 受け取ったユニット配列から生きているユニットだけを抽出して返す
    const livingUnits = [];

    for (let i = 0; i < units.length ;i++) {
      if (units[i].isLiving()) {
        livingUnits.push(units[i]);
      }
    }

    return livingUnits;
  }

  extractActable(units) {
    // 受け取ったユニット配列から行動可能なユニットだけを抽出して返す
    const actableUnits = [];

    for (let i = 0; i < units.length ;i++) {
      if (units[i].isActable()) {
        actableUnits.push(units[i]);
      }
    }

    return actableUnits;
```

```
  }

  sortUnitsByAgi(units) {
    // 受け取ったユニット配列をAGI降順にソート
    units.sort((a, b) => {
      return b.agi - a.agi;
    });

    return units;
  }

  shuffleUnits(units) {
    // 受け取ったユニットをランダムに並べ替える
    // (Fisher-Yatesシャッフル)
    for (let i = units.length - 1; i > 0; i--){
      const target = Math.floor(Math.random() * (i + 1));
      [units[i], units[target]] = [units[target], units[i]];
    }

    return units;
  }

  getEnemies(unitId) {
    // IDのユニットから見た敵側チームを取得
    return this.allyIds.includes(unitId) ? this.enemies : this.allies;
  }

  getLivingEnemies(unitId) {
    // IDのユニットから見た生きている敵側チームを取得
    return this.extractLiving(this.getEnemies(unitId));
  }

  getAllies(unitId) {
    // IDのユニットから見た味方側チームを取得
    return this.allyIds.includes(unitId) ? this.allies : this.enemies;
  }

  getLivingAllies(unitId) {
    // IDのユニットから見た生きている味方側チームを取得
    return this.extractLiving(this.getAllies(unitId));
  }

  getLivingUnits() {
    // 生きているユニットを取得
    return this.extractLiving(this.units);
  }

  dispatch(triggerId) {
    const livingUnits = this.getLivingUnits();
    const livingUnitsSortByAgi = this.sortUnitsByAgi(livingUnits);
    // 生きているユニットをAGI降順に並べ替え

    for (let i = 0; i < livingUnitsSortByAgi.length; i++) {
      livingUnitsSortByAgi[i].dispatch(triggerId);
      // その順番でパッシブスキルを実行
    }
  }
```

```
  judge() {
    const livingAllies  = this.extractLiving(this.allies); // 生き残っている味方チームを取得
    const livingEnemies = this.extractLiving(this.enemies); // 生き残っている敵チームを取得

    if (livingAllies.length && !livingEnemies.length) {
      // 味方が生き残っていて敵が全員戦闘不能 → 勝利
      return 'win';
    } else if (!livingAllies.length && livingEnemies.length) {
      // 味方が全員戦闘不能で敵が生き残っている → 敗北
      return 'lose';
    } else if (!livingAllies.length && !livingEnemies.length) {
      // 味方も敵も全員戦闘不能 → 引き分け
      return 'even';
    } else if (config.battleMaxRound <= this.round) {
      // どちらも生き残っているが現在ラウンドが最大ラウンド以上 → 引き分け
      return 'even';
    } else {
      // そのどれでもない → 戦闘継続
      return 'continue';
    }
  }

  fight() {
    let result = '';

    this.log('<section class="battle">');

    this.log('<section class="battle-start">');
    this.log('<div class="battle-start-call">BATTLE START</div>');
    this.log(`<section class="actions">`);
    this.dispatch('battlestart'); // 戦闘開始時スキルを発動させる
    this.log(`</section>`);
    this.log('</section>');

    round_loop:
    while(true) {
      this.round++;

      this.log('<section class="round">');
      this.log(`<div class="round-start">Round <span class="round-count">${this.round}</span></div
>`);

      this.log('<section class="statuses">');

      for (let i = 0; i < 2; i++) {
        // チームごとにセットアップ処理とステータス表示を行う
        this.log('<section class="team">');
        const targetTeam = i ? this.enemies : this.allies;
        for (let j = 0; j < targetTeam.length; j++) {
          targetTeam[j].setup(); // セットアップ処理
          if (targetTeam[j].isLiving()) { // 生きている場合にのみステータスを表示
            this.log('<section class="unit">');
            this.log(`<div class="unit-name">${targetTeam[j].name}</div>`);

            this.log('<div class="statusbars">');
```

```
          this.log('<div class="statusbar">');
          this.log('<div class="statusbar-desc">');
          this.log('<div class="statusbar-key">HP</div>');
          this.log(`<div class="statusbar-value">${targetTeam[j].hp} / ${targetTeam[j].mhp}</div
>`);
          this.log('</div>');
          this.log('<div class="gauge-wrapper">');
          this.log(`<div class="gauge" style="width: ${targetTeam[j].hp / targetTeam[j].mhp * 10
0 }%;"></div>`);
          this.log('</div>');
          this.log('</div>');

          this.log('<div class="statusbar">');
          this.log('<div class="statusbar-desc">');
          this.log('<div class="statusbar-key">SP</div>');
          this.log(`<div class="statusbar-value">${targetTeam[j].sp} / ${targetTeam[j].msp}</div
>`);
          this.log('</div>');
          this.log('<div class="gauge-wrapper">');
          this.log(`<div class="gauge" style="width: ${targetTeam[j].sp / targetTeam[j].msp * 10
0 }%;"></div>`);
          this.log('</div>');
          this.log('</div>');

          this.log('</div>');

          this.log('</section>');
        }
      }
      this.log('</section>');
    }

    this.log('</section>');
    this.log('<section class="turns">');

    turn_loop:
    while(true) {
      // 全ユニットから行動可能なユニットを抽出
      const actableUnits = this.extractActable(this.units);

      if (actableUnits.length) { // 行動可能な者がいれば
        this.log('<section class="turn">');
        const actor = this.sortUnitsByAgi(actableUnits)[0];
        // それをAGI降順に並べ替え最もAGIが速いものを行動者とし
        this.log(`<div class="actor">${actor.name}のターン！</div>`);
        this.log(`<section class="actions">`);
        actor.act(); // 行動実行
        this.log(`</section>`);
        this.log('</section>');
      } else {            // 行動可能な者がいなければ
        break turn_loop; // そのラウンドは終了処理へ
      }
    }
    this.log('</section>');
    this.log('<section class="clean-up">');

    for (let i = 0; i < this.units.length; i++) { // クリーンアップ処理
```

```
        this.units[i].cleanup();
      }

      this.log('</section>');
      this.log('</section>');

      const judge = this.judge(); // 戦闘判定
      if (judge != 'continue') {
        // 判定結果が戦闘継続でない場合戦闘終了
        this.log('<section class="battle-result">');

        if (judge == 'win') { // ジャッジ結果が勝利のとき
          result = 'win';
          this.log('戦闘に勝利した！');
        } else if (judge == 'lose') { // ジャッジ結果が敗北のとき
          result = 'lose';
          this.log('戦闘に敗北した……');
        } else { // そのどちらでもない（引き分け）とき
          result = 'even';
          this.log('決着がつかなかった……');
        }

        this.log('</section>');

        break round_loop; // 戦闘ループから抜ける
      }
      // 判定結果が戦闘継続かつ最大ラウンドに到達していない場合次のラウンドへ
    }

    // 戦闘ログを返す
    return {
      result: result, // win = 勝利, lose = 敗北, even = 引き分け
      log:    this.battleLog
    };
  }
}

/*------------------------------------------------
  外部読み込み
------------------------------------------------*/

const fight = (allies, enemies) => {

  // 味方ユニットの生成
  const allyUnits = [];
  for (let i = 0; i < allies.length; i++) {
    const ally = new Unit(
      styling.sanitize(allies[i].name),
      allies[i].status,
      allies[i].skillIds
    );
    allyUnits.push(ally);
  }

  // 敵ユニットの生成
  const enemyUnits = [];
  for (let i = 0; i < enemies.length; i++) {
```

```
    const enemy = new Unit(
      styling.sanitize(enemies[i].name),
      enemies[i].status,
      enemies[i].skillIds
    );
    enemyUnits.push(enemy);
  }

  // バトルの生成
  const battle = new Battle(allyUnits, enemyUnits);
  const result = battle.fight();

  // 戦闘を行い、結果を返す
  return result;
};

module.exports = {
  fight: fight
};
```

4.12.12 skill.jsの最終形

「skill.js」については色々なスキルパーツやスキルを実装するだけです。ここでは通常攻撃に加えて以下の10個のスキルを作ります。

アタック
アクティブスキル
30SP / 通常時
敵:攻撃
初期習得

フレア
アクティブスキル
50SP / 9行動毎
敵:攻撃＋自身のSPが30%以上なら敵全:攻撃
ATK20以上で習得

双撃
アクティブスキル
40SP / 3行動毎
敵2:攻撃
初期習得

ポイズン
アクティブスキル
40SP / 4行動毎
敵:攻撃&毒
DEX20以上で習得

ヒール
アクティブスキル
20SP / 味方重傷
弱味:HP回復
初期習得

エリアヒール
アクティブスキル
50SP / 味方重傷
味全:HP回復
MND20以上で習得

シフトアップ
パッシブスキル
戦闘開始時

反撃
パッシブスキル
回避時

自:AGI増
初期習得

敵:攻撃
AGI20以上で習得

アルティメイタム

パッシブスキル
戦闘開始時
敵全:HATE増
初期習得

自己修復

パッシブスキル
被攻撃時
自:HP回復
DEF20以上で習得

最終的には「skill.js」は以下のようになります。「backend/server/skill.js」を上書きして保存してください。

backend/server/skill.js

```
/*-----------------------------------------------
  ターゲットクラス
-----------------------------------------------*/

class Target {
  constructor() {
    this.name = ''; // ターゲット名
  }

  init(unit, battle) { // 初期化処理 スキル使用者とバトル環境を受け取る
    this.unit   = unit;
    this.battle = battle;
  }

  resolve() { // ターゲットになるキャラクターの配列を返す
    return [];
  }
}


class OwnTarget extends Target {
  constructor() {
    super();

    this.name = '自';
  }

  resolve() {
    return [this.unit]; // 自身をターゲットとして返す  返す内容は配列でないといけないことに注意
  }
}

class SomeEnemyTarget extends Target {
  // ヘイトの高い順に敵を指定数ターゲットする、同値はランダム
  constructor(number) {
    super();

    this.name   = 1 < number ? `敵${number}` : '敵';
    this.number = number;
  }
```

```
  resolve() {
    const livingEnemies = this.battle.getLivingEnemies(this.unit.id); // まず生きている敵を取得し
    const shuffledLivingEnemies = this.battle.shuffleUnits(livingEnemies); // ランダムに並べ替え

    const livingEnemiesSortedByHate = shuffledLivingEnemies.sort((a, b) => {
      // 次にヘイトの高い順に並び替える
      const hateA = this.unit.hates[a.id] || 0; // undefinedだったら0扱い
      const hateB = this.unit.hates[b.id] || 0;
      return hateB - hateA;
    });

    return livingEnemiesSortedByHate.slice(0, this.number);
    // 指定数だけ切り出してターゲットとして返す
    // 生きている敵が指定数より少ないときには指定数よりターゲット数が少なくなる
  }
}

class SomeWeakenedAllyTarget extends Target {
  constructor(number) {
    super();

    this.name   = 1 < number ? `弱味${number}` : '弱味';
    this.number = number;
  }

  resolve() {
    // MHPに対するHPの割合が低い順に味方を指定数ターゲットする、同値はランダム
    const livingAllies = this.battle.getLivingAllies(this.unit.id); // まず生きている味方を取得
    const shuffledLivingAllies = this.battle.shuffleUnits(livingAllies); // ランダムに並べ替え

    const livingAlliesSortedByHpRate = shuffledLivingAllies.sort((a, b) => {
      // 次にHP割合の低い順に並び替える
      const hpRateA = a.hp / a.mhp;
      const hpRateB = b.hp / b.mhp;
      return hpRateA - hpRateB;
    });

    return livingAlliesSortedByHpRate.slice(0, this.number);
    // 指定数だけ切り出してターゲットとして返す
    // 生きている味方が指定数より少ないときには指定数よりターゲット数が少なくなる
  }
}

class AllEnemyTarget extends Target {
  constructor() {
    super();

    this.name = '敵全';
  }

  resolve() {
    return this.battle.getLivingEnemies(this.unit.id);
  }
}

class AllAllyTarget extends Target {
```

```
  constructor() {
    super();

    this.name = '味全';
  }

  resolve() {
    return this.battle.getLivingAllies(this.unit.id);
  }
}

/*-------------------------------------------------
  エレメントクラス
-----------------------------------------------*/

class Element {
  constructor() {
    this.name = ''; // 効果要素名
  }

  init(unit, battle) { // 初期化処理 スキル使用者とバトル環境を受け取る
    this.unit   = unit;
    this.battle = battle;
  }

  resolve(target) { // ターゲットに対して効果を発動させる

  }
}

class AttackElement extends Element {
  // 対象に攻撃を行うスキルエレメント
  constructor(potency) {
    super();

    this.name    = '攻撃'; // 効果要素名
    this.potency = potency; // 威力を受け取りその威力を設定
  };

  resolve(target) { // 効果を発動
    target.gainAttack(this.potency, this.unit); // 威力と攻撃者でターゲットの被攻撃メソッドを呼び出す
  }
}

class HateElement extends Element {
  constructor(potency) {
    super();

    this.name    = 'HATE増';
    this.potency = potency;
  }

  resolve(target) {
    target.gainHate(this.potency, this.unit); // 威力と攻撃者でターゲットのヘイト増加メソッドを呼び出す
  }
}
```

```
class HealElement extends Element {
  constructor(potency) {
    super();

    this.name    = 'HP回復';
    this.potency = potency;
  }

  resolve(target) {
    target.gainHeal(this.potency, this.unit); // 威力と回復者でターゲットの被回復メソッドを呼び出す
  }
}

class GainAgiElement extends Element {
  constructor(potency) {
    super();

    this.name    = 'AGI増';
    this.potency = potency;
  }

  resolve(target) {
    target.gainAgi(this.potency);
  }
}

class PoisonElement extends Element {
  constructor(turn) {
    super();

    this.name = '毒';
    this.turn = turn;
  }

  resolve(target) {
    target.gainPoison(this.turn);
  }
}

/*------------------------------------------------
  スキルエフェクト発動条件クラス
------------------------------------------------*/

class SkillEffectCondition {
  constructor() {
    this.name = ''; // スキルエフェクト発動条件名
  }

  init (unit, battle) {
    this.unit   = unit;
    this.battle = battle;
  }

  resolve() {
    return false;
  }
}
```

```
class SpRemain extends SkillEffectCondition {
  // SPがMSPの指定の%以上なら
  constructor(percentage) {
    super();

    this.name       = `自身のSPが${percentage}%以上なら`;
    this.percentage = percentage;
  }

  resolve() {
    return this.percentage <= (this.unit.sp / this.unit.msp) * 100;
  }
}

/*------------------------------------------------
  スキルエフェクトクラス
------------------------------------------------*/

class SkillEffect {
  constructor(target, dodgeable, elements, condition) {
    this.target    = target;    // ターゲット情報
    this.dodgeable = dodgeable; // 回避可能かどうか（true: 回避可能, false: 回避不可）
    this.elements  = elements;  // スキルエフェクト要素
    this.condition = condition; // スキルエフェクト発動条件（指定しない場合特に発動条件はない）
  }

  init(unit, battle) { // 初期化処理 スキル使用者とバトル環境を受け取り
    this.unit   = unit;
    this.battle = battle;
    this.target.init(unit, battle); // ターゲットと

    if (this.condition) {
      this.condition.init(unit, battle); // （あれば）スキルエフェクト発動条件と
    }

    for (let i = 0; i < this.elements.length; i++) {
      this.elements[i].init(unit, battle); // エフェクト要素を初期化
    }
  }

  resolve() { // スキルエフェクト発動
    if (this.condition && !this.condition.resolve()) {
      // スキルエフェクト発動条件があり、発動条件を満たしていないなら
      // 特に何もせず処理を終える
      return;
    }

    const targets = this.target.resolve(); // ターゲットになるキャラクターの配列を取得

    for (let i = 0; i < targets.length; i++) { // ターゲットごとに処理を繰り返す
      if (this.dodgeable && targets[i].tryDodge(this.unit)) {
        // 回避可能なスキルエフェクトなら回避を試み、成功なら
        continue; // 処理をスキップ、次のターゲットへ
      }

      // 回避に失敗したらスキルエフェクト要素たちを発動させる
```

```
      for (let j = 0; j < this.elements.length; j++) {
        this.elements[j].resolve(targets[i]); // 実行
      }
    }
  }

  getDescription() { // スキルの説明を返すメソッド
    let s = '';

    if (this.condition) {
      s += this.condition.name;
    }

    s += this.target.name + ':';
    s += this.elements.map(el => el.name).join('＆');

    return s;
  }
}

/*-----------------------------------------------
  トリガークラス
-----------------------------------------------*/

class Trigger {
  constructor() {
    this.name = ''; // 表示されるトリガー名
    this.id   = ''; // 内部的なトリガーID
  }
}

class BattleStart extends Trigger { // 戦闘開始時
  constructor() {
    super();

    this.name = '戦闘開始時';
    this.id   = 'battlestart';
  }
}

class Dodge extends Trigger { // 回避したとき
  constructor() {
    super();

    this.name = '回避時';
    this.id   = 'dodge';
  }
}

class Attacked extends Trigger { // 攻撃を受けたとき
  constructor() {
    super();

    this.name = '被攻撃時';
    this.id   = 'attacked';
  }
}
```

```
/*---------------------------------------------
  スキル発動条件
---------------------------------------------*/

class Condition {
  constructor() {
    this.name = ''; // 発動条件名
  }

  init (unit, battle) {
    this.unit   = unit;
    this.battle = battle;
  }

  resolve() {
    return false;
  }
}

class Always extends Condition {
  constructor() {
    super();

    this.name = '通常時';
  }

  resolve() { // 常に発動可能
    return true;
  }
}

class PerActions extends Condition {
  constructor(perActions) {
    super();

    this.name       = `${perActions}行動毎`;
    this.perActions = perActions;
  }

  resolve() { //
    return !(this.unit.actionCount % this.perActions); // 指定の行動ごと
  }
}

class WeakenedAllyExists extends Condition {
  constructor() {
    super();

    this.name = '味方重傷';
  }

  resolve() { // HPが50%以下の味方がいるとき
    const livingAllies = this.battle.getLivingAllies(this.unit.id); // まず生きている味方を取得

    for (let i = 0; i < livingAllies.length; i++) {
      if (livingAllies[i].hp / livingAllies[i].mhp <= 0.5) {
```

```
        return true; // HP50%以下の味方がいるなら実行可能
      }
    }

    return false; // いなければ実行不可
  }
}

/*-----------------------------------------------
  スキル
-----------------------------------------------*/

class Skill {
  constructor() {
    this.name = ''; // スキル名
    this.skillEffectChain = null; // 実行されるスキルエフェクトチェーン
  }

  init(unit, battle) { // 初期化処理 スキル使用者とバトル環境を受け取る
    this.unit   = unit;
    this.battle = battle;

    for (let i = 0; i < this.skillEffectChain.length; i++) {
      this.skillEffectChain[i].init(unit, battle); // スキルエフェクトをそれぞれ初期化
    }
  }

  execute() { // スキル実行
    for (let i = 0; i < this.skillEffectChain.length; i++) {
      this.skillEffectChain[i].resolve();
    }
  }

  getDescription() {
    return {
      type: '',
      name: this.name,
      cond: '',
      desc: ''
    }
  }

  static isAvailable(status) {
    return false;
  }
}

class ActiveSkill extends Skill {
  constructor() {
    super();

    this.name = ''; // スキル名
    this.cost = 0;  // 実行時に消費するSP
    this.condition = null; // スキル発動条件を初期化
    this.skillEffectChain = null; // 実行されるスキルエフェクトチェーン
  }
```

```
  init(unit, battle) { // 初期化処理 スキル使用者とバトル環境を受け取る
    super.init(unit, battle); // 親クラスのinitを呼び出してから
    this.condition.init(unit, battle); // スキル発動条件を初期化
  }

  isCostOk() { // SPが足りているかどうか
    return this.cost <= this.unit.sp;
  }

  isExecutable() { // 実行可能かどうか
    return this.condition.resolve();
  }

  afterExecute() { // 実行後処理、SPを消費する
    this.unit.consumeSp(this.cost);
  }

  getDescription() {
    return {
      type: 'Active',
      name: this.name,
      cond: `${this.cost}SP / ${this.condition.name}`,
      desc: this.skillEffectChain.map(se => se.getDescription()).join('+')
    }
  }
}

class PassiveSkill extends Skill {
  constructor() {
    super();

    this.name = '';      // スキル名
    this.rate = 0;       // スキル発動率
    this.trigger = null; // 発動トリガー
    this.skillEffectChain = null; // 実行されるスキルエフェクトチェーン
  }

  isRateCheckOk() { // 発動率チェック
    return Math.random() * 100 < this.rate;
  }

  getTriggerId() {
    return this.trigger.id;
  }

  getDescription() {
    return {
      type: 'Passive',
      name: this.name,
      cond: this.trigger.name,
      desc: this.skillEffectChain.map(se => se.getDescription()).join('+')
    }
  }
}

class NormalAttack extends ActiveSkill {
  constructor() {
```

```
    super();

    this.name = '通常攻撃';
    this.cost = 0;
    this.condition = new Always();
    this.skillEffectChain = [
      new SkillEffect(new SomeEnemyTarget(1), true, [new AttackElement(50)])
      // 【SP0：通常時】敵：攻撃（回避可能、攻撃の威力50）
    ];
  }

  isCostOk() { // SPが足りているかどうか、通常攻撃は必ず実行可能とする
    return true;
  }

  isExecutable() { // 実行条件、通常攻撃は必ず実行可能とする
    return true;
  }

  static isAvailable(status) { // デフォルトで利用可能なだけで習得するものではないのでfalse
    return false;
  }
}

class Attack extends ActiveSkill {
  constructor() {
    super();

    this.name = 'アタック';
    this.cost = 30;
    this.condition = new Always();
    this.skillEffectChain = [
      new SkillEffect(new SomeEnemyTarget(1), true, [new AttackElement(160)])
    ];
  }

  static isAvailable(status) { // 初期習得
    return true;
  }
}

class Flare extends ActiveSkill {
  constructor() {
    super();

    this.name = 'フレア';
    this.cost = 50;
    this.condition = new PerActions(9);
    this.skillEffectChain = [
      new SkillEffect(new SomeEnemyTarget(1), true, [new AttackElement(200)]),
      new SkillEffect(new AllEnemyTarget(), true, [new AttackElement(200)], new SpRemain(30))
    ];
  }

  static isAvailable(status) {
    return 20 <= status.atk;
  }
```

```
}

class TwinAttack extends ActiveSkill {
  constructor() {
    super();

    this.name = '双撃';
    this.cost = 40;
    this.condition = new PerActions(3);
    this.skillEffectChain = [
      new SkillEffect(new SomeEnemyTarget(2), true, [new AttackElement(120)])
    ];
  }

  static isAvailable(status) { // 初期習得
    return true;
  }
}

class PoisonAttack extends ActiveSkill {
  constructor() {
    super();

    this.name = 'ポイズン';
    this.cost = 30;
    this.condition = new PerActions(4);
    this.skillEffectChain = [
      new SkillEffect(new SomeEnemyTarget(1), true, [new AttackElement(30), new PoisonElement(3)])
    ];
  }

  static isAvailable(status) {
    return 20 <= status.dex;
  }
}

class Heal extends ActiveSkill {
  constructor() {
    super();

    this.name = 'ヒール';
    this.cost = 20;
    this.condition = new WeakenedAllyExists();
    this.skillEffectChain = [
      new SkillEffect(new SomeWeakenedAllyTarget(1), false, [new HealElement(40)])
    ];
  }

  static isAvailable(status) { // 初期習得
    return true;
  }
}

class AreaHeal extends ActiveSkill {
  constructor() {
    super();
```

```
    this.name = 'エリアヒール';
    this.cost = 50;
    this.condition = new WeakenedAllyExists();
    this.skillEffectChain = [
      new SkillEffect(new AllAllyTarget(), false, [new HealElement(40)])
    ];
  }

  static isAvailable(status) {
    return 20 <= status.mnd;
  }
}

class ShiftUp extends PassiveSkill {
  constructor() {
    super();

    this.name = 'シフトアップ';
    this.rate = 100;
    this.trigger = new BattleStart();
    this.skillEffectChain = [
      new SkillEffect(new OwnTarget(), false, [new GainAgiElement(5)])
    ];
  }

  static isAvailable(status) { // 初期習得
    return true;
  }
}

class CounterAttack extends PassiveSkill {
  constructor() {
    super();

    this.name = '反撃';
    this.rate = 30;
    this.trigger = new Dodge();
    this.skillEffectChain = [
      new SkillEffect(new SomeEnemyTarget(1), true, [new AttackElement(50)])
    ]
  }

  static isAvailable(status) {
    return 20 <= status.agi;
  }
}

class Ultimatum extends PassiveSkill {
  constructor() {
    super();

    this.name = 'アルティメイタム';
    this.rate = 100;
    this.trigger = new BattleStart();
    this.skillEffectChain = [
      new SkillEffect(new AllEnemyTarget(), false, [new HateElement(200)])
    ];
```

```
  }

  static isAvailable(status) { // 初期習得
    return true;
  }
}

class SelfHealing extends PassiveSkill {
  constructor() {
    super();

    this.name = '自己修復';
    this.rate = 20;
    this.trigger = new Attacked();
    this.skillEffectChain = [
      new SkillEffect(new OwnTarget(), false, [new HealElement(5)])
      // 【被攻撃時】自：回復
    ]
  }

  static isAvailable(status) {
    return 20 <= status.def;
  }
}

/*-------------------------------------------------
  読み込み
-------------------------------------------------*/

const skillArray = [
  /*  0 */ NormalAttack,
  /*  1 */ Attack,
  /*  2 */ Flare,
  /*  3 */ TwinAttack,
  /*  4 */ PoisonAttack,
  /*  5 */ Heal,
  /*  6 */ AreaHeal,
  /*  7 */ ShiftUp,
  /*  8 */ CounterAttack,
  /*  9 */ Ultimatum,
  /* 10 */ SelfHealing
]

const skillResolver = (skillIds) => {
  // スキル読み込み用の関数
  // スキルIDの配列を受けとってスキルのインスタンス配列を返す
  // 外部から呼び出すときnullをスキップできたほうが便利なのでスキップさせる
  const skills = [];

  for (let i = 0; i < skillIds.length; i++) {
    if (skillIds[i] != null) {
      const SkillClass = skillArray[skillIds[i]];
      skills.push(new SkillClass());
    }
  }

  return skills;
```

```
};

const isActiveSkill = (skillInstance) => {
  // アクティブスキルかどうかを判定する関数
  return skillInstance instanceof ActiveSkill;
};

const isPassiveSkill = (skillInstance) => {
  // パッシブスキルかどうかを判定する関数
  return skillInstance instanceof PassiveSkill;
};

const getDescriptions = (skillIds) => {
  // スキル説明文読み込み用の関数
  // スキルID配列を受け取ってスキル説明オブジェクトを返す
  const skillDescriptions = [];

  for (let i = 0; i < skillIds.length; i++) {
    if (skillIds[i] == null) {
      // スキルIDがnullであれば該当の説明オブジェクトはnullにする
      // （外部から参照するときに扱いやすくするため）
      skillDescriptions.push(null);
    } else {
      const SkillClass  = skillArray[skillIds[i]];
      const instance    = new SkillClass();
      const description = instance.getDescription();
      description.id    = skillIds[i]; // ID情報も付加しておく
      skillDescriptions.push(description);
    }
  }

  return skillDescriptions;
};

const getAvailableSkillIds = (status) => {
  // 習得可能なスキルを取得する関数
  // ステータスを受け取って習得可能なスキルのIDの配列を返す
  const skillIds = [];

  for (let i = 0; i < skillArray.length; i++) {
    const SkillClass = skillArray[i];
    if (SkillClass.isAvailable(status)) {
      skillIds.push(i);
    }
  }

  return skillIds;
};

module.exports = {
  skillResolver:        skillResolver,
  isActiveSkill:        isActiveSkill,
  isPassiveSkill:       isPassiveSkill,
  getDescriptions:      getDescriptions,
  getAvailableSkillIds: getAvailableSkillIds
};
```

4.13 戦闘設定

4.13.1 戦闘設定APIの作成

戦闘システムが組み上がったので戦闘設定を作っていきましょう。まずはそれに関連する設定を「backend/config/default.json」に追記します。以下の設定を追加してください。ここではセット可能なスキル数を設定しています。

backend/config/default.json

```
"settableSkillMaxLength": 5
```

次に「backend/server/api.js」で「skill.js」を読み込むようにしましょう。「backend/server/api.js」を開き冒頭部分を以下のように書き換えてください。

backend/server/api.js

```
const express   = require('express');
const config    = require('config');
const passport  = require('passport');
const styling   = require('./styling.js');
const skill     = require('./skill.js');
const Character = require('./model.js').Character;
const Room      = require('./model.js').Room;
const Message   = require('./model.js').Message;
const router    = express.Router();

  （省略）
```

次は戦闘設定APIを作ります。戦闘設定APIではNPの割り振りとスキル設定を行います。まずは「/characters/main/skill」にGETでアクセスすることで戦闘設定ページに必要な情報が受け取れるようにしましょう。「backend/server/api.js」に以下のAPIを追記してください。

backend/server/api.js

```
router.get('/characters/main/skill', checkAuthentication, async (req, res) => {
  try {
    const character = await Character.findById(req.user._id, {
      np:     1,
      status: 1,
      skill:  1
    });

    const availableSkills = skill.getDescriptions(skill.getAvailableSkillIds(character.status));

    return res.status(200).send({
      np:              character.np,
      status:          character.status,
      skillIds:        character.skill,
      availableSkills: availableSkills
```

```
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

現在のNP、ステータス、習得しているスキルIDと習得可能なスキルを返すようにしています。次は更新側の処理です。「/characters/main/skill」にPUTでアクセスすることで戦闘設定を反映できるようにします。「backend/server/api.js」に以下のAPIを追加して保存してください。

backend/server/api.js

```
router.put('/characters/main/skill', checkAuthentication, checkCsrf, async (req, res) => {
  try {
    if (
      !req.body.addStatus ||
      !Array.isArray(req.body.skillIds) ||
      typeof req.body.addStatus.atk != 'number' || !/^[0-9]*$/.test('' + req.body.addStatus.atk) ||
      typeof req.body.addStatus.dex != 'number' || !/^[0-9]*$/.test('' + req.body.addStatus.dex) ||
      typeof req.body.addStatus.mnd != 'number' || !/^[0-9]*$/.test('' + req.body.addStatus.mnd) ||
      typeof req.body.addStatus.agi != 'number' || !/^[0-9]*$/.test('' + req.body.addStatus.agi) ||
      typeof req.body.addStatus.def != 'number' || !/^[0-9]*$/.test('' + req.body.addStatus.def)
    ) {
      // 必要なフィールドがなかったり
      // 加算ステータスの入力内容が数値ではなかったり0以上の整数ではなかったとき400を返して中断
      return res.status(400).send();
    };

    if (config.settableSkillMaxLength < req.body.skillIds.length) {
      // 設定可能なスキル数を超えた数のスキルを設定しようとしていた場合400を返して中断
      return res.status(400).send();
    }

    const skillIdsChecker = {};
    for (let i = 0; i < req.body.skillIds.length; i++) {
      if (req.body.skillIds[i] && !skillIdsChecker[req.body.skillIds[i]]) {
        skillIdsChecker[req.body.skillIds[i]] = true;
      } else if (req.body.skillIds[i] && skillIdsChecker[req.body.skillIds[i]]) {
        // スキルが重複していたとき400を返して中断
        return res.status(400).send();
      }
    }

    // キャラクター情報を取得
    const character = await Character.findById(req.user._id, {
      np:     1,
      status: 1,
      skill:  1
    });

    // 加算するステータスの合計を計算
    const addStatusSum = (
      req.body.addStatus.atk +
      req.body.addStatus.dex +
```

```
      req.body.addStatus.mnd +
      req.body.addStatus.agi +
      req.body.addStatus.def
    );

    if (character.np < addStatusSum) {
      // 割り振れるNPに対して加算するステータス合計が多すぎる場合400を返して中断
      return res.status(400).send();
    }

    const availableSkillIds = skill.getAvailableSkillIds(character.status);
    for (let i = 0; i < req.body.skillIds.length; i++) {
      if (req.body.skillIds[i] && !availableSkillIds.includes(req.body.skillIds[i])) {
        // 習得可能ではないスキルを習得しようとしていた場合400を返して中断
        return res.status(400).send();
      }
    }

    // 問題がなければ更新へ
    await character.update({
      $inc: {
        np: addStatusSum * -1, // 加算ステータス合計の分NPを減算

        "status.atk": req.body.addStatus.atk,
        "status.dex": req.body.addStatus.dex,
        "status.mnd": req.body.addStatus.mnd,
        "status.agi": req.body.addStatus.agi,
        "status.def": req.body.addStatus.def
      },
      skill: req.body.skillIds
    });

    return res.status(200).send();
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

addStatusから加算するステータス、skillIdsからセットするスキルのIDを受け取り更新しています。設定可能な値の範囲が複雑なので検証処理が長いですが、やっていること自体は指定の分ステータスを加算してNPを減らしスキルをセットするというだけです。なお、スキルIDにnullが設定されているとスキル未設定という扱いにしています。

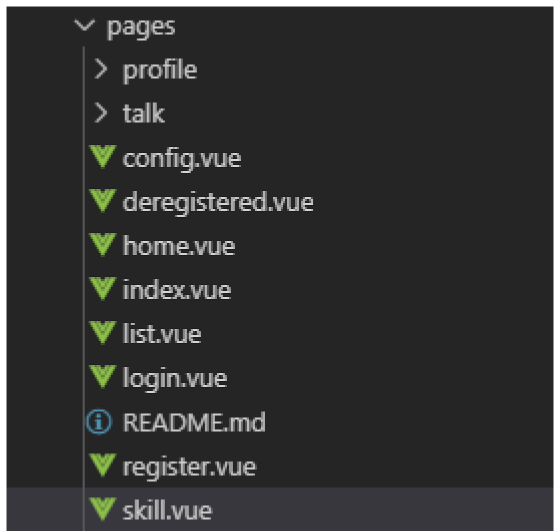
4.13.2 戦闘設定ページの作成

APIができたら戦闘設定ページを作りましょう。まずは設定できる最大スキル数の設定を作ります。「frontend/nuxt.config.js」の「env」内に以下の設定を追記してください。

frontend/nuxt.config.js

```
  settableSkillMaxLength: 5
```

「http://dev.siroisakana.com/ta/skill」というURLからスキル設定を行うことにします。「frontend/pages」内に「skill.vue」を追加します。ファイル構成は以下のようになります。



作成したら以下の内容を記述して保存してください。

frontend/pages/skill.vue

```
<template>
  <section>
    <sub-heading>能力値割り振り</sub-heading>
    <section class="form">
      <div class="form-description">
        所持しているNPを能力値に割り振ることができます。<br>
        入力欄に各能力値に割り振りたい値を入力してください。
      </div>
      <div class="remaining-np">
        残りNP:
        <span class="remaining-np-error" v-if="remainingNp == 'ERR'">エラー</span>
        <span class="remaining-np-error" v-else-if="remainingNp < 0">{{ remainingNp * -1 }}超過</span>
        <span v-else>{{ remainingNp }}</span>
      </div>
      <table class="statuses">
        <tbody>
          <tr class="statuses-key">
            <th>ATK</th>
            <th>DEX</th>
            <th>MND</th>
            <th>AGI</th>
            <th>DEF</th>
          </tr>
          <tr class="statuses-value">
            <td>{{ status.atk }}</td>
            <td>{{ status.dex }}</td>
            <td>{{ status.mnd }}</td>
            <td>{{ status.agi }}</td>
```

```
            <td>{{ status.def }}</td>
          </tr>
          <tr class="statuses-input">
            <td><input type="number" v-model="addStatus.atk" min=0></td>
            <td><input type="number" v-model="addStatus.dex" min=0></td>
            <td><input type="number" v-model="addStatus.mnd" min=0></td>
            <td><input type="number" v-model="addStatus.agi" min=0></td>
            <td><input type="number" v-model="addStatus.def" min=0></td>
          </tr>
          <tr class="statuses-sum">
            <td>{{ sum('atk') }}</td>
            <td>{{ sum('dex') }}</td>
            <td>{{ sum('mnd') }}</td>
            <td>{{ sum('agi') }}</td>
            <td>{{ sum('def') }}</td>
          </tr>
        </tbody>
      </table>
    </section>
    <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
    <div class="button-wrapper">
      <button class="button" @click="update">更新</button>
    </div>
    <sub-heading>スキル設定</sub-heading>
    <section class="form">
      <div class="form-description">
        スキルを設定することができます。アクティブスキルはリストの上から順番に実行判断されます。<br>
        スキルを重複して選択することはできません。
      </div>
      <table class="skills">
        <tbody>
          <tr v-for="(skillId, index) in skillIds" :key="index">
            <th class="skill-number">{{ index + 1 }}.</th>
            <td class="skill-select-wrapper">
              <select class="skill-select" v-model="skillIds[index]">
                <option :value="null">-- スキルを選択 --</option>
                <option v-for="availableSkill in availableSkills" :key="availableSkill.id" :value=
"availableSkill.id">{{ availableSkill.id }}. {{ availableSkill.name }}</option>
              </select>
            </td>
            <td class="skill-description" v-if="skillId">【{{ skillCond(skillId) }}】{{ skillDesc
(skillId) }}</td>
            <td class="skill-description" v-else>----</td>
          </tr>
        </tbody>
      </table>
    </section>
    <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
    <div class="button-wrapper">
      <button class="button" @click="update">更新</button>
    </div>
  </section>
</template>

<script>
import SubHeading    from '~/components/SubHeading.vue'
import MessageBanner from '~/components/MessageBanner.vue'
```

```
export default {
  components: {
    SubHeading,
    MessageBanner
  },
  middleware: 'authenticated',
  head() {
    return {
      title: '戦闘設定'
    };
  },
  data() {
    return {
      settableSkillMaxLength: process.env.settableSkillMaxLength,

      // inputにv-modelで紐付けるので値の型はStringなことに注意
      addStatus: {
        atk: '0',
        dex: '0',
        mnd: '0',
        agi: '0',
        def: '0'
      },
      errorMessage:    '',
      waitingResponse: false
    }
  },
  asyncData: async function(context) {
    const response = await context.$axios.get('/api/characters/main/skill');

    // アイコン配列が最大長に満たない場合nullで埋めて延長
    // nullは設定スキルなしの扱い
    const skillIds = [];
    for (let i = 0; i < process.env.settableSkillMaxLength; i++) {
      skillIds.push(response.data.skillIds[i] || null);
    }

    return {
      np:              response.data.np,
      status:          response.data.status,
      skillIds:        skillIds,
      availableSkills: response.data.availableSkills
    };
  },
  computed: {
    remainingNp: {
      get() {
        if (
          !/^[0-9]*$/.test(this.addStatus.atk) ||
          !/^[0-9]*$/.test(this.addStatus.dex) ||
          !/^[0-9]*$/.test(this.addStatus.mnd) ||
          !/^[0-9]*$/.test(this.addStatus.agi) ||
          !/^[0-9]*$/.test(this.addStatus.def)
        ) {
          // どれかの入力内容が空文字列や0以上の整数ではなかったときはERRを返す
          return 'ERR';
```

```
      }

      return (
        this.np
        - Number(this.addStatus.atk)
        - Number(this.addStatus.dex)
        - Number(this.addStatus.mnd)
        - Number(this.addStatus.agi)
        - Number(this.addStatus.def)
      );
    }
  }
},
methods: {
  sum: function(key) {
    if (!/^[0-9]*$/.test(this.addStatus[key])) {
      // 入力内容が空文字列や0以上の整数ではなかったときはERRを返す
      return 'ERR';
    }

    return '' + (this.status[key] + Number(this.addStatus[key]));
  },
  skillCond: function(id) {
    for (let i = 0; i < this.availableSkills.length; i++) {
      if (this.availableSkills[i].id == id) {
        return this.availableSkills[i].cond;
      }
    }
  },
  skillDesc: function(id) {
    for (let i = 0; i < this.availableSkills.length; i++) {
      if (this.availableSkills[i].id == id) {
        return this.availableSkills[i].desc;
      }
    }
  },
  update: async function() {
    if (this.waitingResponse) {
      // 接続待ち状態であればアラートを表示しreturn、以降の処理を行わない
      return alert('しばらくお待ち下さい');
    }

    // 入力内容の検証を行う、問題があればエラーメッセージを表示、returnして処理を中断
    if (this.remainingNp == 'ERR') {
      return this.errorMessage = '能力値割り振りの入力内容に誤りがあります';
    }
    if (this.remainingNp < 0) {
      return this.errorMessage = 'NPが足りません'
    }

    // スキルの重複チェック
    const skillIdsChecker = {};
    for (let i = 0; i < this.skillIds.length; i++) {
      if (this.skillIds[i] == null) {
        // nullであればチェックをスキップ
        continue;
      }
```

```
        if (!skillIdsChecker[this.skillIds[i]]) {
          // skillIdsCheckerがまだtrueでないならtrueに
          skillIdsChecker[this.skillIds[i]] = true;
        } else if (tskillIdsChecker[this.skillIds[i]]) {
          // すでにskillIdsCheckerがtrueなら重複
          return this.errorMessage = 'スキルが重複しています';
        }
      }

      // 問題がなければ接続に入る、接続待ち状態をON(true)に
      this.waitingResponse = true;

      try {
        // 更新を行う
        await this.$axios.put('/api/characters/main/skill', {
          csrf:     this.$store.getters['auth/loginCharacter'].csrf,
          skillIds: this.skillIds,
          addStatus: { // Number型に変換しておく
            atk: Number(this.addStatus.atk),
            dex: Number(this.addStatus.dex),
            mnd: Number(this.addStatus.mnd),
            agi: Number(this.addStatus.agi),
            def: Number(this.addStatus.def)
          }
        });

        // 更新結果を受け取りデータ更新
        const response = await this.$axios.get('/api/characters/main/skill');
        this.addStatus = {
          atk: '0',
          dex: '0',
          mnd: '0',
          agi: '0',
          def: '0'
        };
        this.np              = response.data.np;
        this.status          = response.data.status;
        this.skillIds        = response.data.skillIds;
        this.availableSkills = response.data.availableSkills;

        // エラーメッセージを解除し接続待ち状態をOFF(false)に
        this.errorMessage = '';
        this.waitingResponse = false;
      } catch (e) {
        // エラーが発生した場合エラーメッセージを表示して接続待ち状態をOFF(false)に
        this.errorMessage = '更新中にエラーが発生しました';
        this.waitingResponse = false;
      }
    }
  }
}
</script>

<style lang="scss" scoped>
$statusPaddingLeft: 11px;
$statusInputWidth: 100px;
```

```
$font: 'BIZ UDPGothic', 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ', Meiryo, 'ＭＳ Ｐゴシック', sans-serif;

.remaining-np {
  padding: 20px;
  font-weight: bold;
  font-family: $font;
  font-size: 24px;
  color: #666;

  .remaining-np-error {
    color: #c53f4d;
  }
}

.statuses {
  padding-left: 20px;
  font-family: $font;

  .statuses-key {
    th {
      padding-left: $statusPaddingLeft;
    }
  }

  .statuses-value {
    td {
      padding-left: $statusPaddingLeft;
    }
  }

  .statuses-input {
    td {
      padding-left: 0;

      input {
        margin: 0;
        width: $statusInputWidth;
      }
    }
  }

  .statuses-sum {
    td {
      padding-left: $statusPaddingLeft;
    }
  }
}

.skills {
  padding-top: 20px;

  tr:first-child th,
  tr:first-child td {
    border-top: 1px solid #E1E1E1;
  }
```

```
  .skill-number {
    padding-left: 10px;
  }

  .skill-select {
    margin: 0;
  }

  .skill-description {
    min-width: 400px;
  }
}
</style>
```

ページを表示すると以下のようになります(長いので一部のみ)。いろいろとスキルや能力値を設定してみながらテストしてみましょう。

NPとステータス、現在スキルと習得可能なスキルの説明リストを取得して編集を行うページになります。取得時にスキル配列の長さが最大長ではないときはnull(スキル未設定)で埋めてデータを扱いやすくしています。

能力値入力ではremainingNpという算出プロパティを用意しており、これで残りのNPが取得できる他、超過していたらマイナスになったり入力内容がおかしければ"ERR"が返るようになっていたりしてエラーチェックにも使えるようになっています。他にもsumというメソッドを用意して加算後のステータスも取得しやすくしています。

スキル入力ではselectにv-modelを使ってスキルを選択できるようにしています。このとき、スキルを未設定にすると値がnullになるようになっています。また、選択されたスキルを表示するためにskillCond、skillDescというメソ

ッドを作っています。これはそれぞれスキルIDを受け取りavailableSkills配列からスキルIDが合致するものを取得して指定の情報を返すためのメソッドになります。

4.13.3 プロフィールの改変

スキルが習得できるようになったので、それぞれのキャラクターページやホーム、キャラクター設定ページなどにスキル情報を表示するようにしましょう。まずはAPIからスキル情報が返るようにします。「backend/server/api.js」を開き、「router.get('/characters/main/home', … 」「router.get('/characters/:key(\\d+|main\)', …」「router.get('/characters/main/profile', …」の3つのAPIをそれぞれ以下のように書き換えてください。

backend/server/api.js

```
router.get('/characters/main/home', checkAuthentication, async (req, res) => {
  try {
    const character = await Character.findById(req.user._id, {
      eno:           1,
      name:          1,
      tags:          1,
      profile:       1,
      icons:         1,
      ap:            1,
      np:            1,
      status:        1,
      skill:         1,
      declare:       1,
      profileImages: 1
    });

    return res.status(200).send({
      eno:     character.eno,
      name:    character.name,
      tags:    character.tags,
      profile: character.profile,
      icons:   character.icons,
      ap:      character.ap,
      np:      character.np,
      status:  character.status,
      skill:   skill.getDescriptions(character.skill),
      isDeclared: { // 宣言をしているかどうか
        diary: character.declare.diary != null,
        story: character.declare.storyId != null,
        party: character.declare.party.length != 0
      },
      profileImages: character.profileImages.length ? character.profileImages[Math.floor(Math.rand
om() * character.profileImages.length)] : null
      // プロフィール画像配列が存在するときランダムに1つを返す、なければnull
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

```
backend/server/api.js
router.get('/characters/:key(\\d+|main\)', async (req, res) => {
  if (/main/.test(req.params.key) && !req.user) {
    return res.status(401).send(); // 自キャラ表示なのにログインしていなければ401
  }

  const query = /main/.test(req.params.key) ? {_id: req.user._id} : {eno: Number(req.params.key)};
  // 自キャラ表示であればセッションユーザーのキャラ、そうでなければ指定のENoで検索

  query.deleted = false;
  // 削除フラグが立っていないことを検索条件に追加

  try {
    const character = await Character.findOne(query, {
      eno:           1,
      name:          1,
      tags:          1,
      profile:       1,
      icons:         1,
      status:        1,
      skill:         1,
      profileImages: 1
    });

    if (!character) {
      return res.status(404).send(); // キャラクターの検索結果が存在しなければ404
    } else if (req.user && !(/main/.test(req.params.key))) {
      // キャラクターが見つかり、かつログインしていて自キャラ表示ではない場合
      // 接続者のお気に入り、ブロック、ミュート情報を取得
      const user = await Character.findById(req.user._id, {
        _id:   0,
        fav:   1,
        block: 1,
        mute:  1
      });

      // キャラクター情報とお気に入り、ブロック、ミュートしているかという情報を返す
      return res.status(200).send({
        eno:           character.eno,
        name:          character.name,
        tags:          character.tags,
        profile:       character.profile,
        icons:         character.icons,
        status:        character.status,
        skill:         skill.getDescriptions(character.skill),
        profileImage:  character.profileImages.length ? character.profileImages[Math.floor(Math.ra
ndom() * character.profileImages.length)] : null,
        isFaved:       user.fav.includes(character._id),
        isBlocked:     user.block.includes(character._id),
        isMuted:       user.mute.includes(character._id)
      });
    } else {
      // キャラクターは見つかったがログインはしていないとき
      // そのまま情報を返す
      return res.status(200).send({
        eno:           character.eno,
        name:          character.name,
```

```
        tags:          character.tags,
        profile:       character.profile,
        icons:         character.icons,
        status:        character.status,
        skill:         skill.getDescriptions(character.skill),
        profileImage:  character.profileImages.length ? character.profileImages[Math.floor(Math.ra
ndom() * character.profileImages.length)] : null
      });
    }
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

```
backend/server/api.js
router.get('/characters/main/profile', checkAuthentication, async (req, res) => {
  try {
    const character = await Character.findById(req.user._id, {
      eno:           1,
      name:          1,
      nickname:      1,
      tags:          1,
      summary:       1,
      mainicon:      1,
      profileImages: 1,
      profile:       1,
      icons:         1,
      status:        1,
      skill:         1
    });

    return res.status(200).send({
      eno:           character.eno,
      name:          character.name,
      nickname:      character.nickname,
      tags:          character.tags,
      summary:       character.summary,
      mainicon:      character.mainicon,
      profileImages: character.profileImages,
      profile:       character.profile,
      icons:         character.icons,
      status:        character.status,
      skill:         skill.getDescriptions(character.skill)
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

APIを修正したら次はフロントエンド側を変更します。コンポーネントから修正していきましょう。「frontend/components/CharacterProfile.vue」を開いてください。<template>内の<div class="skills">となっている所と<sc

ript>内のpropsを以下のように書き換えて保存します。skillを受け取ってv-forでスキルごとの説明を表示するようにしています。スキル未設定だと説明オブジェクトがnullになるので、nullの場合は「----」が表示されるようになっています。

frontend/components/CharacterProfile.vue

```
（省略）

      <div class="skills">
        <div class="skill" v-for="(skillDescription, index) in skill" :key="index">
          <div class="skill-prop">
            <div class="skill-name">{{ skillDescription ? skillDescription.name : '----'}}</div>
            <div class="skill-cond">{{ skillDescription ? skillDescription.cond : '----'}}</div>
          </div>
          <div class="skill-effect">{{ skillDescription ? skillDescription.desc : '----'}}</div>
        </div>
      </div>

（省略）

props: {
  mode:         String, // 表示モード home = ホーム、profile = キャラクターページ、preview = プレビュー
  eno:          Number,
  name:         String,
  tags:         Array,
  profile:      String,
  profileImage: String,
  icons:        Array,
  ap:           Number,
  status:       Object,
  skill:        Array,
  isFaved:      Boolean,
  isBlocked:    Boolean,
  isMuted:      Boolean
},

（省略）
```

　あとはそれぞれのページでskillをAPIから受け取って渡すようにするだけです。「frontend/pages/home.vue」「frontend/pages/profile/_key.vue」「frontend/pages/profile/main/edit.vue」のそれぞれのページの<template>内の<character-profile>と、<script>のasyncData内のreturnをそれぞれ以下のように書き換えて保存してください。

frontend/pages/home.vue

```
（省略）

  <character-profile
    mode="home"
    :eno="eno"
    :name="name"
    :tags="tags"
    :profile="profile"
```

```
    :profileImage="profileImage"
    :icons="icons"
    :ap="ap"
    :status="status"
    :skill="skill" />

(省略)

  return {
    eno:          response.data.eno,
    name:         response.data.name,
    tags:         response.data.tags,
    profile:      response.data.profile,
    profileImage: response.data.profileImage,
    icons:        response.data.icons,
    ap:           response.data.ap,
    np:           response.data.np,
    isDeclared:   response.data.isDeclared,
    status:       response.data.status,
    skill:        response.data.skill
  };

(省略)
```

frontend/pages/profile/_key.vue

```
(省略)

  <character-profile
    mode="profile"
    :eno="eno"
    :name="name"
    :tags="tags"
    :profile="profile"
    :profileImage="profileImage"
    :icons="icons"
    :status="status"
    :skill="skill"
    :isFaved="isFaved"
    :isBlocked="isBlocked"
    :isMuted="isMuted"
    @relation="relation"/>

(省略)

  return {
    eno:          response.data.eno,
    name:         response.data.name,
    tags:         response.data.tags,
    profile:      response.data.profile,
    profileImage: response.data.profileImage,
    icons:        response.data.icons,
    status:       response.data.status,
    skill:        response.data.skill,
    isFaved:      response.data.isFaved,
    isBlocked:    response.data.isBlocked,
    isMuted:      response.data.isMuted
```

```
  };

（省略）
```

frontend/pages/profile/main/edit.vue

```
（省略）

  <character-profile
    mode="preview"
    :eno="eno"
    :name="name"
    :tags="joinedTags ? joinedTags.split(/\s+/) : []"
    :profile="profile | styling"
    :profileImage="joinedProfileImages ? joinedProfileImages.split(/\n/)[Math.floor(Math.random(
) * joinedProfileImages.split(/\n/).length)] : null"
    :icons="icons"
    :status="status"
    :skill="skill" />

（省略）

  return {
    eno:        response.data.eno,
    name:       response.data.name,
    nickname:   response.data.nickname,
    summary:    response.data.summary,
    profile:    profile,
    mainicon:   response.data.mainicon,
    icons:      icons,
    status:     response.data.status,
    skill:      response.data.skill,
    joinedTags: response.data.tags.join(' '),
    joinedProfileImages: response.data.profileImages.join('\n')
  };

（省略）
```

4.14 探索戦(AP)

4.14.1 EJS

ここからは実際に戦闘を行って戦闘結果ページが生成されるようにしていきます。まずは戦闘ログページをどうやって生成するかについて学びます。単純に文字列を連結してHTMLを生成してもいいのですが、いちいち文字列を+で連結していくのは非常に面倒です。

そこでテンプレートエンジンというものを用います。テンプレートエンジンとは予めどういう感じの出力にするかをテンプレートに記述しておき、そこにデータを入れて結果を生成するための仕組みのことです。

本書ではテンプレートエンジンはEJSを用います。EJSはHTMLにJavaScriptを突っ込んだような感じでテンプレートを記述することができる非常に習得が容易なライブラリです。EJSは以下のような感じで呼び出します。第1引数がテンプレート、第2引数がテンプレートに渡すデータです。

```javascript
const rendered = ejs.render(template, {
  hoge: 'hogehoge',
  fuga: 'foobar'
});
```

この例においてtemplateの内容が以下のような感じだったとしましょう。

```ejs
<div>
  hogeの内容は<%- hoge %>！
  fugaの内容は<%= fuga %>！
</div>
```

そうすると結果、つまり変数renderedの内容は以下のようになります。

```html
<div>
  hogeの内容はhogehoge！
  fugaの内容はfoobar！
</div>
```

EJSを使うことこのような感じで非常に簡単にHTML内に表示したい内容を組み込むことができます。なお、値を表示する構文には<%=　　%>と<%-　　%>の2種類があるのですが前者はサニタイズをしてから出力、後者は特にサニタイズをせずそのまま出力するといった違いがあります。基本は<%=　　%>の方を使いつつ、必要があれば<%-　　%>の方を使うといいでしょう。(後者を使う場合ユーザーが入力しうる箇所のサニタイズは忘れずに行いましょう。)

他にもEJSを使うとJavaScriptを使って柔軟なレンダリングを行うことができたりします。例えばifで出力を出し

分けたり、forで繰り返し出力を行ったりすることができます。<%　　%>で囲まれた部分がJavaScriptで処理を行える部分になります。例としては以下のような感じでしょうか。

EJS
```
<div>
  <% if (error) { %>
    <% const errorMessageMax = 10 < errorMessage.length ? 10 : errorMessage.length; %>
    エラー中！<br>
    エラーログ ↓<br>
    <% for (let i = 0; i < errorMessageMax; i++) { %>
      <%= errorMessage[i] %><br>
    <% } %>
  <% } %>
</div>
```

余談はこれぐらいにして、実際に探索戦で使うテンプレートを作っていきましょう。テンプレートについては別のディレクトリで管理することにします。「backend」内に「template」ディレクトリを作りその中に「explore.ejs」を作成してください。ファイル構成は以下のようになります。

```
⌄ backend
  › config
  › node_modules
  › server
  ⌄ template
    <> explore.ejs
```

作成したら以下の内容を入力して保存してください。

backend/template/explore.ejs
```
<!DOCTYPE html>
<html lang="ja">
  <head>
    <meta charset="UTF-8">
    <title><%- title %></title>
    <style>
      html {
        overflow-y: scroll;
      }

      body {
        background: #F8F8F8;
        color: #333;
        font-size: 16px;
        font-family: 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ', Meiryo, 'ＭＳ Ｐゴシック', sans-serif;
      }

      .stage {
```

```
        margin: 0 auto;
        width: 1000px;
      }

      .title {
        box-sizing: border-box;
        padding: 20px 20px;
        margin-bottom: 20px;
        width: 100%;
        font-size: 36px;
        letter-spacing: 2px;
        border-bottom: 1px solid lightgray;
        color: #444;
        font-family: 'BIZ UDPGothic', 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ
', Meiryo, 'ＭＳ Ｐゴシック', sans-serif;
      }

      .description {
        padding: 20px;
      }

      .battle {
        margin: 0 auto;
        width: 1000px;
      }

      .battle-start {
        width: 100%;
      }

      .battle-start-call {
        box-sizing: border-box;
        padding: 20px 20px;
        margin-bottom: 20px;
        width: 100%;
        font-size: 36px;
        letter-spacing: 2px;
        border-bottom: 1px solid lightgray;
        color: #444;
        font-family: 'BIZ UDPGothic', 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ
', Meiryo, 'ＭＳ Ｐゴシック', sans-serif;
      }

      .battle-start-call::first-letter {
        font-size: 44px;
      }

      .round {
        padding: 10px;
      }

      .round-start {
        box-sizing: border-box;
        padding: 20px 20px;
        width: 100%;
        border-bottom: 1px solid lightgray;
        font-size: 24px;
```

```
        color: #666;
        font-family: 'BIZ UDPGothic', 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ
', Meiryo, 'ＭＳ Ｐゴシック', sans-serif;
      }

      .round-count {
        font-size: 48px;
        color: #444;
      }

      .skill {
        margin-left: 10px;
      }

      .message {
        font-size: 12px;
      }

      .statuses {
        display: flex;
        justify-content: space-around;
        margin: 20px 0;
      }

      .team {
        width: 45%;
      }

      .unit {
        margin: 16px 0;
      }

      .unit-name {
        font-weight: bold;
        font-size: 24px;
        color: #666;
      }

      .statusbars {
        margin: 0 10px;
        display: flex;
      }

      .statusbar {
        width: 150px;
        margin: 0 10px;
      }

      .statusbar-desc {
        margin: 0 10px;
        display: flex;
        justify-content: space-between;
      }

      .gauge-wrapper {
        position: relative;
        width: 100%;
```

```
    height: 10px;
    background: #DDD;
    transform: skewX(-30deg);
  }

  .gauge {
    position: absolute;
    height: 10px;
    background: #777;
  }

  .turns {
    margin: 20px 0px 0px 20px;
  }

  .turn {
    padding: 10px;
  }

  .actor {
    font-weight: bold;
    font-size: 20px;
    color: #444;
  }

  .action {
    margin: 10px 0 10px 16px;
  }

  .action-twice {
    font-weight: bold;
    font-size: 16px;
    color: #444;
  }

  .skill-name {
    font-weight: bold;
    font-size: 22px;
    color: #444;
  }

  .damage {
    font-weight: bold;
  }

  .damage-gte50 {
    font-size: 150%;
  }

  .damage-gte100 {
    font-size: 200%;
  }

  .heal {
    font-weight: bold;
  }
```

```
      .clean-up {
        margin-left: 30px;
      }

      .battle-result {
        box-sizing: border-box;
        width: 100%;
        padding: 20px;
        border-top: 1px solid lightgray;
        font-weight: bold;
        font-size: 30px;
      }
    </style>
  </head>
  <body>
    <section class="stage">
      <div class="title"><%- title %></div>
      <section class="description"><%- description %></section>
    </section>
    <%- log %>
  </body>
</html>
```

「explore.ejs」ではステージタイトル(title)、戦闘前に表示される文章(description)、battle.jsで生成された戦闘ログ(log)を受け取って結果となるHTMLを生成します。戦闘ログの内容に関してはbattle.jsが担っているのでここでは付加情報を表示してスタイルを適用するだけになります。

4.14.2 敵キャラクターの作成

次は敵キャラクターを作っていきましょう。これは探索戦からも物語戦からも参照できたほうが便利なので独立したファイルに記述します。「backend/server」内に「enemy.js」を作成してください。ファイル構成は以下のようになります。

```
server
  api.js
  battle.js
  enemy.js
  index.js
  initialize.js
  model.js
  passport.js
  skill.js
  styling.js
```

それでは敵ユニットを作っていきましょう。まずは作成したファイルに以下の内容を記述してください。

```
backend/server/enemy.js
const enemy = {
  /*
  enemyid: {
    name:   敵の名前,
    status: 敵のステータス,
    skill:  敵スキル
  }
  */
  slime: {
    name:     'スライム',
    status:   { atk: 10, dex:  0, mnd:  0, agi:  0, def:  0 },
    skillIds: [1]
  },
  goblin: {
    name:     'ゴブリン',
    status:   { atk: 10, dex: 10, mnd:  0, agi:  0, def: 10 },
    skillIds: [3]
  },
  scorpion: {
    name:     'サソリ',
    status:   { atk: 10, dex:  0, mnd:  0, agi: 20, def:  0 },
    skillIds: [7, 8]
  }
};
```

enemyオブジェクトに敵ユニットを作っていっています。enemyオブジェクトのプロパティ名が各敵ユニットのIDになっていて、そのプロパティの内容に敵ユニットの情報が格納されています。

これを外部から呼び出すことを考えましょう。このenemyオブジェクトをそのまま外部に渡してよしなにしてもらうのも悪くないですが、若干外部での処理が複雑になってしまいます。ではどういったように渡せば処理が簡単になるでしょうか。表現方法は他にもいくらでもあると思うのであくまで一例になりますが、["slime", "slime", "slime", "goblin", "goblin"]というような配列を受け取り「battle.js」にそのまま渡せるような敵ユニット配列に変換して返すようにすれば処理が楽になるのではないでしょうか。

というわけでそんな関数を作っていきましょう。ついでに同じ敵が複数いたときは識別しやすくするために名前を"スライムA"、"スライムB" …とするようにします。enemyオブジェクトの次に以下のコードを追記して保存してください。

```
backend/server/enemy.js
const generateEnemyObjectArray = (enemiesArray) => {
  // ["goblin", "goblin", "slime", "slime"]といった配列から
  // battle.jsに渡すことのできる配列を生成する
  // この際、重複している敵はゴブリンA、ゴブリンBというように名前の最後に識別子をつける

  // エネミーの種類ごとに登場数をチェック
  const repeatedEnemyChecker = {};
  for (let i = 0; i < enemiesArray.length; i++) {
    // 初期値はundefinedなので0扱いにするように
    repeatedEnemyChecker[enemiesArray[i]] = (repeatedEnemyChecker[enemiesArray[i]] || 0) + 1;
  }
```

```
  // battle.jsに渡す形式の配列に変換
  const enemyObjectArray     = [];
  const repeatedEnemyCounter = {};
  for (let i = 0; i < enemiesArray.length; i++) {
    if (2 <= repeatedEnemyChecker[enemiesArray[i]]) {
      // 重複している敵なら名前に識別子をつける
      // 識別子付与にはString.fromCharCodeを使う
      // String.fronCharCode(65) == 'A', String.fronCharCode(66) == 'B' ...

      // 重複のうち今が何個目の重複なのかをカウント
      repeatedEnemyCounter[enemiesArray[i]] = (repeatedEnemyCounter[enemiesArray[i]] || 0) + 1;

      enemyObjectArray.push({
        name:     enemy[enemiesArray[i]].name + String.fromCharCode(repeatedEnemyCounter[enemiesAr
ray[i]] + 64),
        status:   enemy[enemiesArray[i]].status,
        skillIds: enemy[enemiesArray[i]].skillIds
      });
    } else {
      // 重複していなければそのまま配列に格納
      enemyObjectArray.push({
        name:     enemy[enemiesArray[i]].name,
        status:   enemy[enemiesArray[i]].status,
        skillIds: enemy[enemiesArray[i]].skillIds
      });
    }
    repeatedEnemyChecker[enemiesArray[i]] = (repeatedEnemyChecker[enemiesArray[i]] || 0) + 1;
  }

  return enemyObjectArray;
};

module.exports = {
  generateEnemyObjectArray: generateEnemyObjectArray
};
```

処理の手順としてはまずそれぞれの敵ユニットがどれだけ要求されているかをカウントし、repeatedEnemyCheckerに格納しています。これによりどの敵ユニットが重複しているのかを見つけ出します。次にもう一度ループを行い、今の重複数はどれだけか?という情報をrepeatedEnemyCounterでカウントしつつ敵ユニット配列に情報を格納していっています。

このとき、A, B, … などの識別子の付与にはString.fromCharCodeを使っています。String.fromCharCodeは文字コードを示す数値から文字列を生成する関数です。Aは文字コード65、Bは文字コード66、Cは文字コード67…というようになっているので、文字コード64+重複のうち何個目かという値を渡すことで表示したい識別子を生成することができます。

4.14.3 ステージの作成

敵を作ったら次は探索戦のステージを作っていきましょう。探索戦に関連する処理は「explore.js」にまとめることにします。まずは「backend/server」内に「explore.js」を作成してください。ファイル構成は以下のようになります。

```
backend
  config
  node_modules
  server
    api.js
    battle.js
    enemy.js
    explore.js
    index.js
    initialize.js
    model.js
    passport.js
    skill.js
    styling.js
```

作ったファイルにステージを記述していきます。ステージにはどんな情報が必要でしょうか。探索戦はステージごとに必要なクリア回数が設定されていて、それを制覇することでステージが開放されていく仕様になっています。なので、必要クリア回数と「どのステージを制覇すればこのステージは開放されるのか?」という情報を持っておくべきでしょう。それ以外にはステージのタイトル、戦闘前に出る文章、出現する敵も必要でしょう。

また、ステージクリア状況をデータベースに配列で格納する以上ステージ全体は配列で管理したほうがいいでしょう。ここまでまとまったら実際にステージ情報を作っていきます。「backend/server/explore.js」に以下の内容を記述してください。

backend/server/explore.js

```
const stage = [
  /*
    配列のインデックスがそのステージのIDになる。
    {
      title:       タイトル
      description: そのステージの説明文
      enemy:       敵IDの配列
      required:    完全クリアに必要な回数
      condition:   他ステージのクリア状況を受け取り、挑戦可能かどうかをtrue/falseで返す
    }
  */
  /*  0 */ {
    title:       'はじまりの丘',
    description: '気持ちいい風が吹き抜ける草原の丘。',
    enemy:       ["slime", "slime", "slime"],
    required:    5,
    condition: (clearFlags) => {
      return true;
    }
  },
  /*  1 */ {
    title:       '風の渓谷',
    description: 'ひんやりとした空気に支配された谷間。時折強風が吹く。',
```

```
    enemy:       ["slime", "slime", "slime", "goblin"],
    required:    3,
    condition: (clearFlags) => {
      return clearFlags[0]; // はじまりの丘制覇で挑戦可能
    }
  },
  /*  2 */ {
    title:       '灼熱砂漠',
    description: '灼けるような暑さの砂漠。',
    enemy:       ["scorpion", "scorpion", "scorpion", "goblin", "goblin"],
    required:    3,
    condition: (clearFlags) => {
      return clearFlags[1]; // 風の渓谷制覇で挑戦可能
    }
  }
];
```

配列のインデックスがそのままステージのIDになります。titleがステージタイトル、descriptionが戦闘前に出る文章、enemyが敵IDの配列、requiredが制覇に必要なクリア回数になります。また、conditionはどのステージを制覇しているか?という情報をtrue/falseの配列で受け取って挑戦可能かどうかをtrue/falseで返す関数になっています。ここでは「はじまりの丘」は常に挑戦可能、他は前のステージIDを制覇していたら挑戦可能というようにしていますが、conditionを変えることにより○○と××を制覇しないと挑戦できないステージや制覇すると一度に2つ以上のステージが開いたりするステージなどを表現することができます。

これを外部から呼び出すことを考えましょう。外部から呼び出すとき、どのステージに挑戦可能なのか?を返す関数とステージ説明を返す関数があったら便利です。また、どのステージに挑戦可能なのかを処理する際どのステージを制覇済みなのか?を返す関数もあったほうが処理しやすくなるでしょう。それでは実装していきます。先程のコードの次の場所に以下のコードを入力してください。

backend/server/explore.js

```
const getClearFlags = (clearStatus) => {
  // [5, 2]といったクリア回数配列から
  // それぞれのステージを制覇しているかという配列を返す関数
  // trueなら制覇済み、falseなら未制覇

  const clearFlags = [];
  for (let i = 0; i < stage.length; i++) {
    if (!clearStatus[i]) {
      // クリア回数配列に値が存在しなかったら未制覇
      // （未挑戦、1回もクリアしていないなど）
      clearFlags.push(false);
    } else if (clearStatus[i] < stage[i].required) {
      // クリア回数が必要数に届いていなければ未制覇
      clearFlags.push(false);
    } else {
      // どちらでもなければ制覇済み
      clearFlags.push(true);
    }
  }

  return clearFlags;
```

```
};

const getAvailableStageIds = (clearStatus) => {
  // 挑戦可能なステージID配列を返す関数

  const clearFlags = getClearFlags(clearStatus);

  const availableStageIds = [];
  for (let i = 0; i < stage.length; i++) {
    if (stage[i].condition(clearFlags)) {
      availableStageIds.push(i);
    }
  }

  return availableStageIds;
};

const getStageDescriptions = (clearStatus, stageIds) => {
  // クリア状況とステージID配列からステージ説明オブジェクトの配列を返す関数
  // 主にクライアント参照用なのでクライアントから参照してほしくない必要ないデータは返さない
  // （敵の内容など）

  const stageDescriptions = [];
  for (let i = 0; i < stageIds.length; i++) {
    const targetStage = stage[stageIds[i]];

    stageDescriptions.push({
      id:          stageIds[i],                   // ID
      title:       targetStage.title,             // タイトル
      required:    targetStage.required,          // 制覇に必要なクリア回数
      count:       clearStatus[stageIds[i]] || 0, // クリア回数
      isCompleted: targetStage.required <= clearStatus[stageIds[i]]
      // 制覇したかどうか
    })
  }

  return stageDescriptions;
};
```

getClearFlagsはクリア回数配列を受け取ってそれぞれのステージを制覇しているか？という情報をtrue/falseの配列で返す関数です。getAvailableStageIdsはクリア回数配列を受け取ってそれぞれのステージのconditionにgetClearFlagsの結果を渡し、trueが帰ってきたステージ（挑戦可能なステージ）を取得して挑戦可能なステージのIDの配列を返す関数になっています。

getStageDescriptionsはクリア回数配列とステージID配列を受け取ってステージの説明の配列を返す関数です。主にクライアントの探索戦ページから情報を参照するための関数になっているので、敵情報などは返さないようになっています。返す内容はステージID、ステージのタイトル、制覇に必要なクリア回数、制覇済みかどうかです。

4.14.4 探索戦ログの生成

戦闘エンジン・探索戦テンプレート・敵・ステージが全て実装できたので探索戦ログを生成できるようにしましょう。探索戦を外部から呼び出すとき、味方の配列と行き先になるステージIDを受け取って探索戦の結果ログを返

せると便利です。それではそのような関数を実装していきましょう。必要になる外部ファイルを読み込みます。「backend/server/explore.js」の冒頭部分に以下のコードを追加してください。

```
backend/server/explore.js
const fs       = require('fs');
const ejs      = require('ejs');
const battle   = require('./battle.js');
const enemy    = require('./enemy.js');
const template = fs.readFileSync('./template/explore.ejs', 'utf8');
```

fsはファイルを読み書きしたりするためのライブラリです。fsではfs.readFileSyncから同期処理でファイルを読み込むことができます。第1引数に読み込むファイルへのパスを指定し、そのファイルがテキストファイルならそのファイルの文字コードを第2引数に指定します。第1引数に相対パスを指定するとき基準になるファイルは実行しているファイルではなくプロジェクトの場所、ここでいうと「backend/package.json」になるのでパスを記述するときは注意してください。

（なお、ファイルの読み込みを同期処理にしてしまうと読み込み中プログラム全体が止まってしまうので基本的にはfs.readFileSyncではなく非同期処理のfs.readFileを使うべきなのですが、このテンプレートの読み込み処理に関しては起動時に1回だけなので同期処理でもさほど問題ありません。）

さて、ライブラリとテンプレートを読み込んだら実際に探索戦ログを生成する処理を書きます。「backend/server/explore.js」の末尾に以下のコードを追加してください。探索戦を行う関数exploreを定義し、外部から参照したい関数をmodule.exportsに設定しています。exploreではまず「battle.js」に受け取った味方キャラクターの配列とステージの内容に応じた敵キャラクターの配列を渡し戦闘ログを生成しています。受け取った戦闘ログはステージタイトル、ステージ文とともにテンプレートに渡されてレンダリングされ、最終的に戦闘結果とレンダリングされた戦闘ログを返しています。

```
backend/server/explore.js
const explore = (stageId, allies) => {
  // 探索戦を行う関数

  // 戦闘ログを生成
  const result = battle.fight(
    allies,
    enemy.generateEnemyObjectArray(stage[stageId].enemy)
  );

  // EJSで戦闘ログをレンダリング
  const renderedLog = ejs.render(template, {
    title:       stage[stageId].title,
    description: stage[stageId].description,
    log:         result.log
  });

  // 戦闘結果とレンダリングされた戦闘ログを返す
  return {
    result: result.result,
    log:    renderedLog
  };
```

```
};

module.exports = {
  getAvailableStageIds: getAvailableStageIds,
  getStageDescriptions: getStageDescriptions,
  explore: explore
};
```

4.14.5 探索戦ログの保存

今のままだと戦闘を行っても後から参照することができないので、探索戦ログを保存できるようにしていきます。保存の方法はいくつか考えられます。ログをHTMLファイルとして保存するという方法はアクセスしやすく便利ですし、MongoDBは1ドキュメントに16MBまで入るのでそこに入れてしまうという方法も管理がしやすくなるので悪くないでしょう。今回はHTMLファイルとして保存する方法を利用します。

HTMLファイルとして保存するとはいっても、指定のキャラクターが連れ出して戦闘を行った探索戦ログなどを検索できるようにするためにログ情報についてはMongoDBに格納したほうがいいでしょう。まずは探索戦ログのスキーマを定義します。まず必要になるのは行き先、連れ出しキャラクター、パーティーメンバー、戦闘結果、戦闘日時になるでしょう。また、そのログがどこに保存されているのかという情報も必要になってくるでしょう。以上を踏まえてスキーマを作ると以下のようになります。「backend/server/model.js」を開き、MessageSchemaの宣言部分の次に以下のスキーマを記述してください。

backend/server/model.js

```
const LogSchema = new Schema({
  logId:        Number, // ログ ID（サブディレクトリごとに 1 に戻るため重複しうる）
  subdirectory: {type: String, required: true},                // サブディレクトリ名
  stage:        {type: Number, required: true},                // 行き先ステージ ID
  result:       {type: Number, required: true, index: false}, // 戦闘結果 -1:敗北 0:引き分け 1:勝利
  timestamp:    {type: Date, default: Date.now},               // 戦闘日時
  character:    {type: Schema.Types.ObjectId,   ref: 'Character', index: true}, // 連れ出したキャラクター
  party:        {type: [Schema.Types.ObjectId], ref: 'Character', index: true}  // パーティーメンバー
});
```

モデル化と外部から呼び出せるようにする処理も忘れずに行います。末尾部分に以下のようにコードを追記してください。追記した部分は斜体で表示してあります。

backend/server/model.js

```
  （省略）

const Character = mongoose.model('Character', CharacterSchema);
const Room      = mongoose.model('Room',      RoomSchema     );
const Message   = mongoose.model('Message',   MessageSchema  );
const Log       = mongoose.model('Log',       LogSchema      );

mongoose.connect(`mongodb://${config.dbUser}:${encodeURIComponent(config.dbPassword)}@127.0.0.1:${
config.dbPort}/${config.dbName}`);
```

```
module.exports = {
  Character: Character,
  Room:      Room,
  Message:   Message,
  Log:       Log
};
```

次にログの保存場所を作ります。今回は「/var/www/teikiadv/explore/log」にログファイルを保存することにしましょう。まずはログの保管場所を設定ファイルに記述しておきましょう。「backend/config/default.json」を開き以下の設定を追記してください。

backend/config/default.json

```
  "logDirectory": "/var/www/teikiadv/explore/log"
```

次は実際にディレクトリを作ります。Tera Termでログインしrootユーザーになってください。rootでログインできたら以下のコマンドを順番に実行します。これで「/var/www」ディレクトリがroot権限で作成され、その中にteiki権限で「teikiadv/explore/log」ディレクトリが作成されます。

「cd /var」
「mkdir -p www/teikiadv/explore/log」
「cd www」
「chown -R teiki:teiki teikiadv」

実際にログを保存する際は日付ごとにサブディレクトリを作ってその中にログファイルを保存します。ディレクトリを分けずにそのままログ保存ディレクトリ内にログファイルを保管し続けていると最終的には何万ものファイルが1ディレクトリ内に存在するようになり、管理が非常に面倒になったりアクセス性能が低下したりするからです。

4.14.6 探索戦APIの作成

それでは探索戦のAPIを作成します。必要な設定を「backend/config/default.json」に追記します。以下の設定を追加してください。ここではPT人数の上限とステージを制覇したときに得られるNPの量を設定しています。

backend/config/default.json

```
  "partyMembersMax": 5,
  "exploreCompleteReward": 10
```

次に必要なライブラリなどを読み込みましょう。「backend/server/api.js」を開き、冒頭部分を以下のように書き換えてください。

backend/server/api.js

```
const express    = require('express');
const config     = require('config');
```

```
const passport  = require('passport');
const fs        = require('fs').promises;
const styling   = require('./styling.js');
const skill     = require('./skill.js');
const explore   = require('./explore.js');
const Character = require('./model.js').Character;
const Room      = require('./model.js').Room;
const Message   = require('./model.js').Message;
const Log       = require('./model.js').Log;
const router    = express.Router();

  （省略）
```

ライブラリを読み込んだら探索戦の実行に必要な情報を返すAPIを実際に作ります。AP、挑戦可能なステージの情報と連れ出せる味方（＝お気に入りしている味方）の情報を返す必要があるでしょう。ここでは「/characters/main/explore」にGETリクエストすれば情報を受け取れることにします。APIを追加してください。

backend/server/api.js

```
router.get('/characters/main/explore', checkAuthentication, async (req, res) => {
  try {
    const leader = await Character.findById(req.user._id, {
      eno:      1,
      nickname: 1,
      mainicon: 1,
      status:   1,
      summary:  1,
      fav:      1,
      ap:       1,
      explore:  1
    }).populate({
      path: 'fav',
      model: 'Character',
      select: {
        _id: 0,
        eno:      1,
        nickname: 1,
        mainicon: 1,
        status:   1,
        summary:  1,
      }
    });

    // 挑戦可能なステージ情報を受け取る
    const availableStageIds = explore.getAvailableStageIds(leader.explore);
    const availableStages   = explore.getStageDescriptions(leader.explore, availableStageIds);

    return res.status(200).send({
      leader: {
        ap:       leader.ap,
        eno:      leader.eno,
        nickname: leader.nickname,
        mainicon: leader.mainicon,
        status:   leader.status,
        summary:  leader.summary
```

```
      },
      stages:     availableStages,
      characters: leader.fav
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

ユーザーとお気に入りしているキャラクターをデータベースから取得してそこから挑戦可能なステージ情報を取得して返しています。ここについては非常にシンプルなので特に解説する内容はありません。

次は実際に探索戦を行うAPIを作ります。以下のAPIを追記して保存してください。「/characters/main/explore」にPOSTリクエストをすることで戦闘が行えるようになっています。

backend/server/api.js

```
router.post('/characters/main/explore', checkAuthentication, checkCsrf, async (req, res) => {
  try {
    if (
      typeof req.body.stage != 'number' ||
      !Array.isArray(req.body.party)
    ) {
      return res.status(400).send(); // 要求の内容の型がおかしければ400を返し中断
    }

    // 同じキャラクターを重複して連れ出そうとしていないかをチェック
    // ついでに型がおかしくないかをチェック
    const enoesChecker = {};
    for (let i = 0; i < req.body.party.length; i++) {
      if (typeof req.body.party[i] != 'number') {
        // partyに入っているENoがNumber型でなければ400を返して中断
        return res.status(400).send();
      }

      if (!enoesChecker[req.body.party[i]]) {
        enoesChecker[req.body.party[i]] = true;
      } else if (enoesChecker[req.body.party[i]]) {
        // PTが重複していたとき400を返して中断
        return res.status(400).send();
      }
    }

    const leader = await Character.findById(req.user._id, {
      ap:       1,
      nickname: 1,
      status:   1,
      skill:    1,
      fav:      1,
      explore:  1
    }).populate({
      path: 'fav',
      model: 'Character',
      select: {
```

```
      eno:      1,
      nickname: 1,
      status:   1,
      skill:    1
    }
  });

  // APがない場合は400を返して中断
  if (leader.ap <= 0) {
    return res.status(400).send();
  }

  // 行けないはずのステージに行こうとしていた場合は400を返し中断
  const availableStageIds = explore.getAvailableStageIds(leader.explore);
  if (!availableStageIds.includes(req.body.stage)) {
    return res.status(400).send();
  }

  // お気に入りしているキャラクターのENoの配列を生成
  const favEnoes = leader.fav.map(character => character.eno);

  // お気に入りしていないキャラクターを連れ出そうとしていたら400を返して中断
  for (let i = 0; i < req.body.party.length; i++) {
    if (!favEnoes.includes(req.body.party[i])) {
      return res.status(400).send();
    }
  }

  // 味方チームを生成
  const allies = [];

  // 味方チームにリーダーを追加
  allies.push({
    name:     leader.nickname,
    status:   leader.status,
    skillIds: leader.skill
  });

  // 味方チームに連れ出しメンバーを追加
  // ついでにパーティーメンバーの_idリストを作る
  const partyDocumentIds = [];
  for (let i = 0; i < req.body.party.length; i++) {
    for (let j = 0; j < leader.fav.length; j++) {
      if (req.body.party[i] == leader.fav[j].eno) {
        // PTのENoをfavリストから照合して合致すれば
        // そのキャラクターのステータス情報を味方チームに追加
        allies.push({
          name:     leader.fav[j].nickname,
          status:   leader.fav[j].status,
          skillIds: leader.fav[j].skill
        });
        partyDocumentIds.push(leader.fav[j]._id);
      }
    }
  }

  // 戦闘を行い結果を生成
```

```
    const result = explore.explore(req.body.stage, allies);

    // 敗北であればresultIdを-1、引き分けなら0、勝利なら1に
    // （データベース保存用）
    let resultId = -1;
    if (result.result == 'even') {
      resultId = 0;
    } else if (result.result == 'win') {
      resultId = 1;
    }

    // 現在日時を取得し保存するべき場所になるサブディレクトリを求める
    // 例えば2019年12月1日ならサブディレクトリ名は"191201"
    const currentDate  = new Date();
    const subdirectory = (
      (''  + (currentDate.getFullYear() )).slice(-2) +
      ('0' + (currentDate.getMonth() + 1)).slice(-2) +
      ('0' + (currentDate.getDate()     )).slice(-2)
    );

    // ログドキュメントを作成し保存
    // 検索しやすくするためPTメンバー情報に連れ出したキャラクターを含める
    const log = new Log({
      subdirectory: subdirectory,
      stage:        req.body.stage,
      result:       resultId,
      character:    leader._id,
      party:        [leader.id].concat(partyDocumentIds)
    });
    await log.save();

    // 同じサブディレクトリのログを全件取得
    const logs = await Log.find({subdirectory: subdirectory}, {_id: 1});

    // そのログのうち保存したログが何番目に該当するか検索
    // それによりlogidをセット
    let logId = 0;
    for (let i = logs.length - 1; 0 <= i; i--) {
      if (logs[i]._id.toString() == log._id.toString()) {
        logId = i + 1; // ログIDは1から始まるようにする
        await log.update({logId: logId});
        break;
      }
    }

    if (!logId) {
      // 保存したはずのログ情報が全件ログになければ500を返して中断
      return res.status(500).send();
    }

    try {
      // 指定のサブディレクトリを作る
      await fs.mkdir(`${config.logDirectory}/${subdirectory}`);
    } catch (e) {
      // すでにサブディレクトリがある場合はエラーになる
      // これは想定内のエラーなのでそのまま進む
      // サブディレクトリがすでに存在する以外の理由であればそのまま例外をスロー
```

```
      if (e.code != 'EEXIST') {
        throw e;
      }
    }

    // ログファイルを保存
    await fs.writeFile(`${config.logDirectory}/${subdirectory}/${logId}.html`, result.log);

    // 情報更新用のオブジェクトを作成
    const query = {$inc: {ap: -1}};

    // 勝利していた場合はそのステージのクリア回数を+1
    if (resultId == 1) {
      const clearStatus = leader.explore;

      // 該当のステージが現状のクリア配列の外のIDの場合
      // クリア回数0で埋めつつ配列を伸ばしていく
      if (clearStatus.length < req.body.stage) {
        for (let i = clearStatus.length; i <= req.body.stage; i++) {
          clearStatus[i] = 0;
        }
      }

      clearStatus[req.body.stage] = (clearStatus[req.body.stage] || 0) + 1;
      query.explore = clearStatus;

      // ステージ情報を取得
      const stageDescription = explore.getStageDescriptions(clearStatus, [req.body.stage])[0];

      // 今回のクリアで制覇になった場合は
      // NP報酬を獲得
      if (stageDescription.required == stageDescription.count) {
        query.$inc.np = config.exploreCompleteReward;
      }
    }

    // 情報を更新
    await leader.update(query);

    // 保存されたログの場所を返す
    return res.status(200).send({
      subdirectory: subdirectory,
      logId:        logId
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

長いですが少しずつ読み解いていきましょう。最初に入力された内容がおかしくないかをチェックしています。許容される値の範囲が複雑なので長くなってしまっていますが基本的にはおかしいデータがあれば400を返して弾くだけです。

特におかしい内容がなければ「explore.js」にわたす味方ユニットのオブジェクト配列を生成していきます。まず

はリーダーとしてユーザーのキャラクターを追加します。次に連れ出したPTメンバーも追加していきます。リーダーキャラクターを取得するときにお気に入りキャラクターの戦闘情報を取得しているので、連れ出したPTメンバーはそこから該当するデータを探してオブジェクトに設定しています。このとき、PTメンバーのデータベースのドキュメントIDリストも一緒に作っておきます。

味方ユニット配列ができたら「explore.js」にステージIDと一緒に渡して戦闘を行わせ、その結果を受け取ります。このとき戦闘結果は"win"，"lose"，"even"のうちその結果に該当するものが返されるのですが、データベースに保管するためにここから勝利であれば1、引き分けであれば0、敗北であれば-1という値を作っておきます。

次はログを保管していきます。ここでは日付をフォルダー名、1，2，3 …といったログIDをファイル名としてファイルを保管していきます。まずは現在の日付からフォルダー名を作成しています。2019年12月1日であれば「191201」といったように、年月日を表すそれぞれ2桁の数字を連結したものをフォルダー名としています。

これを他の情報と合わせてログ情報としてとりあえずMongoDBに保管します。保管したらENoやRNoと同じ要領でログIDを割り振ります。これで保存するべきログIDが求まります。

次は実際にファイルを保管します。まずは実際にすでにその日付のフォルダーがあるかないかに関わらずフォルダーを作ろうとしてみます。このときすでにその日付のフォルダーがあればEEXISTエラーになりますがこれは想定内のエラーなのでそのまま処理を進めていきます。（もちろんそれ以外のエラーが発生していれば異常な状態なのでそれは例外をスローして中断します。）フォルダーができたら指定のログIDでファイルを保存します。

最後にユーザーの情報更新を行います。これは更新内容をオブジェクトに組み立てていきます。まずは基本の更新内容としてAP-1を設定します。勝利していた場合はさらに探索戦クリア状況を更新し、その上で今回のクリアでステージ制覇になっていた場合はNPに規定の報酬量を加算します。なお、探索戦クリア状況の配列が指定のステージIDの箇所に届かない場合、クリア回数0でデータを埋めながら配列を延長していっています。更新内容が組み上がったらユーザーの情報を更新します。

最後にログがどこに保存されたのかという情報を返せば処理は完了です。

4.14.7 Nginxの設定変更

保存したログを外部からアクセスできるようにしましょう。Tera Termを起動しログイン後「su」コマンドを実行しパスワードを入力してrootユーザーになってください。rootになったら「vi /etc/nginx/conf.d/default.conf」を実行してください。テキストエディタが立ち上がるので、「location /ta/api/」の項目の次のところに以下の設定を追記し保存、「nginx -s reload」を実行します。ペーストは右クリックから行えます。

/etc/nginx/conf.d/default.conf

```
location /ta/log/ {
    alias  /var/www/teikiadv/explore/log/;
}
```

これで「http://dev.siroisakana.com/ta/log/(日付)/(ログID).html」というURLでログにアクセスできるようになります。

4.14.8 APの配布

探索戦ができるようになったとはいえAPがないと探索戦を行うことができないので、次はAPを配布できるように

しましょう。まずは設定ファイルにAPに関する設定を記述します。「backend/config/default.json」を開き、以下の内容を書き加えてください。

backend/config/default.json

```
  "givingAp": true,
  "maxAp": 15
```

maxApはAP上限で、givingApはAPを配布するかどうかの設定です。givingApは例えば事前登録期間など何らかの理由でAPを配布したくないときにfalseにします。それではAP配布の処理を書いていきましょう。「backend/server」内に「ap.js」を作成し以下の内容を記述して保存して下さい。

backend/server/ap.js

```
const config    = require('config');
const Character = require('./model.js').Character;

const main = async () => {
  // AP配布が有効なら
  if (config.givingAp) {
    // 現在APが最大AP未満のキャラにAPを1配布する
    // 最大APが15ならAPが0～14のキャラにAPを配布するということ
    await Character.updateMany({ap: {$lt: config.maxAp}}, {$inc: {ap: 1}});
  }

  // 処理が終わったら終了する
  process.exit(0);
};

main();
```

APがAP上限に達していないキャラクターにAPを1配布するだけのプログラムになっています。次はこれをコマンドラインから「npm run ap」で実行できるようにしましょう。「backend/package.json」のscriptの項目に以下の内容を追記して保存します。

backend/package.json

```
    "ap": "node server/ap.js"
```

ここまで実装したらためしにAPを配布してみましょう。Visual Studio Code上で新しいターミナルを起動して「cd /home/teiki/nodejs/teikiadv/backend」を実行し、続けて「npm run ap」を実行します。

```
1: npm, npm, bash

[teiki@                backend]$ cd /home/teiki/nodejs/teikiadv/ba
ckend
[teiki@                backend]$ npm run ap
```

色々と表示が出た後に処理が完了します。MongoDB Compassからキャラクターのを見てみましょう。APが増えていれば成功です。AP配布に使ったターミナルは閉じても構いません。

これでAPを配布することができるようになりましたが、これを毎日手動でやるのは無理があるので毎日決まった時間にAPが自動で配布されるようにしましょう。これには**cron**を利用します。cronは指定した時間に指定したプログラムを定期的に実行してくれるもので、設定には**crontab**というコマンドを使用します。

それでは設定していきましょう。Visual Studio Codeで新しくターミナルを開き「crontab -e」を実行します。ターミナル上でテキストエディタが起動するので、viコマンドのテキストエディタと同じ要領で以下の内容を入力して保存します。

crontab -e

```
0 20 * * * cd /home/teiki/nodejs/teikiadv/backend; npm run ap
```

cronでは左から順番に分、時間、日、月、曜日の順番で実行したいタイミングを指定していきます。特に限定しない箇所では*を入力します。この例では「毎日20:00」が指定されています。このとき誤って「* 20 * * *」と指定しないように注意してください。「毎日20時の間毎分」という意味になってしまいます。不安な場合は以下のサイトで正しいスケジュールが組めているか確認できます。

crontab.guru - the cron schedule expression editor
https://crontab.guru/

さて、その後に続けて実行したいコマンドを指定します。ここではcdでディレクトリを移動して「npm run ap」を実行しAPを配布しています。ここについて特別解説は必要ないでしょう。これによって毎日20:00にAPを配布できるというわけです。なお、スケジュールのタイミングはVPSの時刻に依存します。VPSの時刻が日本標準時ではなく協定世界時などの場合は注意してください。

4.14.9 探索戦ページの作成

ここまで実装したら探索を実行するページを作成していきましょう。まずは設定ファイルにPT上限人数を設定します。「frontend/nuxt.config.js」を開き、envに以下の項目を追記します。

frontend/nuxt.config.js

```
  partyMembersMax: 5
```

次にページに必要なコンポーネントを作っていきます。探索戦ではお気に入りしているキャラクターを連れ出すことができます。なので、キャラクターを選択できるようなコンポーネントがあるといいでしょう。コンポーネント化することで物語戦の宣言ページでも使い回すことができます。では作りましょう。「frontend/components」内に「SelectableCharacterList.vue」を作成し、以下の内容を記述して保存してください。

frontend/components/SelectableCharacterList.vue

```
<template>
  <div class="character-list">
    <div
      class="character"
      v-for="character in characters"
      :key="character.eno"
      @click="select(character)">
      <div class="icon">
        <character-icon :src="character.mainicon"/>
      </div>
      <div class="description">
        <a
          :href="`${$router.options.base}profile/${character.eno}`"
          target="_blank"
          class="profile-link">
          <span class="name">{{ character.nickname }}</span>
          <span class="eno">&lt; ENo.{{character.eno}} &gt;</span>
        </a>
        <table class="statuses">
          <tbody>
            <tr>
              <td class="status-name">ATK</td>
              <td class="status-value">{{ character.status.atk }}</td>
              <td class="status-name">DEX</td>
              <td class="status-value">{{ character.status.dex }}</td>
              <td class="status-name">MND</td>
              <td class="status-value">{{ character.status.mnd }}</td>
              <td class="status-name">AGI</td>
              <td class="status-value">{{ character.status.agi }}</td>
              <td class="status-name">DEF</td>
              <td class="status-value">{{ character.status.def }}</td>
            </tr>
          </tbody>
        </table>
        <div class="summary">
          {{ character.summary }}
        </div>
      </div>
    </div>
  </div>
</template>

<script>
import CharacterIcon from '~/components/CharacterIcon.vue'

export default {
  components: {
    CharacterIcon
  },
  props: ['characters'],
  methods: {
    select: function(character) {
      this.$emit('select', character);
    }
  }
}
```

```
</script>

<style lang="scss" scoped>
$font: 'BIZ UDPGothic', 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ', Meiryo, 'ＭＳ Ｐゴシック', sans-serif;

.character-list {

  .character {
    display: flex;
    padding: 10px;
    border: 1px solid lightgray;
    margin: 10px;
    border-radius: 4px;
    cursor: pointer;

    .description {
      display: block;
      margin-left: 10px;

      .profile-link {
        text-decoration: none;
      }

      .name {
        font-size: 16px;
        font-weight: bold;
        color: #222222;
      }

      .eno {
        font-size: 13px;
        margin-left: 3px;
        color: gray;
      }

      .statuses {
        margin: 0;

        th, td {
          padding: 0 8px 0 0;
          border: none;
          font-family: $font;
        }

        .status-name {
          font-weight: bold;
          color: #555;
          margin-right: 20px;
        }
      }
    }
  }
}
</style>
```

キャラクターの配列を受け取って表示するコンポーネントになっています。また、それぞれのキャラクターをクリックしたときにselectイベントとともに選択されたキャラクター情報が帰るようになっています。

それではこれを利用して探索戦ページを作りましょう。「frontend/pages」内に「explore.vue」を作成し、以下の内容を入力して保存します。

frontend/pages/explore.vue

```
<template>
  <section>
    <sub-heading>残りAP</sub-heading>
    <div class="remaining-ap">
      <span :class="{'remaining-ap-zero': !leader.ap}">残りAP: {{ leader.ap }}</span>
    </div>
    <sub-heading>ステージ選択</sub-heading>
    <section class="form">
      <section class="form-description">
        探索戦の行き先を選択します。<br>
        ステージを規定の回数クリアするとコンプリートとなり新たなステージが解放されます。
      </section>
      <div class="stage-selector">
        <select class="stage-select" v-model="stage">
          <option :value="null">-- ステージを選択 --</option>
          <option v-for="stageOption in stages" :key="stageOption.id" :value="stageOption">
            {{ stageOption.id }}. {{ stageOption.title }}{{ stageOption.isCompleted ? '（済）
' : '' }}
          </option>
        </select>
        <div class="stage-requires" v-if="stage">
          コンプリート条件：{{ stage.count }} / {{ stage.required }}<span v-if="stage.isCompleted">
（コンプリート済）</span>
        </div>
      </div>
    </section>
    <sub-heading>パーティーメンバー選択</sub-heading>
    <section class="form">
      <div class="form-description">
        連れ出すパーティーメンバーを選択します。<br>
        お気に入りしているキャラクターの中から選択することができます。<br>
        パーティーメンバーはあなたを含めて最大{{ partyMembersMax }}人までです。
      </div>
      <div class="form-title">
        パーティーメンバー（クリックで選択キャンセル）
      </div>
      <selectable-character-list
        :characters="[leader].concat(party)"
        @select="unselect"/>
      <div class="form-title">
        連れ出せるキャラクター
      </div>
      <selectable-character-list
        v-if="characters.length"
        :characters="characters"
        @select="select"/>
      <div v-else class="form-description">
        お気に入りしているキャラクターがいません。<br>
```

```
        お気に入りは<nuxt-link to="/list">キャラクター一覧</nuxt-link>からキャラクターページにアクセスし
<br>
        「お気に入りする」ボタンをクリックすることで行なえます。
      </div>
    </section>
    <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
    <div class="button-wrapper">
      <button class="button" @click="explore">探索実行</button>
    </div>
  </section>
</template>

<script>
import SubHeading              from '~/components/SubHeading.vue'
import MessageBanner           from '~/components/MessageBanner.vue'
import SelectableCharacterList from '~/components/SelectableCharacterList.vue'

export default {
  components: {
    SubHeading,
    MessageBanner,
    SelectableCharacterList
  },
  head() {
    return {
      title: '探索'
    };
  },
  data() {
    return {
      partyMembersMax: process.env.partyMembersMax,

      stage: null,
      party: [],
      errorMessage:    '',
      waitingResponse: false
    };
  },
  asyncData: async function(context) {
    const response = await context.$axios.get('/api/characters/main/explore');
    return {
      leader:     response.data.leader,
      stages:     response.data.stages,
      characters: response.data.characters
    };
  },
  methods: {
    select: function(character) {
      if (this.partyMembersMax <= this.party.length) {
        // PT人数上限に届いていれば中断
        return;
      }

      for (let i = 0; i < this.party.length; i++) {
        if (character.eno == this.party[i].eno) {
          return; // すでに追加されているキャラクターであれば中断
        }
```

```
      }

      this.party.push(character);
    },
    unselect: function(character) {
      for (let i = 0; i < this.party.length; i++) {
        if (character.eno == this.party[i].eno) {
          return this.party.splice(i, 1);
          // パーティーメンバー配列から指定のキャラクターを取り除く
          // パーティーメンバー配列にリーダーは入っていないので
          // リーダーは取り除かれない
        }
      }
    },
    explore: async function() {
      if (this.waitingResponse) {
        // 接続待ち状態であればアラートを表示しreturn、以降の処理を行わない
        return alert('しばらくお待ち下さい');
      }

      // 入力内容の検証を行う、問題があればエラーメッセージを表示し処理を中断
      if (this.leader.ap <= 0) {
        return this.errorMessage = 'APがありません';
      }
      if (this.stage == null) {
        return this.errorMessage = 'ステージが選択されていません';
      }

      // 問題がなければ接続に入る、接続待ち状態をON(true)に
      this.waitingResponse = true;

      // 連れ出すキャラクターをENo配列に変換
      const partyEnoes = [];
      for (let i = 0; i < this.party.length; i++) {
        partyEnoes.push(this.party[i].eno);
      }

      try {
        // 探索戦を実行
        const response = await this.$axios.post('/api/characters/main/explore', {
          csrf:  this.$store.getters['auth/loginCharacter'].csrf,
          stage: this.stage.id,
          party: partyEnoes
        });

        // 正常にログが生成されたらログページへ移動
        location.href = `${this.$router.options.base}log/${response.data.subdirectory}/${response.
data.logId}.html`;
      } catch (e) {
        // エラーが発生した場合エラーメッセージを表示して接続待ち状態をOFF(false)に
        this.errorMessage = '実行中にエラーが発生しました';
        this.waitingResponse = false;
      }
    }
  }
}
</script>
```

```
<style lang="scss" scoped>
$font: 'BIZ UDPGothic', 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ', Meiryo, 'ＭＳ Ｐゴシック', sans-serif;

.remaining-ap {
  padding: 0 30px;
  font-weight: bold;
  font-family: $font;
  font-size: 24px;
  color: #666;

  .remaining-ap-zero {
    color: #c53f4d;
  }
}

.stage-selector {
  padding: 20px 10px;
  display: flex;
  align-items: center;

  .stage-select {
    margin: 0;
  }

  .stage-requires {
    margin-left: 10px;
  }
}

.form-title {
  margin: 20px 0;
}
</style>
```

保存してページを表示すると以下のような感じになっているはずです。

それではコードの流れを追ってみましょう。まずasyncData内で探索戦APIにアクセスし、自身のデータ、挑戦できるステージのデータ、連れ出せるキャラクターたちのデータを受け取っています。また、data内で現在選択しているステージ、現在選択しているパーティーメンバーの変数を用意しています。

まずはステージ選択について見ていきましょう。<template>内の<div class="stage-selector">内でv-forを利用してstagesの情報を元に挑戦できるステージのリストを作っています。このとき、ステージを選択していない状態(stage = null)のときは「-- ステージを選択 --」が選択されるようにします。あとは<select>にv-model="stage"とすることでステージ選択のUIを作ることができます。

次にキャラクター選択を見ていきます。これは連れ出せるキャラクターのリスト(クリックで連れ出す)とPTメンバーのリスト(クリックで連れ出しキャンセル)の2つのリストで構成されています。前者は選択された際にselectメソッドが呼ばれるようになっており、PT上限を超えたり対象のキャラクターがすでに追加されていたりしなければPTメンバーに対象を追加します。後者は選択された際にunselectが呼ばれるようになっとり、対象のキャラクターをPTメンバーから除外するようになっています。

あとは探索実行ボタンを押した際に入力内容を検証しAPIにアクセスして探索戦を実行、出力された場所を受け取ってそのページに移動するだけです。試しに探索実行を押してみると以下のような感じになるはずです。

4.14.10 ログ検索API

ログが出力できるようになったので、次はログを検索できるようにしましょう。まずは検索の仕様を考えます。自身がリーダーのログを検索するのが基本となりますが、他のENoのキャラクターがリーダーのログやリーダーではないものの連れ出されているログを調べられると便利だと思われるのでそれも仕様に組み込みましょう。それではAPIを作っていきます。最初にAPIに関わる部分の設定を記述します。「backend/config/default.json」を開き、以下の項目を追記して下さい。1ページあたりのログ検索結果の数を記述しています。

backend/config/default.json
```
"logsPerPage": 20
```

設定が記述できたら「backend/server/api.js」を開き以下のAPIを追記してください。

backend/server/api.js
```
router.get('/logs', checkAuthentication, async (req, res) => {
  try {
    let target;
    if (!req.query.eno) { // enoの指定がなければアクセスしているユーザーのキャラを対象に
      target = req.user._id;
    } else { // enoの指定があればそのキャラを対象に
      if (/^[1-9][0-9]*$/.test(req.query.eno)) {
        const targetDoc = await Character.findOne({eno: Number(req.query.eno)}, {_id: 1});
        // 指定のENoを検索、このときは削除されているかは問わない

        if (!targetDoc) { // 指定のenoのキャラが見つからなければ検索結果0で返す
          return res.status(200).send({
            list:         [],
```

```
        isContinuing: false
      });
    }

    target = targetDoc._id; // 指定のenoのキャラを対象に
  } else { // enoの指定内容がおかしければ検索結果0で返す
    return res.status(200).send({
      list:         [],
      isContinuing: false
    });
  }
}

const query = {};
if (req.query.p) {           // PT参加表示モードなら
  query.party     = target; // 検索条件に「対象がPTにいる」を追加
} else {                     // リーダー表示モードなら
  query.character = target; // 検索条件に「対象がリーダー」を追加
}

const currentPage = Number(req.query.page);

let logs = await Log.find(query, {
  logId:        1,
  subdirectory: 1,
  stage:        1,
  result:       1,
  timestamp:    1,
  character:    1,
  party:        1
}).sort({timestamp: -1}).skip(currentPage * config.logsPerPage).limit(config.logsPerPage + 1).populate({
  path:  'character',
  model: 'Character',
  select: {
    _id:      0,
    mainicon: 1,
    eno:      1
  }
}).populate({
  path:  'party',
  model: 'Character',
  select: {
    _id:      0,
    mainicon: 1,
    eno:      1
  }
}).exec();

let isContinuing = false;
if (config.logsPerPage < logs.length) {
  isContinuing = true;
  logs = logs.slice(0, config.logsPerPage);
}

// ステージのタイトル情報を取得
const stageArray = logs.map(log => log.stage);
const stageInfos = explore.getStageDescriptions([], stageArray);
```

```
    // APIが返却する形に変換
    const logArray = [];
    for (let i = 0; i < logs.length; i++) {
      logArray.push({
        url:       `${logs[i].subdirectory}/${logs[i].logId}.html`,
        stage:     stageInfos[i].title,
        result:    logs[i].result,
        timestamp: logs[i].timestamp,
        character: logs[i].character,
        party:     logs[i].party
      });
    }

    return res.status(200).send({
      list:         logArray,
      isContinuing: isContinuing
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

このAPIは以下のURLパラメーターを受け取ります。

eno	検索対象のENo、指定のない場合アクセスしているユーザーのキャラクターが検索対象
p	PTモード検索、有効でない場合検索対象がリーダーのログを検索

あとはキャラクターの検索などとほぼ変わりません。最後に、データベースにはステージ情報がステージIDの形式で保存されているのでこれをステージタイトルにするようにしてから結果を返します。

4.14.11 ログ検索ページ

APIができたのでログ検索ページを作っていきます。まずはログがどのようなURLからアクセスできるかの情報を設定に記載しておきましょう。「frontend/nuxt.config.js」を開き、envに以下の項目を追加してください。このときドメイン名だけでなくhttps化しているかどうかによってhttp、httpsを書き分ける必要があるので注意してください。

frontend/nuxt.config.js

```
    logDirectory: 'https://dev.siroisakana.com/ta/log/'
```

次にログリストのコンポーネントを作ります。「frontend/components」内に「LogList.vue」を作成し、以下の内容を記述して保存してください。

frontend/components/LogList.vue

```
<template>
  <table class="logs">
    <tr v-for="(log, index) in logs" :key="index">
```

```
      <td class="title">
        <a :href="`${logDirectory}${log.url}`" target="_blank">{{ log.stage }}</a>
      </td>
      <td class="leader">
        <nuxt-link
          :to="`profile/${log.character.eno}`"
          class="character">
          <character-icon :src="log.character.mainicon"/>
        </nuxt-link>
      </td>
      <td class="party">
        <div class="characters-wrapper">
          <nuxt-link
            v-for="party in log.party"
            :key="party.eno"
            :to="`profile/${party.eno}`"
            class="character">
            <character-icon :src="party.mainicon"/>
          </nuxt-link>
        </div>
      </td>
      <td class="date">
        {{ log.timestamp | date }}
      </td>
      <td class="result">
        <span v-if      ="log.result == -1">敗北</span>
        <span v-else-if="log.result ==  0">引き分け</span>
        <span v-else                       >勝利</span>
      </td>
    </tr>
  </table>
</template>

<script>
import CharacterIcon from '~/components/CharacterIcon.vue'

export default {
  components: {
    CharacterIcon
  },
  props: ['logs'],
  data() {
    return {
      logDirectory: process.env.logDirectory
    };
  },
  filters: {
    date: (str) => {
      const date = new Date(str);
        return (
          ('0' + (date.getMonth() + 1)).slice(-2) + '/' +
          ('0' + (date.getDate()     )).slice(-2) + '\n' +
          ('0' + (date.getHours()    )).slice(-2) + ':' +
          ('0' + (date.getMinutes()  )).slice(-2)
        );
    }
  }
```

```
}
</script>

<style lang="scss" scoped>
.logs {
  margin: 0;

  td {
    padding: 4px;
  }

  .title {
    padding-left: 20px;
    width: 100%;
  }

  .leader {
    padding-right: 10px;
  }

  .characters-wrapper {
    display: flex;
  }

  .character {
    padding: 4px;
  }

  .result {
    width: 75px;
    padding: 0 20px;
    text-align: center;
    word-break: keep-all;
  }
}
</style>
```

このコンポーネントはログ情報の配列を受け取ってリストで表示するだけのコンポーネントです。戦闘ログが生成された日付を表示するためfiltersにdateを定義している他、ログにアクセスできるようにするため先程設定したログディレクトリの値を使ってリンクを表示しています。

あとはこのコンポーネントを使ってページを表示するだけです。「frontend/pages」内に「log.vue」を作成して以下の内容を入力して保存してください。ページング関連の処理はありますが、ほぼAPIからログ情報を受け取ってコンポーネントに渡しているだけです。

frontend/pages/log.vue

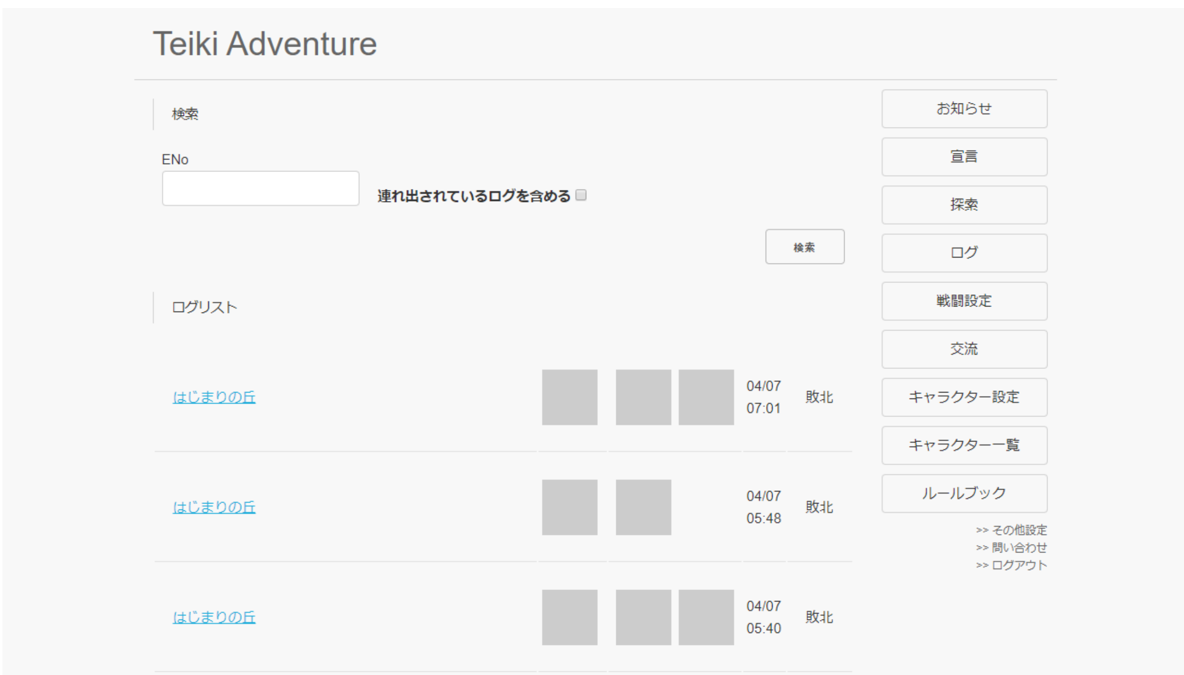
```
<template>
  <section>
    <section>
      <sub-heading>検索</sub-heading>
      <div class="search">
        <div class="search-form">
          <div class="search-target">
```

```
            ENo
          </div>
          <input type="text" v-model="form.eno">
        </div>
        <div class="search-form">
          <label>連れ出されているログを含める <input type="checkbox" v-model="form.party"></label>
        </div>
      </div>
      <div class="button-wrapper">
        <nuxt-link :to="form | queries"><button class="button">検索</button></nuxt-link>
      </div>
    </section>
    <section>
      <sub-heading>ログリスト</sub-heading>
      <div v-if="list.length">
        <log-list :logs="list"/>
        <div class="pagelink-wrapper">
          <nuxt-link
            class="pagelink"
            v-if="Number($route.query.page)"
            :to="`${$route.path}${queryWithoutPage}&page=${Number($route.query.page) - 1}`">
            前のページへ
          </nuxt-link>
          <nuxt-link
            class="pagelink"
            v-if="isContinuing"
            :to="`${$route.path}${queryWithoutPage}&page=${Number($route.query.page || 0) + 1}`">
            次のページへ
          </nuxt-link>
        </div>
      </div>
      <div v-else>
        検索結果はありません。
      </div>
    </section>
  </section>
</template>

<script>
import SubHeading from '~/components/SubHeading.vue'
import LogList    from '~/components/LogList.vue'

export default {
  components: {
    SubHeading,
    LogList
  },
  middleware: 'authenticated',
  watchQuery: true,
  head() {
    return {
      title: 'ログリスト'
    }
  },
  data() {
    return {
      form: {
```

```
        eno:   this.$route.query.eno,
        party: this.$route.query.p == 't'
      }
    };
  },
  asyncData: async function(context) {
    const queries  = context.route.fullPath.slice(context.route.path.length);
    const response = await context.$axios.get(`/api/logs${queries}`);

    return {
      list:          response.data.list,
      isContinuing: response.data.isContinuing
    }
  },
  filters: {
    queries: (form) => {
      const queries = [];
      if (form.eno)   queries.push(`eno=${encodeURIComponent(form.eno)}`);
      if (form.party) queries.push('p=t');
      return queries.length ? '?' + queries.join('&') : '';
    }
  },
  computed: {
    queryWithoutPage: {
      get() {
        const queries = [];
        for (const prop in this.$route.query) {
          if (prop != 'page') {
            queries.push(`${prop}=${this.$route.query[prop]}`);
          }
        }
        return '?' + queries.join('&');
      }
    }
  }
};
</script>

<style lang="scss" scoped>
.search {
  display: flex;
  align-items: flex-end;

  .search-form {
    margin: 0 10px;
  }
}

.pagelink-wrapper {
  padding: 20px;
  display: flex;
  justify-content: space-around;

  .pagelink {
    display: inline-flex;
    justify-content: center;
    min-width: 120px;
```

```
    padding: 4px 12px;
    border: 1px solid #333;
    border-radius: 4px;
    text-decoration: none;
    font-weight: bold;
    color: #666;
  }
}
</style>
```

ページを確認した際に以下のような感じになればOKです。



4.14.12　初期化処理の追加

「npm run init」をしたときにログも削除するようにしておきましょう。「backend/server/initialize.js」を開き、Redisに保存されたセッション情報の削除の項目の後に以下の内容を追記して保存してください。これで初期化時にログも一緒に削除されるようになります。これで探索戦の実装は完了です。

frontend/components/LogList.vue

```
// 保存された探索ログをすべて削除
const exploreLogDirectories = await fs.readdir(config.logDirectory);
for (let i = 0; i < exploreLogDirectories.length; i++) {
  await fs.rmdir(`${config.logDirectory}/${exploreLogDirectories[i]}`, {recursive: true});
}
```

4.15 物語戦（定期更新）

4.15.1 テンプレートとステージの作成

次に物語戦を実装していきましょう。まずは更新結果ページのテンプレートを作成します。「backend/template」内に「story.ejs」を作成し、以下の内容を入力して保存してください。日記を表示する部分の追加があるだけで探索戦のテンプレートとほぼ変わりありません。日記を表示する関係上、装飾に関連したCSSの追加も必要なのでその点に注意してください。

backend/template/story.ejs

```
<!DOCTYPE html>
<html lang="ja">
  <head>
    <meta charset="UTF-8">
    <title><%- title %></title>
    <style>
      html {
        overflow-y: scroll;
      }

      body {
        background: #F8F8F8;
        color: #333;
        font-size: 16px;
        font-family: 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ', Meiryo, 'ＭＳ Ｐ
ゴシック', sans-serif;
      }

      .diary {
        margin: 0 auto;
        width: 1000px;
      }

      .diary-heading {
        box-sizing: border-box;
        padding: 20px 20px;
        margin-bottom: 20px;
        width: 100%;
        font-size: 36px;
        letter-spacing: 2px;
        border-bottom: 1px solid lightgray;
        color: #444;
        font-family: 'BIZ UDPGothic', 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ
', Meiryo, 'ＭＳ Ｐゴシック', sans-serif;
      }

      .small {
        font-size: 66%;
      }

      .large {
        font-size: 150%;
      }
```

```
      .bold {
        font-weight: bold;
      }

      .diary-article {
        padding: 20px;
      }

      .stage {
        margin: 0 auto;
        width: 1000px;
      }

      .title {
        box-sizing: border-box;
        padding: 20px 20px;
        margin-bottom: 20px;
        width: 100%;
        font-size: 36px;
        letter-spacing: 2px;
        border-bottom: 1px solid lightgray;
        color: #444;
        font-family: 'BIZ UDPGothic', 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ
', Meiryo, 'ＭＳ Ｐゴシック', sans-serif;
      }

      .description {
        padding: 20px;
      }

      .battle {
        margin: 0 auto;
        width: 1000px;
      }

      .battle-start {
        width: 100%;
      }

      .battle-start-call {
        box-sizing: border-box;
        padding: 20px 20px;
        margin-bottom: 20px;
        width: 100%;
        font-size: 36px;
        letter-spacing: 2px;
        border-bottom: 1px solid lightgray;
        color: #444;
        font-family: 'BIZ UDPGothic', 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ
', Meiryo, 'ＭＳ Ｐゴシック', sans-serif;
      }

      .battle-start-call::first-letter {
        font-size: 44px;
      }
```

```
      .round {
        padding: 10px;
      }

      .round-start {
        box-sizing: border-box;
        padding: 20px 20px;
        width: 100%;
        border-bottom: 1px solid lightgray;
        font-size: 24px;
        color: #666;
        font-family: 'BIZ UDPGothic', 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ
', Meiryo, 'ＭＳ Ｐゴシック', sans-serif;
      }

      .round-count {
        font-size: 48px;
        color: #444;
      }

      .skill {
        margin-left: 10px;
      }

      .message {
        font-size: 12px;
      }

      .statuses {
        display: flex;
        justify-content: space-around;
        margin: 20px 0;
      }

      .team {
        width: 45%;
      }

      .unit {
        margin: 16px 0;
      }

      .unit-name {
        font-weight: bold;
        font-size: 24px;
        color: #666;
      }

      .statusbars {
        margin: 0 10px;
        display: flex;
      }

      .statusbar {
        width: 150px;
        margin: 0 10px;
      }
```

```
.statusbar-desc {
  margin: 0 10px;
  display: flex;
  justify-content: space-between;
}

.gauge-wrapper {
  position: relative;
  width: 100%;
  height: 10px;
  background: #DDD;
  transform: skewX(-30deg);
}

.gauge {
  position: absolute;
  height: 10px;
  background: #777;
}

.turns {
  margin: 20px 0px 0px 20px;
}

.turn {
  padding: 10px;
}

.actor {
  font-weight: bold;
  font-size: 20px;
  color: #444;
}

.action {
  margin: 10px 0 10px 16px;
}

.action-twice {
  font-weight: bold;
  font-size: 16px;
  color: #444;
}

.skill-name {
  font-weight: bold;
  font-size: 22px;
  color: #444;
}

.damage {
  font-weight: bold;
}

.damage-gte50 {
  font-size: 150%;
```

```
    }

    .damage-gte100 {
      font-size: 200%;
    }

    .heal {
      font-weight: bold;
    }

    .clean-up {
      margin-left: 30px;
    }

    .battle-result {
      box-sizing: border-box;
      width: 100%;
      padding: 20px;
      border-top: 1px solid lightgray;
      font-weight: bold;
      font-size: 30px;
    }
  </style>
</head>
<body>
  <section class="diary">
    <div class="diary-heading">日記</div>
    <div class="diary-article">
      <%- diary %>
    </div>
  </section>
  <section class="stage">
    <div class="title"><%- title %></div>
    <section class="description"><%- description %></section>
  </section>
  <%- log %>
</body>
</html>
```

次にステージを作ります。こちらも探索戦のものをほとんど流用することができます。「backend/server」内に「story.js」を作成し、以下の内容を記述して保存してください。

backend/server/story.js

```
const fs       = require('fs');
const ejs      = require('ejs');
const styling  = require('./styling.js');
const battle   = require('./battle.js');
const enemy    = require('./enemy.js');
const template = fs.readFileSync('./template/story.ejs', 'utf8');

const stage = [
  /*
    配列のインデックスがそのステージのIDになる。
    {
```

```
      title:       タイトル
      description: そのステージの序文
      enemy:       敵IDの配列
      condition:   他ステージのクリア状況を受け取り、挑戦可能かどうかをtrue/falseで返す
    }
  */
  /*  0 */ {
    title:       '街近郊にて',
    description: 'スライムたちが街に攻めてこようとしている。討伐しよう。',
    enemy:       ["slime", "slime", "slime", "slime", "slime"],
    condition: (clearFlags) => {
      return true;
    }
  },
  /*  1 */ {
    title:       '魔物の巣攻略戦',
    description: '魔物の巣を突き止めた。討伐隊を組み魔物を殲滅して街に平和を取り戻そう。<br>ただ、本拠地に
してはやけに魔物の数が少ないような気がする。',
    enemy:       ["slime", "slime", "slime", "goblin", "goblin"],
    condition: (clearFlags) => {
      return clearFlags[0]; // 街近郊にて制覇で挑戦可能
    }
  },
  /*  2 */ {
    title:       '燃える街',
    description: '魔物の巣から帰ると街が魔物に急襲され燃えていた。急ぎ魔物を討伐しよう。',
    enemy:       ["scorpion", "scorpion", "goblin", "goblin", "goblin"],
    condition: (clearFlags) => {
      return clearFlags[1]; // 魔物の巣攻略戦制覇で挑戦可能
    }
  }
];

const getClearFlags = (clearStatus) => {
  // それぞれのステージを制覇しているかという配列を返す関数
  // trueなら制覇済み、falseなら未制覇

  const clearFlags = [];
  for (let i = 0; i < stage.length; i++) {
    if (!clearStatus[i]) {
      clearFlags.push(false);
    } else {
      clearFlags.push(true);
    }
  }

  return clearFlags;
};

const getAvailableStageIds = (clearStatus) => {
  // 挑戦可能なステージID配列を返す関数

  const clearFlags = getClearFlags(clearStatus);

  const availableStageIds = [];
  for (let i = 0; i < stage.length; i++) {
    if (stage[i].condition(clearFlags)) {
```

```
      availableStageIds.push(i);
    }
  }

  return availableStageIds;
};

const getStageDescriptions = (clearStatus, stageIds) => {
  // クリア状況とステージID配列からステージ説明オブジェクトの配列を返す関数
  // 主にクライアント参照用なのでクライアントから参照してほしくない必要ないデータは返さない
  // （敵の内容など）

  const stageDescriptions = [];
  for (let i = 0; i < stageIds.length; i++) {
    const targetStage = stage[stageIds[i]];

    stageDescriptions.push({
      id:          stageIds[i],                      // ID
      title:       targetStage.title,                // タイトル
      isCompleted: targetStage.required <= clearStatus[stageIds[i]]
      // 制覇したかどうか
    })
  }

  return stageDescriptions;
};

const story = (stageId, allies, diary) => {
  // 物語戦を行う関数

  // 戦闘ログを生成
  // ただしstageIdがnullのとき（行き先が指定されていないとき）は戦闘を行わない
  let result = null;
  if (stageId != null) {
    result = battle.fight(
      allies,
      enemy.generateEnemyObjectArray(stage[stageId].enemy)
    );
  }

  // EJSで戦闘ログをレンダリング
  const renderedLog = ejs.render(template, {
    diary:       styling.profile(diary),
    title:       stageId != null ? stage[stageId].title :       '----',
    description: stageId != null ? stage[stageId].description : '行き先が指定されていないため戦闘は行
われませんでした。',
    log:         stageId != null ? result.log :                 ''
  });

  // 戦闘結果とレンダリングされた戦闘ログを返す
  return {
    result: result ? result.result : null,
    log:    renderedLog
  };
};

module.exports = {
```

```
  getAvailableStageIds: getAvailableStageIds,
  getStageDescriptions: getStageDescriptions,
  story: story
};
```

ほぼ変わりはありませんが、探索戦とは違って物語戦の行き先が指定されておらずstory関数で受け取るstageIdの値がnullになることがあります。そのため、その場合は戦闘を行わず行き先が指定されていない等のメッセージを表示しています。

4.15.2 行動宣言APIの作成

次は行動内容を宣言するAPIを作っていきます。宣言の内容は更新時に表示される日記、更新時に物語戦で挑むステージ、物語戦のパーティーです。物語戦のパーティーについてはお気に入りしているキャタクターのみ連れ出せることとします。

まずは宣言に必要な情報を返すAPIを作ります。「/characters/main/declare」にGETでアクセスすることで取得できることとしましょう。最初に「story.js」をAPIから呼び出せるようにします。「backend/server/api.js」を開き、冒頭部分を以下のように書き換えてください。

backend/server/api.js

```
const express   = require('express');
const config    = require('config');
const passport  = require('passport');
const fs        = require('fs').promises;
const styling   = require('./styling.js');
const skill     = require('./skill.js');
const explore   = require('./explore.js');
const story     = require('./story.js');
const Character = require('./model.js').Character;
const Room      = require('./model.js').Room;
const Message   = require('./model.js').Message;
const Log       = require('./model.js').Log;
const router    = express.Router();
```

書き換え終わったら以下のAPIを追記してください。

backend/server/api.js

```
router.get('/characters/main/declare', checkAuthentication, async (req, res) => {
  try {
    // キャラクター情報を取得
    const leader = await Character.findById(req.user._id, {
      eno:      1,
      nickname: 1,
      mainicon: 1,
      status:   1,
      summary:  1,
      fav:      1,
      story:    1,
      "declare.diary":   1,
      "declare.storyId": 1,
```

```
    "declare.party":   1,
  }).populate({
    path: 'fav',
    model: 'Character',
    select: {
      _id: 0,
      eno:      1,
      nickname: 1,
      mainicon: 1,
      status:   1,
      summary:  1,
    }
  }).populate({
    path: 'declare.party',
    model: 'Character',
    select: {
      _id: 0,
      deleted:  1,
      eno:      1,
      nickname: 1,
      mainicon: 1,
      status:   1,
      summary:  1,
    }
  });

  // 挑戦可能なステージ情報を受け取る
  const availableStageIds = story.getAvailableStageIds(leader.story);
  const availableStages   = story.getStageDescriptions(leader.story, availableStageIds);

  // 宣言時と状況が変わって連れ出そうとしているキャラクターが削除やブロックなどで
  // 連れ出しできなくなっている場合があるので検証を行う
  const party    = [];
  const vacancy  = false;
  const favEnoes = leader.fav.map(character => character.eno);

  // 何らかの理由で連れ出せなくなったキャラクターがいる場合vacancyをtrueに
  for (let i = 0; i < leader.declare.party.length; i++) {
    if (favEnoes.includes(leader.declare.party[i].eno)) {
      party.push({
        eno:      leader.declare.party[i].eno,
        nickname: leader.declare.party[i].nickname,
        mainicon: leader.declare.party[i].mainicon,
        status:   leader.declare.party[i].status,
        summary:  leader.declare.party[i].summary,
      });
    } else {
      vacancy = true;
    }
  }

  // 欠員状況、宣言情報等を返す
  return res.status(200).send({
    vacancy:    vacancy,
    leader: {
      eno:      leader.eno,
      nickname: leader.nickname,
```

```
        mainicon: leader.mainicon,
        status:   leader.status,
        summary:  leader.summary
      },
      declare: {
        diary:    leader.declare.diary,
        storyId:  leader.declare.storyId,
        party:    party
      },
      stages:     availableStages,
      characters: leader.fav
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

全体としては宣言内容とお気に入りしているキャタクター、挑戦できるステージ情報を返すAPIになっており、探索戦のステージ選択APIとさほど変わりません。ただし、探索戦とは違って物語戦は行動内容の指定から行動実行までに時間が開いてしまうためその間にブロック、お気に入り解除、キャラクター削除などの理由によりパーティーメンバーに欠員が生じてしまう場合があります。そのため、欠員が生じていないかの情報も返すようにし、もし欠員があった場合はユーザーに欠員が生じていることを伝えられるようにします。

では次は行動宣言を受け取るAPIを作りましょう。「/characters/main/declare」にPUTアクセスをすることで行動宣言を行えることとします。「backend/server/api.js」に以下のAPIを追記して保存してください。

backend/server/api.js

```
router.put('/characters/main/declare', checkAuthentication, checkCsrf, async (req, res) => {
  try {
    if (
      (typeof req.body.stage != 'number' && req.body.stage != null) ||
      !Array.isArray(req.body.party)
    ) {
      return res.status(400).send(); // 要求の内容の型がおかしければ400を返し中断
    }

    // 同じキャラクターを重複して連れ出そうとしていないかをチェック
    // ついでに型がおかしくないかをチェック
    const enoesChecker = {};
    for (let i = 0; i < req.body.party.length; i++) {
      if (typeof req.body.party[i] != 'number') {
        // partyに入っているENoがNumber型でなければ400を返して中断
        return res.status(400).send();
      }

      if (!enoesChecker[req.body.party[i]]) {
        enoesChecker[req.body.party[i]] = true;
      } else if (enoesChecker[req.body.party[i]]) {
        // PTが重複していたとき400を返して中断
        return res.status(400).send();
      }
    }
```

```
    const leader = await Character.findById(req.user._id, {
      fav:   1,
      story: 1
    }).populate({
      path: 'fav',
      model: 'Character',
      select: {
        eno: 1
      }
    });

    // 行けないはずのステージに行こうとしていた場合は400を返し中断
    const availableStageIds = story.getAvailableStageIds(leader.story);
    if (req.body.stage != null && !availableStageIds.includes(req.body.stage)) {
      return res.status(400).send();
    }

    // お気に入りしているキャラクターのENoの配列を生成
    const favEnoes = leader.fav.map(character => character.eno);

    // お気に入りしていないキャラクターを連れ出そうとしていたら400を返して中断
    for (let i = 0; i < req.body.party.length; i++) {
      if (!favEnoes.includes(req.body.party[i])) {
        return res.status(400).send();
      }
    }

    // 連れ出すキャラクターのENo配列をドキュメントIDの配列に
    const partyIds = [];
    for (let i = 0; i < req.body.party.length; i++) {
      for (let j = 0; j < leader.fav.length; j++) {
        if (req.body.party[i] == leader.fav[j].eno) {
          partyIds.push(leader.fav[j]._id);
        }
      }
    }

    // 問題がなければ宣言情報を更新
    await Character.findByIdAndUpdate(req.user._id, {
      "declare.diary":   req.body.diary,
      "declare.storyId": req.body.stage,
      "declare.party":   partyIds
    });

    return res.status(200).send();
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

こちらに関しては探索戦APIのデータ検証部分とほぼ同じです。宣言内容におかしなところがないかどうか検証し、問題がなければ宣言情報を更新しています。ここでは実際に戦闘を行ったりするわけではないので、この

APIでの処理はデータを保存するだけで終わりです。

4.15.3 行動宣言ページの作成

APIができたので行動宣言ページを作ります。「frontend/pages」内に「declare.vue」を作成し、以下の内容を入力して保存してください。

frontend/pages/declare.vue

```
<template>
  <section>
    <message-banner v-if="declare.vacancy || !declare.diary || !declare.party.length || stage == n
ull" type="warning">
      <div v-if="declare.vacancy">
        連れ出しできなくなったキャラクターがいます
      </div>
      <div v-if="!declare.diary">
        日記が入力されていません
      </div>
      <div v-if="!declare.party.length">
        物語戦のパーティーメンバーが指定されていません
      </div>
      <div v-if="stage == null">
        物語戦の行き先が指定されていません
      </div>
    </message-banner>
    <sub-heading>日記</sub-heading>
    <section class="form">
      <textarea class="diary" v-model="declare.diary"></textarea>
    </section>
    <sub-heading>ステージ選択</sub-heading>
    <section class="form">
      <section class="form-description">
        物語戦の行き先を選択します。<br>
        ステージをクリアすると新たなステージが解放されます。
      </section>
      <div class="stage-selector">
        <select class="stage-select" v-model="stage">
          <option :value="null">-- ステージを選択 --</option>
          <option v-for="stageOption in stages" :key="stageOption.id" :value="stageOption">
            <option>{{ stageOption.id }}. {{ stageOption.title }}{{ stageOption.isCompleted ? ' (
済) ' : '' }}</option>
          </option>
        </select>
      </div>
    </section>
    <sub-heading>パーティーメンバー選択</sub-heading>
    <section class="form">
      <div class="form-description">
        連れ出すパーティーメンバーを選択します。<br>
        お気に入りしているキャラクターの中から選択することができます。<br>
        パーティーメンバーはあなたを含めて最大{{ partyMembersMax }}人までです。
      </div>
      <div class="form-title">
        パーティーメンバー（クリックで選択キャンセル）
      </div>
      <selectable-character-list
```

```
        :characters="[leader].concat(declare.party)"
        @select="unselect"/>
      <div class="form-title">
        連れ出せるキャラクター
      </div>
      <selectable-character-list
        v-if="characters.length"
        :characters="characters"
        @select="select"/>
      <div v-else class="form-description">
        お気に入りしているキャラクターがいません。<br>
        お気に入りは<nuxt-link to="/list">キャラクター一覧</nuxt-link>からキャラクターページにアクセスし
<br>
        「お気に入りする」ボタンをクリックすることで行なえます。
      </div>
    </section>
    <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
    <div class="button-wrapper">
      <button class="button" @click="update">送信</button>
    </div>
  </section>
</template>

<script>
import SubHeading              from '~/components/SubHeading.vue'
import MessageBanner           from '~/components/MessageBanner.vue'
import SelectableCharacterList from '~/components/SelectableCharacterList.vue'

export default {
  components: {
    SubHeading,
    MessageBanner,
    SelectableCharacterList
  },
  middleware: 'authenticated',
  head() {
    return {
      title: '行動宣言'
    };
  },
  data() {
    return {
      partyMembersMax: process.env.partyMembersMax,

      errorMessage:    '',
      waitingResponse: false
    };
  },
  asyncData: async function(context) {
    const response = await context.$axios.get('/api/characters/main/declare');

    let stage = null;
    for (let i = 0; i < response.data.stages.length; i++) {
      if (response.data.stages[i].id == response.data.declare.storyId) {
        stage = response.data.stages[i];
      }
    }
```

```
  return {
    vacancy:    response.data.vacancy,
    leader: {
      eno:      response.data.leader.eno,
      nickname: response.data.leader.nickname,
      mainicon: response.data.leader.mainicon,
      status:   response.data.leader.status,
      summary:  response.data.leader.summary
    },
    declare: {
      diary:    response.data.declare.diary,
      storyId:  response.data.declare.storyId,
      party:    response.data.declare.party
    },
    stages:     response.data.stages,
    characters: response.data.characters,

    stage:      stage,
    party:      response.data.declare.party
  };
},
methods: {
  select: function(character) {
    if (this.partyMembersMax <= this.party.length) {
      // PT人数上限に届いていれば中断
      return;
    }

    for (let i = 0; i < this.party.length; i++) {
      if (character.eno == this.party[i].eno) {
        return; // すでに追加されているキャラクターであれば中断
      }
    }

    this.party.push(character);
  },
  unselect: function(character) {
    for (let i = 0; i < this.party.length; i++) {
      if (character.eno == this.party[i].eno) {
        return this.party.splice(i, 1);
        // パーティーメンバー配列から指定のキャラクターを取り除く
        // パーティーメンバー配列にリーダーは入っていないので
        // リーダーは取り除かれない
      }
    }
  },
  update: async function() {
    if (this.waitingResponse) {
      // 接続待ち状態であればアラートを表示しreturn、以降の処理を行わない
      return alert('しばらくお待ち下さい');
    }

    // 接続に入る、接続待ち状態をON(true)に
    this.waitingResponse = true;

    // 連れ出すキャラクターをENo配列に変換
```

```
      const partyEnoes = [];
      for (let i = 0; i < this.declare.party.length; i++) {
        partyEnoes.push(this.declare.party[i].eno);
      }

      try {
        await this.$axios.put('/api/characters/main/declare', {
          csrf:  this.$store.getters['auth/loginCharacter'].csrf,
          diary: this.declare.diary,
          stage: this.stage ? this.stage.id : null,
          party: partyEnoes
        });

        this.waitingResponse = false;
      } catch (e) {
        // エラーが発生した場合エラーメッセージを表示して接続待ち状態をOFF(false)に
        this.errorMessage = '実行中にエラーが発生しました';
        this.waitingResponse = false;
      }
    }
  }
}
</script>

<style lang="scss" scoped>
.diary {
  width: 100%;
  height: 300px;
  resize: none;
}

.stage-selector {
  padding: 20px 10px;
  display: flex;
  align-items: center;

  .stage-select {
    margin: 0;
  }

  .stage-requires {
    margin-left: 10px;
  }
}

.form-title {
  margin: 20px 0;
}
</style>
```

API同様、こちらも探索戦とあまり違いはありません。ただし物語戦特有の要素などにより多少違う部分があります。まず日記という要素が加わっているため、入力欄が追加されている他、PTメンバーに欠員が生じていた場合にその旨をエラーメッセージで表示しています。

また、探索戦ではステージは最初選ばれていない状態にしておけばよかったのですが物語戦では宣言してあっ

た場合にその宣言内容のステージを予め選択しておいたほうがユーザーフレンドリーなため、asyncData内で現在選んでいるステージを変数stageに入れてからreturnし表示するようにしています。このとき、変数stageの宣言はconstではなくletである必要があるため注意してください。（選び直すときにstageに再代入が入るため）

4.15.4 定期更新の仕様

　これから定期更新の更新処理部分の実装に入っていくのですが、その前に定期更新の仕様をもう少し細かく詰めていきましょう。定期更新ゲームにおいて更新は主に以下の3ステップで行われます。

更新

宣言内容に基づいて日記や戦闘などの結果ページを生成します。

再更新

スキルが不具合でうまく発動していないなど結果におかしな点があった場合、更新をやり直します。

更新結果の確定

特に結果に不具合などが見つからない・報告されない場合、結果を確定とし以降の再更新を行いません。

　定期更新だけのゲームであればこれを実現するのはそう難しくはないのですが、定期更新＋AP制のゲームにおいては再更新までの間にステータスやスキル構成などが変わってしまう可能性があるため少し一工夫をする必要があります。

　具体的にどうするのかというと、更新の宣言内容、ステータス、スキル構成を保存しておいて更新の際はそれを元に更新しステータスやスキル構成が変わっても支障が出ないようにします。また、物語戦の勝利報酬を更新時に配布してしまうと「もし勝利報酬を使ってしまって再更新時に敗北になってしまった場合どうすればいいのか？」という問題が生じてしまうので更新結果の確定時に勝利報酬は配布することとします。

4.15.5 宣言内容の保持

　細かい仕様を詰めたところで更新を担う部分を作っていきましょう。まずは宣言内容を保存できるようにしましょう。宣言内容についてはMongoDBにバックアップすることとします。「backend/server/model.js」を開き、スキーマ宣言部分の末尾に以下のコードを追記してください。

frontend/server/model.js

```
const DeclarationSchema = new Schema({
  character: {type: Schema.Types.ObjectId, ref: 'Character', required: true, index: true}, // 宣言を行ったキャラクター
  nth:       {type: Number, required: true, index: true}, // 第何更新目のログか
  diary:     {type: String, index: false},                // 日記
  storyId:   {type: Number, default: null},               // 物語戦行き先
  result:    {type: Number, index: false}, // 戦闘結果 null:戦闘を行っていない -1:敗北 0:引き分け 1:勝利
  party:     [{
    character: Schema.Types.ObjectId,
    nickname:  String,
    status: {
```

```
      atk: Number,
      dex: Number,
      mnd: Number,
      agi: Number,
      def: Number
    },
    skill: [Number]
  }]
});
```

Declarationは宣言内容を保存しておくためのコレクションです。日記の内容、物語戦の行き先、PTメンバーの更新時のステータスや名前、スキル構成を保存しておけるようになっています。また、確定時の報酬の付与のために勝敗も保存できるようになっている他、「いつの」「誰の」宣言内容であるかを区別できるように第何更新の誰の宣言であるかのデータも保持します。

スキーマを追加したらしたら忘れずに外部から参照できるようにしましょう。「backend/server/model.js」の最下部を以下のように書き換えて保存してください。追加された部分は斜体で表示しています。

frontend/server/model.js

```
const Character    = mongoose.model('Character',    CharacterSchema   );
const Room         = mongoose.model('Room',         RoomSchema        );
const Message      = mongoose.model('Message',      MessageSchema     );
const Log          = mongoose.model('Log',          LogSchema         );
const Declaration = mongoose.model('Declaration', DeclarationSchema);

  （省略）

module.exports = {
  Character:   Character,
  Room:        Room,
  Message:     Message,
  Log:         Log,
  Declaration: Declaration
};
```

次にこのDeclarationに宣言内容を保存する処理を書いていきます。「backend/server」の中に「update.js」を作成し、以下の内容を入力して保存してください。

frontend/server/update.js

```
const Character     = require('./model.js').Character;
const Declaration   = require('./model.js').Declaration;

const createDeclarationDocuments = async (nth) => {
  // 更新回数を受け取って宣言内容をデータベースにコピー

  // ENo降順で最後のキャラクターを取得
  const lastCharacter = await Character.find({
    deleted: false,
    eno:     {$gte: 1}
  }).sort({eno: -1}).select({_id: 0, eno: 1}).limit(1).lean().exec();
```

```
// そこから最大ENoを取得（キャラが1人も登録されていないときは0になる）
const maxEno = lastCharacter[0] ? lastCharacter[0].eno : 0;

for (let i = 1; i <= maxEno; i++) {
  const target = await Character.findOne({eno: i}).select({
    nickname: 1,
    status:   1,
    skill:    1,
    fav:      1,
    deleted:  1,
    "declare.diary":   1,
    "declare.storyId": 1,
    "declare.party":   1,
  }).populate({
    path: 'declare.party',
    model: 'Character',
    select: {
      nickname: 1,
      status:   1,
      skill:    1
    }
  }).exec();

  // 対象のキャラクターが削除されていなければ宣言内容を登録
  if (!target.deleted) {
    // パーティーの構築
    const party = [];

    // リーダーをパーティーに追加
    party.push({
      _id:      target._id,
      nickname: target.nickname,
      status:   target.status,
      skill:    target.skill
    });

    // パーティーメンバーの構築と検証
    // 何らかの理由でお気に入りから外れているキャラクターはPTの対象外に
    for (let i = 0; i < target.declare.party.length; i++) {
      if (target.fav.includes(target.declare.party[i]._id)) {
        party.push({
          character: target.declare.party[i]._id,
          nickname:  target.declare.party[i].nickname,
          status:    target.declare.party[i].status,
          skill:     target.declare.party[i].skill
        });
      }
    }

    // 宣言内容を保存
    const declaration = new Declaration({
      character: target._id,
      nth:       nth,
      diary:     target.declare.diary,
      storyId:   target.declare.storyId,
      result:    null,
```

```
        party:     party
      });

      await declaration.save();

      // キャラクターの宣言内容を初期化
      await target.update({
        declare: {
          diary:   null,
          storyId: null,
          party:   []
        }
      });
    }
  }
};
```

createDeclarationDocumentsは更新回数を受け取ってDeclarationのドキュメントを作って保存していく関数になります。まずはENoが最大でいくつあるのか取得し、そのデータをもとにそれぞれのキャラクターの宣言内容などを取得します。このとき、対象のキャラクターが削除されている場合は更新が行われないようにDeclarationドキュメントを作成しないようにします。

あとはお気に入りしていないキャラを連れ出そうとしていないかチェックし、宣言内容のコピーを取ったらそのキャラクターの宣言内容を初期化します。宣言内容の検証などは入力時に行っているためこちらでやることは比較的シンプルです。

4.15.6 更新

それでは次に更新処理を作っていきましょう。更新を行うときに重要なのは現在の更新状況のデータです。次が何更新目なのかわからないとDeclarationドキュメントを作れない他に、確定しているのか未確定なのかがわからないと未確定のはずなのに誤って更新してしまう可能性があり危険です。

そのため、まずは現在の更新状況のデータも持つようにしましょう。やり方はいくつか考えられるのですが、データベースのバックアップなどのときに情報が一緒にバックアップされて便利なのでここでは更新状況のデータもMongoDBに持たせることにします。「backend/server/model.js」を開き、スキーマ宣言部分の末尾に以下のコードを追記してください。

frontend/server/model.js

```
const UpdateStatusSchema = new Schema({
  nth:       {type: Number,  required: true}, // 今が第何更新目か
  finalized: {type: Boolean, required: true}  // 更新結果が確定されているか
});
```

UpdateStatusは今が第何更新目か、更新結果が確定されているかという情報だけを持つ比較的シンプルなコレクションです。このコレクションにドキュメントを1つだけ入れてあとはそのドキュメントを更新したり読んだりすることでデータを管理していきます。スキーマを追加したら忘れずにモデル化と外部参照の有効化を行いましょう。最後尾部分のコードを以下のように書き換えてください。追加された部分は斜体で表示しています。

```
frontend/server/model.js
const Character    = mongoose.model('Character',    CharacterSchema   );
const Room         = mongoose.model('Room',         RoomSchema        );
const Message      = mongoose.model('Message',      MessageSchema     );
const Log          = mongoose.model('Log',          LogSchema         );
const Declaration  = mongoose.model('Declaration',  DeclarationSchema );
const UpdateStatus = mongoose.model('UpdateStatus', UpdateStatusSchema);

  （省略）

module.exports = {
  Character:    Character,
  Room:         Room,
  Message:      Message,
  Log:          Log,
  Declaration:  Declaration,
  UpdateStatus: UpdateStatus
};
```

あとはこのUpdateStatusコレクションにドキュメントを作る必要があります。これは初期化時に自動で作成されるようにしましょう。ついでにLogやDeclarationの初期化処理も追加します。「backend/server/initialize.js」を開いてください。開いたらまずは冒頭の読み込み部分を以下のように書き換えます。追記されている部分は斜体で表示しています。

```
frontend/server/initialize.js
const fs       = require('fs').promises;
const config   = require('config');
const jsSHA    = require('jssha');
const redis    = require('redis').createClient();
const readline = require('readline');
const model    = require('./model.js');

const Character    = model.Character;
const Room         = model.Room;
const Message      = model.Message;
const Log          = model.Log;
const Declaration  = model.Declaration;
const UpdateStatus = model.UpdateStatus;
```

次に「// キャラクター、トークルーム、メッセージ情報を全て削除」と書かれているところを以下のように書き換えてください。LogとDeclarationとUpdateStatusも削除されるようにします。

```
frontend/server/initialize.js
  // DBのデータを全て削除
  await Character.deleteMany({});
  await Room.deleteMany({});
  await Message.deleteMany({});
  await Log.deleteMany({});
  await Declaration.deleteMany({});
```

```
await UpdateStatus.deleteMany({});
```

最後に「console.log('初期化処理が完了しました。');」となっているところの上に以下のコードを追記し、保存しましょう。ここでは現在の更新回数0、更新確定済としてデータを作成します。これは初回の更新処理を挟むときにこの値にしておくと特別な例外処理などをしなくてよくなり都合がいいためです。

frontend/server/initialize.js

```
// アップデート状況を初期化（更新回数0、更新確定済）
const updateStatus = new UpdateStatus({
  nth:       0,
  finalized: true
});
await updateStatus.save();
```

保存したらバックエンド側の実行を停止し「npm run init」を実行しましょう。データが全て初期化されてUpdateStatusのドキュメントが作成されます。

次に更新に必要な設定を記述します。「backend/config/default.json」を開き、以下の設定を追記して保存してください。更新結果の保存先を指定しています。

backend/config/default.json

```
"resultDirectory": "/var/www/teikiadv/result"
```

更新結果の保存先ディレクトリも作っておきましょう。バックエンド側コンソールから以下の2つのコマンドを順番に実行してください。

「cd /var/www/teikiadv」
「mkdir result」

あとは「cd /home/teiki/nodejs/teikiadv/backend」と「npm run dev」を順番に実行してバックエンドを再起動し、デバッグのために適当にキャラクターを登録しておきましょう。

これで準備は整ったので実際の更新処理を書いていきます。まずは必要なライブラリなどを読み込みましょう。「backend/server/update.js」を開き、冒頭部分を以下のように書き換えてください。

frontend/server/update.js

```
const fs           = require('fs').promises;
const config       = require('config');
const readline     = require('readline');
const story        = require('./story.js');
const Character    = require('./model.js').Character;
const Declaration  = require('./model.js').Declaration;
const UpdateStatus = require('./model.js').UpdateStatus;

// コンソールからの入力を受け付けるためのインターフェース
```

```
const readlineInterface = readline.createInterface({
  input:  process.stdin,
  output: process.stdout
});
```

書き換えたら「frontend/server/update.js」の末尾に以下のコードを追記しましょう。

frontend/server/update.js

```
const generateResults = async (nth) => {
  // 更新回数を受け取って宣言内容から結果を生成

  // ENo降順で最後のキャラクターを取得
  const lastCharacter = await Character.find({
    deleted: false,
    eno:     {$gte: 1}
  }).sort({eno: -1}).select({_id: 0, eno: 1}).limit(1).lean().exec();

  // そこから最大ENoを取得（キャラが1人も登録されていないときは0になる）
  const maxEno = lastCharacter[0] ? lastCharacter[0].eno : 0;

  for (let i = 1; i <= maxEno; i++) {
    const target = await Character.findOne({eno: i}).select({
      eno:     1,
      deleted: 1
    }).lean().exec();

    // 対象のキャラクターが削除されていなければ宣言内容から結果を生成
    if (!target.deleted) {
      // 該当の更新回数の宣言内容を取得
      const declaration = await Declaration.findOne({
        character: target._id,
        nth:       nth
      });

      // 宣言内容からパーティーを構築
      const allies = [];

      for (let i = 0; i < declaration.party.length; i++) {
        allies.push({
          name:     declaration.party[i].nickname,
          status:   declaration.party[i].status,
          skillIds: declaration.party[i].skill
        });
      }

      // 結果を生成
      const result = story.story(declaration.storyId, allies, declaration.diary);

      try {
        // キャラディレクトリを作る
        await fs.mkdir(`${config.resultDirectory}/${target.eno}`);
      } catch (e) {
        // すでにキャラディレクトリがある場合はエラーになる
        // これは想定内のエラーなのでそのまま進む
        // キャラディレクトリがすでに存在する以外の理由であればそのまま例外をスロー
```

```
        if (e.code != 'EEXIST') {
          throw e;
        }
      }

      // リザルトを更新回数.htmlとlatest.htmlに保存
      await fs.writeFile(`${config.resultDirectory}/${target.eno}/${nth}.html`, result.log);
      await fs.writeFile(`${config.resultDirectory}/${target.eno}/latest.html`, result.log);

      // 敗北であればresultIdを-1、引き分けなら0、勝利なら1に
      // （データベース保存用）
      let resultId = null;
      if (result.result == 'lose') {
        resultId = -1;
      } else if (result.result == 'even') {
        resultId = 0;
      } else if (result.result == 'win') {
        resultId = 1;
      }

      // 勝敗結果を更新
      await declaration.update({
        result: resultId
      });
    }
  }
};
```

作成したgenerateResults関数は更新回数を受け取ってDeclarationのデータを元に結果を生成して保存する関数です。まずはcreateDeclarationDocuments関数と同様に最大ENoを取得し、順次それぞれのキャラクターに処理を行っていきます。更新回数と対象のキャラクターのドキュメントIDをもとに対象のDeclarationドキュメントを取得し、そのデータをもとに結果を生成します。

生成したログは先程設定で指定したディレクトリ内の「/{対象のENo}/{更新回数}.html」と「/{対象のENo}/latest.html」に保存します。書き込む際、すでにファイルが存在している場合は上書きされるため再更新の際はこの関数を再び呼び出すだけでいいですし、latest.htmlは常に最新の更新で上書きされるためこれにアクセスすることで最新の結果を知ることができます。あとは結果を受け取ってDeclarationのresult（勝敗結果）を更新すれば処理は完了です。

次はこれらの関数を適切に呼び出し更新を行うupdate関数を実装しましょう。「frontend/server/update.js」の末尾に以下のコードを追記してください。

frontend/server/update.js

```
const update = async () => {
  // 前回の結果が確定されているか検証
  const updateStatus = await UpdateStatus.findOne({});
  if (!updateStatus.finalized) {
    return console.log('前回の更新が確定されていないため更新はできません。処理を終了します。'); // 確定されていなければ異常操作なので終了
  }

  // 各キャラの宣言内容を宣言コレクションに登録
```

```
  await createDeclarationDocuments(updateStatus.nth + 1);

  // 結果を生成して保存
  await generateResults(updateStatus.nth + 1);

  // 更新回数を+1し、確定状況を未確定（false）に
  await updateStatus.update({
    nth:       updateStatus.nth + 1,
    finalized: false
  });

  return console.log('処理が完了しました。');
};
```

update関数はまずUpdateStatusからデータを読み取ります。このとき誤操作防止のため更新が未確定であれば更新処理を行いません。問題がなければ更新を行っていきます。updateStatusから読み取れる更新回数は前回のもののため、それを+1した回数でcreateDeclarationDocuments関数とgenerateResults関数を呼び出します。最後にupdateStatusの更新回数を+1、確定状況を未確定にして保存すれば処理は完了です。

4.15.7 再更新

次は再更新について考えましょう。再更新については比較的簡単で、すでにできているDeclarationを元に再び結果を生成すればいいだけです。それでは実装しましょう。「frontend/server/update.js」の末尾に以下のコードを追記してください。

frontend/server/update.js

```
const reupdate = async () => {
  // 前回の結果が確定されているか検証
  const updateStatus = await UpdateStatus.findOne({});
  if (updateStatus.finalized) {
    return console.log('更新がすでに確定されているため再更新はできません。処理を終了します。'); // 確定されていれば異常操作なので終了
  }

  // 結果を生成して保存
  await generateResults(updateStatus.nth);

  return console.log('処理が完了しました。');
};
```

コードを見ても分かるように非常にシンプルで、更新処理からcreateDeclarationDocuments関数の呼び出しとupdateStatusの更新を抜いただけです。ただしこちらは更新時とは逆で更新が確定されている場合は再更新処理を行わないようにするため注意してください。

4.15.8 結果の確定と報酬の付与

次は確定処理の実装です。このときに物語戦をクリアしており、そのクリアが初回であれば報酬を付与するようにします。これは探索戦のクリアフラグ処理を流用すればほぼそのまま実現できそうです。それではこちらも実装し

ましょう。まずは初クリア時の報酬を設定に記述しておきます。「backend/config/default.json」を開き、以下の設定を追記して保存してください。

```
backend/config/default.json
  "storyCompleteReward": 10
```

　設定が書けたら「backend/server/update.js」の末尾に以下のコードを追記してください。

```
frontend/server/update.js
const finalize = async () => {
  // 前回の結果が確定されているか検証
  const updateStatus = await UpdateStatus.findOne({});
  if (updateStatus.finalized) {
    return console.log('更新はすでに確定されています。処理を終了します。'); // 確定されていれば異常操作なので終了
  }

  // 結果に基づき報酬の付与を行う
  // 現更新のうち、勝利しているものを取得
  const declarations = await Declaration.find({
    nth:    updateStatus.nth,
    result: 1
  }).populate({
    path: 'character',
    model: 'Character',
    select: {
      story: 1
    }
  }).lean().exec();

  for (let i = 0; i < declarations.length; i++) {
    const clearStatus = declarations[i].character.story;

    // 情報更新用のオブジェクトを作成
    const query = {};

    // 該当のステージが現状のクリア配列の外のIDの場合
    // クリア回数0で埋めつつ配列を伸ばしていく
    if (clearStatus.length < declarations[i].storyId) {
      for (let i = clearStatus.length; i <= declarations[i].storyId; i++) {
        clearStatus[i] = 0;
      }
    }

    clearStatus[declarations[i].storyId] = (clearStatus[declarations[i].storyId] || 0) + 1;
    query.story = clearStatus;

    // 今回が初クリアの場合はNP報酬を獲得
    if (clearStatus[declarations[i].storyId] == 1) {
      query.$inc = {
        np: config.storyCompleteReward
      }
    }
```

```
    // 情報を更新
    await Character.findByIdAndUpdate(declarations[i].character._id, query);
  }

  // 確定状況を確定（true）に
  await updateStatus.update({
    finalized: true
  });

  return console.log('処理が完了しました。');
};
```

まず最初にupdateStatusが未確定であるかをチェックし、問題なければDeclarationのうち報酬を付与する必要がある可能性のある勝利しているログのみを抽出しています。あとは順次初クリアの場合先程設定した分だけ報酬を付与し、クリア情報を更新して処理完了です。

4.15.9 各処理の呼び出し

最後にこれらの処理をコンソールからコマンドで呼び出せるようにしましょう。更新、再更新、結果の確定をそれぞれ「npm run update」「npm run reupdate」「npm run finalize」で呼び出せることとします。そのためのコードを書き加えましょう。「backend/server/update.js」の末尾に以下のコードを書き加えてください。

frontend/server/update.js

```
const updateCheck = async () => {
  if (process.argv.includes('update')) { // 新規更新する時
    readlineInterface.question('本当に更新を行いますか？(kousinと入力することで実行)', async (answer) => {
      if (answer == 'kousin') {
        await update();
      }
      process.exit(0);
    });
  } else if (process.argv.includes('reupdate')) { // 再更新する時
    readlineInterface.question('本当に再更新を行いますか？(saikousinと入力することで実行)', async (answer) => {
      if (answer == 'saikousin') {
        await reupdate();
      }
      process.exit(0);
    });
  } else if (process.argv.includes('finalize')) { // 結果を確定させるとき
    readlineInterface.question('本当に更新結果を確定しますか？(kakuteiと入力することで実行)', async (answer) => {
      if (answer == 'kakutei') {
        await finalize();
      }
      process.exit(0);
    });
  } else {
    console.log('コマンドの入力が正しくありません。プログラムを終了します。');
    process.exit(0);
  }
};

// 実行してすぐはいろいろ表示が出るので確認を1秒待つ
```

```
setTimeout(updateCheck, 1000);
```

updateCheck関数ではprocess.argvの情報を元にどの処理を行うか判定し、コマンド入力間違いでないか確認を行って実際に処理をしていきます。process.argvとは何かというと、プログラムを実行するときに渡される引数の配列で、例えば「node server/update.js finalize」を実行するとprocess.argvには「finalize」が含まれるようになります。

これをコマンドラインから呼び出せるようにします。「backend/package.json」を開き、"scripts"の項目を以下のように書き換えて保存してください。追記された部分は斜体で表示しています。

backend/package.json

```
"scripts": {
  "dev": "nodemon server/index.js --watch config --watch server",
  "start": "node server/index.js",
  "init": "node server/initialize.js",
  "ap": "node server/ap.js",
  "update": "node server/update.js update",
  "reupdate": "node server/update.js reupdate",
  "finalize": "node server/update.js finalize"
},
```

これで「npm run update」「npm run reupdate」「npm run finalize」からそれぞれの処理を呼び出せるようになりました。これで更新処理は完成です。

4.15.10 Nginxの設定変更

更新処理が書き上がったので物語戦の結果ログにアクセスできるようにNginxの設定変更を行っていきましょう。Tera Termでログインしrootユーザーに昇格してください。次に「vi /etc/nginx/conf.d/default.conf」を実行し、「location /ta/log/」セクションの次の場所に以下の内容を追記して上書き保存してください。

/etc/nginx/conf.d/default.conf

```
  location /ta/result/ {
      alias  /var/www/teikiadv/result/;
  }
```

保存したら「nginx -s reload」を実行し設定を反映させましょう。これで生成した結果ログが確認できるようになるはずなので、キャラクター登録と宣言を行ってから試しに「npm run update」を行ってみましょう。これで「http://dev.siroisakana.com/ta/result/(ENo)/(latest or 更新回数).html」というURLでログにアクセスできるようになります。アクセスして以下のような感じになれば成功です。

日記

日記入力テスト

街近郊にて

スライムたちが街に攻めてこようとしている。討伐しよう。

BATTLE START

テスト5の**シフトアップ！**
テスト5のAGIが増加した！

テスト1の**アルティメイタム！**
スライムAはテスト1へのヘイトが高まった！
スライムBはテスト1へのヘイトが高まった！
スライムCはテスト1へのヘイトが高まった！
スライムDはテスト1へのヘイトが高まった！
スライムEはテスト1へのヘイトが高まった！

勝利できるようなPTを組んで更新と結果の確定を行い、実際にステージクリア処理などがうまくいっているかなどもチェックしてください。

4.15.11 キャラクターページからの更新結果参照

結果が生成できてもリンクがないとアクセスすることが難しいので、キャラクターページから更新結果を参照できるようにしましょう。そのためにまずはキャラクターページAPIを変更します。さて、生成されている物語戦ログの情報はDeclarationから得ることができます（Declarationは更新と同時に生成されているためDeclarationがある回＝該当キャラクターの更新が行われた回）。なのでAPIからDeclarationを参照できるようにしておきましょう。「backend/server/api.js」を開き、冒頭部分を以下のように変更してください。追加された部分は斜体で表示しています。

backend/server/api.js

```
（省略）

const Message     = require('./model.js').Message;
const Log         = require('./model.js').Log;
const Declaration = require('./model.js').Declaration;
const router      = express.Router();
```

次にAPIを変更していきます。「router.get('/characters/:key(\\d+|main\)'」となっているAPIのtry部分を以下のように変更してください。resultsとして該当のキャラクターの更新が行われた回のデータを返却しています。

backend/server/api.js

```
try {
  const character = await Character.findOne(query, {
    eno:          1,
    name:         1,
    tags:         1,
    profile:      1,
```

```
      icons:         1,
      status:        1,
      skill:         1,
      profileImages: 1
    });

    // Declarationから更新が行われた回のデータを取得
    const declarations = await Declaration.find({character: character._id}).sort({nth: 1}).select(
{_id: 0, nth: 1}).exec();

    const results = [];
    for (let i = 0; i < declarations.length; i++) {
      results.push(declarations[i].nth);
    }

    if (!character) {
      return res.status(404).send(); // キャラクターの検索結果が存在しなければ404
    } else if (req.user && !(/main/.test(req.params.key))) {
      // キャラクターが見つかり、かつログインしていて自キャラ表示ではない場合
      // 接続者のお気に入り、ブロック、ミュート情報を取得
      const user = await Character.findById(req.user._id, {
        _id:   0,
        fav:   1,
        block: 1,
        mute:  1
      });

      // キャラクター情報とお気に入り、ブロック、ミュートしているかという情報を返す
      return res.status(200).send({
        eno:           character.eno,
        name:          character.name,
        tags:          character.tags,
        profile:       character.profile,
        icons:         character.icons,
        status:        character.status,
        skill:         skill.getDescriptions(character.skill),
        profileImage:  character.profileImages.length ? character.profileImages[Math.floor(Math.ra
ndom() * character.profileImages.length)] : null,
        isFaved:       user.fav.includes(character._id),
        isBlocked:     user.block.includes(character._id),
        isMuted:       user.mute.includes(character._id),
        results:       results
      });
    } else {
      // キャラクターは見つかったがログインはしていないとき
      // そのまま情報を返す
      return res.status(200).send({
        eno:           character.eno,
        name:          character.name,
        tags:          character.tags,
        profile:       character.profile,
        icons:         character.icons,
        status:        character.status,
        skill:         skill.getDescriptions(character.skill),
        profileImage:  character.profileImages.length ? character.profileImages[Math.floor(Math.ra
ndom() * character.profileImages.length)] : null,
        results:       results
```

```
      });
    }
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
```

ホームAPIも変更しておきましょう。「router.get('/characters/main/home'」となっているAPIを以下のように書き換えて保存してください。

backend/server/api.js

```
router.get('/characters/main/home', checkAuthentication, async (req, res) => {
  try {
    const character = await Character.findById(req.user._id, {
      eno:           1,
      name:          1,
      tags:          1,
      profile:       1,
      icons:         1,
      ap:            1,
      np:            1,
      status:        1,
      skill:         1,
      declare:       1,
      profileImages: 1
    });

    // Declarationから更新が行われた回のデータを取得
    const declarations = await Declaration.find({character: character._id}).sort({nth: 1}).select(
{_id: 0, nth: 1}).exec();

    const results = [];
    for (let i = 0; i < declarations.length; i++) {
      results.push(declarations[i].nth);
    }

    return res.status(200).send({
      eno:     character.eno,
      name:    character.name,
      tags:    character.tags,
      profile: character.profile,
      icons:   character.icons,
      ap:      character.ap,
      np:      character.np,
      status:  character.status,
      skill:   skill.getDescriptions(character.skill),
      isDeclared: { // 宣言をしているかどうか
        diary: character.declare.diary != null,
        story: character.declare.storyId != null,
        party: character.declare.party.length != 0
      },
      profileImages: character.profileImages.length ? character.profileImages[Math.floor(Math.rand
om() * character.profileImages.length)] : null,
      // プロフィール画像配列が存在するときランダムに1つを返す、なければnull
```

```
      results: results
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

APIを追加したら今度はフロントエンド側で表示できるようにしましょう。まずは更新結果URLを設定に記述しておきます。「frontend/nuxt.config.js」を開き、「env」の項目に以下の設定を追記して保存してください。ドメイン名だけでなくhttp/httpsの部分を環境によって書き換えるのを忘れないようにしましょう。

frontend/nuxt.config.js

```
    resultDirectory: 'https://dev.siroisakana.com/ta/result/'
```

次に「frontend/pages/profile/_key.vue」を開き、<template>部分とasyncData部分を以下のように書き換えて保存ください。APIからresultsを受け取るように変更します。

frontend/pages/profile/_key.vue

```
<template>
  <section>
    <character-profile
      mode="profile"
      :eno="eno"
      :name="name"
      :tags="tags"
      :profile="profile"
      :profileImage="profileImage"
      :icons="icons"
      :status="status"
      :skill="skill"
      :isFaved="isFaved"
      :isBlocked="isBlocked"
      :isMuted="isMuted"
      :results="results"
      @relation="relation"/>
  </section>
</template>

  （省略）

  asyncData: async function(context) {
    const response = await context.$axios.get(`/api/characters/${context.params.key}`);
    return {
      eno:          response.data.eno,
      name:         response.data.name,
      tags:         response.data.tags,
      profile:      response.data.profile,
      profileImage: response.data.profileImage,
      icons:        response.data.icons,
      status:       response.data.status,
```

```
      skill:        response.data.skill,
      isFaved:      response.data.isFaved,
      isBlocked:    response.data.isBlocked,
      isMuted:      response.data.isMuted,
      results:      response.data.results
    };
  },
```

「frontend/pages/home.vue」も同じように書き換えて保存しましょう。

frontend/pages/home.vue

```
<template>
  <section>
    <message-banner v-if="np || !isDeclared.diary || !isDeclared.story || !isDeclared.party" type="warning">
      <div v-if="np">
        NPを{{ np }}割り振ることができます
      </div>
      <div v-if="!isDeclared.diary">
        日記が入力されていません
      </div>
      <div v-if="!isDeclared.story">
        物語戦の行き先が指定されていません
      </div>
      <div v-if="!isDeclared.party">
        物語戦のパーティーメンバーが指定されていません
      </div>
    </message-banner>
    <character-profile
      mode="home"
      :eno="eno"
      :name="name"
      :tags="tags"
      :profile="profile"
      :profileImage="profileImage"
      :icons="icons"
      :ap="ap"
      :status="status"
      :skill="skill"
      :results="results" />
  </section>
</template>

  (省略)

  asyncData: async function(context) {
    const response = await context.$axios.get('/api/characters/main/home');
    return {
      eno:          response.data.eno,
      name:         response.data.name,
      tags:         response.data.tags,
      profile:      response.data.profile,
      profileImage: response.data.profileImage,
      icons:        response.data.icons,
      ap:           response.data.ap,
      np:           response.data.np,
```

```
      isDeclared:   response.data.isDeclared,
      status:       response.data.status,
      skill:        response.data.skill,
      results:      response.data.results
    };
  }
```

最後にコンポーネントを更新してresultsを表示できるようにします。「frontend/components/CharacterProfile.vue」を開き、以下の内容で上書きして保存してください。変更された箇所は斜体で表示しています。

frontend/components/CharacterProfile.vue

```
<template>
  <section>
    <h2 class="name">ENo.{{ eno }} {{ name }}</h2>
    <section
      class="relation"
      v-if="
        mode == 'profile' &&
        $store.getters['auth/isAuthenticated'] &&
        eno != $store.getters['auth/loginCharacter'].eno">
      <div
        class="relation-button"
        v-if="!isFaved && !isBlocked"
        @click="relation('fav')">
          お気に入りする
      </div>
      <div
        class="relation-button done"
        v-if="isFaved && !isBlocked"
        @click="relation('unfav')">
          お気に入り中
      </div>
      <div
        class="relation-button"
        v-if="!isBlocked"
        @click="relation('block')">
          ブロックする
      </div>
      <div
        class="relation-button done"
        v-if="isBlocked"
        @click="relation('unblock')">
          ブロック中
      </div>
      <div
        class="relation-button"
        v-if="!isMuted"
        @click="relation('mute')">
          ミュートする
      </div>
      <div
        class="relation-button done"
        v-if="isMuted"
        @click="relation('unmute')">
```

```
        ミュート中
      </div>
    </section>
    <section class="main">
      <section class="image-wrapper">
        <img v-if="profileImage" class="image" :src="profileImage">
      </section>
      <section class="summary">
        <div v-if="mode == 'home'" class="ap-wrapper">
          <div class="ap">AP</div>
          <div class="ap-value">{{ ap }}</div>
        </div>
        <table class="statuses">
          <tbody>
            <tr>
              <td class="status-name">ATK</td>
              <td class="status-value">{{ status.atk }}</td>
              <td class="status-name">DEX</td>
              <td class="status-value">{{ status.dex }}</td>
              <td class="status-name">MND</td>
              <td class="status-value">{{ status.mnd }}</td>
            </tr>
            <tr>
              <td class="status-name">AGI</td>
              <td class="status-value">{{ status.agi }}</td>
              <td class="status-name">DEF</td>
              <td class="status-value">{{ status.def }}</td>
            </tr>
          </tbody>
        </table>
        <div class="skills">
          <div class="skill" v-for="(skillDescription, index) in skill" :key="index">
            <div class="skill-prop">
              <div class="skill-name">{{ skillDescription ? skillDescription.name : '----'}}</div>
              <div class="skill-cond">{{ skillDescription ? skillDescription.cond : '----'}}</div>
            </div>
            <div class="skill-effect">{{ skillDescription ? skillDescription.desc : '----'}}</div>
          </div>
        </div>
      </section>
    </section>
    <section v-if="tags.length">
      <span class="tag" v-for="(tag, index) in tags" :key="index">{{ tag }}</span>
    </section>
    <section v-if="results && results.length">
      <sub-heading>更新結果</sub-heading>
      <div class="results">
        <a
          class="result latest"
          :href="`${resultDirectory}${eno}/latest.html`"
          target="_blank">
          最新
        </a>
        <a
          v-for="result in results"
          :key="result.index"
          class="result"
```

```
          :href="`${resultDirectory}${eno}/${result}.html`"
          target="_blank">
          第{{ result }}回
        </a>
      </div>
    </section>
    <section>
      <sub-heading>プロフィール</sub-heading>
      <div class="profile" v-html="profile"></div>
    </section>
    <section>
      <sub-heading>アイコン</sub-heading>
      <div class="character-icons-wrapper">
        <div class="character-icon-wrapper" v-for="i in iconsMaxLength" :key="i">
          <character-icon :src="icons[i - 1] ? icons[i - 1].url : ''"/>
        </div>
      </div>
    </section>
  </section>
</template>

<script>
import SubHeading    from '~/components/SubHeading.vue'
import CharacterIcon from '~/components/CharacterIcon.vue'

export default {
  components: {
    SubHeading,
    CharacterIcon
  },
  props: {
    mode:         String, // 表示モード home = ホーム、profile = キャラクターページ、preview = プレビュー
    eno:          Number,
    name:         String,
    tags:         Array,
    profile:      String,
    profileImage: String,
    icons:        Array,
    ap:           Number,
    status:       Object,
    skill:        Array,
    isFaved:      Boolean,
    isBlocked:    Boolean,
    isMuted:      Boolean,
    results:      Array
  },
  data() {
    return {
      resultDirectory: process.env.resultDirectory,
      iconsMaxLength:  process.env.iconsMaxLength
    };
  },
  methods: {
    relation: function(action) {
      this.$emit('relation', action);
    }
```

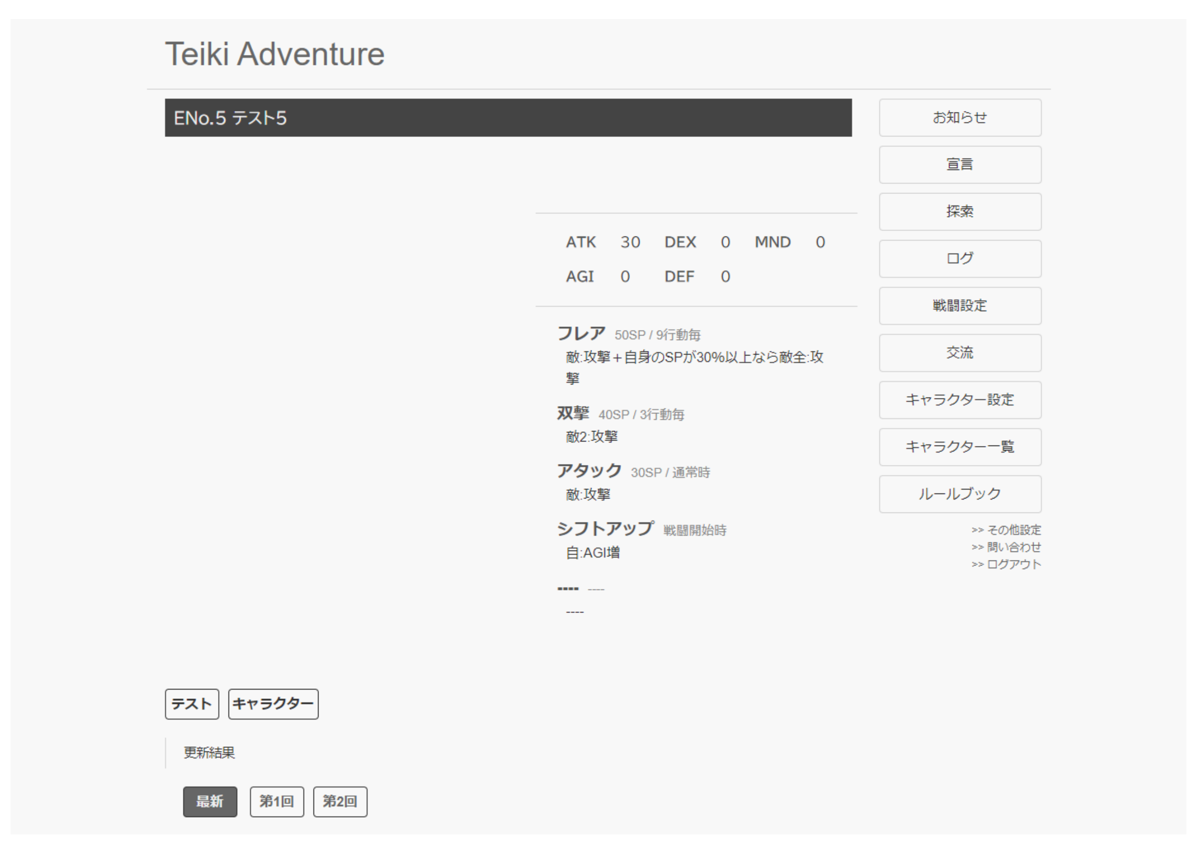
```
  }
}
</script>

<style lang="scss" scoped>
$font: 'BIZ UDPGothic', 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ', Meiryo, 'ＭＳ Ｐゴシック', sans-serif;

.name {
  display: block;
  background: #444;
  color: #EEE;
  margin: 10px 0;
  padding: 10px;
  font-size: 18px;
  font-family: $font;
  letter-spacing: 0;
}

.relation {
  display: flex;
  justify-content: flex-end;
  width: 100%;

  .relation-button {
    display: inline-flex;
    justify-content: center;
    padding: 4px 10px;
    border: 1px solid #666;
    border-radius: 4px;
    margin: 0 5px;
    text-decoration: none;
    font-weight: bold;
    color: #666;
    cursor: pointer;
  }

  .done {
    border: 1px solid #333;
    background: #666;
    color: #EEE;
  }
}

.main {
  display: flex;

  .image-wrapper {
    flex-grow: 0;
    flex-shrink: 0;
    width: 400px;
    height: 600px;

    .image {
      width: 100%;
      height: 100%;
    }
```

```
  }

  .summary {
    flex-grow: 1;
    padding: 10px;
    display: flex;
    align-items: center;
    justify-content: center;
    flex-direction: column;

    .ap-wrapper {
      width: 100%;
      display: flex;
      align-items: baseline;
      justify-content: center;
      font-family: $font;

      .ap {
        font-weight: bold;
        font-size: 24px;
        color: #AAA;
      }

      .ap-value {
        margin-left: 40px;
        font-weight: bold;
        font-size: 48px;
        color: #666;
      }
    }

    .statuses {
      box-sizing: border-box;
      margin: 0;
      padding: 12px 20px;
      width: 100%;
      border-top: 1px solid lightgray;
      border-bottom: 1px solid lightgray;
      font-family: $font;

      td {
        border: none;
        padding: 6px 12px;
      }

      .status-name {
        font-weight: bold;
        color: #555;
        margin-right: 20px;
      }
    }
  }
}

.skills {
  box-sizing: border-box;
  width: 100%;
```

```
  padding: 16px 24px 0 24px;

  .skill {
    margin-bottom: 12px;

    .skill-prop {
      display: flex;
      align-items: baseline;

      .skill-name {
        font-weight: bold;
        color: #555;
        font-size: 18px;
        margin-right: 10px;
      }

      .skill-cond {
        font-size: 14px;
        color: #888;
      }
    }

    .skill-effect {
      margin-left: 10px;
      color: #333;
    }

    &:last-child {
      margin: 0;
    }
  }
}

.tag {
  display: inline-flex;
  justify-content: center;
  min-width: 50px;
  padding: 4px;
  border: 1px solid #333;
  border-radius: 4px;
  margin-right: 10px;
  text-decoration: none;
  font-weight: bold;
  color: #333;
}

.results {
  margin: 0 20px;

  .result {
    display: inline-flex;
    justify-content: center;
    min-width: 50px;
    padding: 4px;
    border: 1px solid #333;
    border-radius: 4px;
    margin-right: 10px;
```

```
    text-decoration: none;
    font-weight: bold;
    color: #666;
  }

  .latest {
    background-color: #666;
    color: #EEE;
  }
}

.profile {
  margin: 0 20px;
}

.character-icons-wrapper {
  display: flex;
  flex-wrap: wrap;
  justify-content: center;

  .character-icon-wrapper {
    margin: 5px;
  }
}
</style>
```

ここまで実装したらキャラクターページを表示してみましょう。以下のような感じになっていればOKです。



更新結果リンクをクリックしてみてきちんとそれぞれの結果ページに飛ぶことも確認しておきましょう。

4.15.12 初期化設定の変更

最後に初期化時に定期更新のログも削除するようにしておきましょう。「backend/server/initialize.js」を開き、「// 保存された探索ログをすべて削除」の部分の次の箇所に以下のコードを追記して保存してください。

backend/server/initialize.js

```
// 保存された物語ログをすべて削除
const resultDirectories = await fs.readdir(config.resultDirectory);
for (let i = 0; i < resultDirectories.length; i++) {
  await fs.rmdir(`${config.resultDirectory}/${resultDirectories[i]}`, {recursive: true});
}
```

4.16 管理者用ページ

4.16.1 問い合わせページ

ゲームを運営していく上でユーザーからの問い合わせを受け付けられるフォームがあったほうが便利でいいでしょう。ここではMongoDBに問い合わせ内容を保管するようにします。まずは問い合わせ内容をを保持するためのコレクションを作成しましょう。必要なデータとしては問い合わせを受けた日時、問い合わせているユーザーの名前、返信先、問い合わせ内容となるでしょう。「backend/server/model.js」を開き、UpdateStatusSchemaの次の箇所に以下のコードを追記してください。

backend/server/model.js

```
const InquirySchema = new Schema({
  name:      {type: String,  required: true},              // 名前
  content:   {type: String,  required: true, index: false}, // 問い合わせ内容
  timestamp: {type: Date,    default: Date.now},            // 問い合わせ日時
  resolved:  {type: Boolean, default: false},               // 解決済みかどうか
  address:   String                                         // 返信先
});
```

nameが名前、contentが問い合わせ内容、addressが返信先です。またtimestampは問い合わせ日時、resolvedは解決済みかどうかを表すステータスで、それぞれ初期値は問い合わせが行われたときの日時、未解決です。モデル化して外部から参照できるようにする作業も忘れずに行っておきます。ここまで変更したら保存しておきましょう。

backend/server/model.js

```
const Character    = mongoose.model('Character',    CharacterSchema    );
const Room         = mongoose.model('Room',         RoomSchema         );
const Message      = mongoose.model('Message',      MessageSchema      );
const Log          = mongoose.model('Log',          LogSchema          );
const Declaration  = mongoose.model('Declaration',  DeclarationSchema  );
const UpdateStatus = mongoose.model('UpdateStatus', UpdateStatusSchema);
const Inquiry      = mongoose.model('Inquiry',      InquirySchema      );

  （省略）

module.exports = {
  Character:    Character,
  Room:         Room,
  Message:      Message,
  Log:          Log,
  Declaration:  Declaration,
  UpdateStatus: UpdateStatus,
  Inquiry:      Inquiry
};
```

モデルが作成できたら問い合わせを受け取るAPIを作成します。その前にAPI側でInquiryを参照できるように

しましょう。「backend/servre/api.js」の冒頭部分に以下の内容を追記してください。

backend/server/api.js

```
（省略）

const Message     = require('./model.js').Message;
const Log         = require('./model.js').Log;
const Declaration = require('./model.js').Declaration;
const Inquiry     = require('./model.js').Inquiry;
const router      = express.Router();
```

追記したら「/inquiry」にPOSTでアクセスすることで問い合わせを受け取れるようにしましょう。以下のAPIを適当な箇所に追記して保存してください。

backend/server/api.js

```
router.post('/inquiry', async (req, res) => {
  try {
    if (
      typeof req.body.name    != 'string' ||
      typeof req.body.content != 'string' ||
      (req.body.address && typeof req.body.address != 'string')
    ) {
      // 問い合わせの型がおかしければ400を返して中断
      return res.status(400).send();
    }

    // 問題なければ問い合わせ内容をDBに登録
    const inquiry = new Inquiry({
      name:    req.body.name,
      address: req.body.address,
      content: req.body.content
    });

    await inquiry.save();

    return res.status(200).send();
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

内容はシンプルで送られてきたデータの型がおかしくないかをチェックし問題なければデータベースに問い合わせの内容を保存するだけです。

APIができたらページを作成しましょう。「/ta/inquiry」でアクセスを受け取るようにするのですが、ここの中にさらに別のページも追加したいので「frontend/pages」内に「inquiry」ディレクトリを作成し、その中に「index.vue」を作成してください。ファイル構成は以下のようになります。

```
pages
  inquiry
    index.vue
```

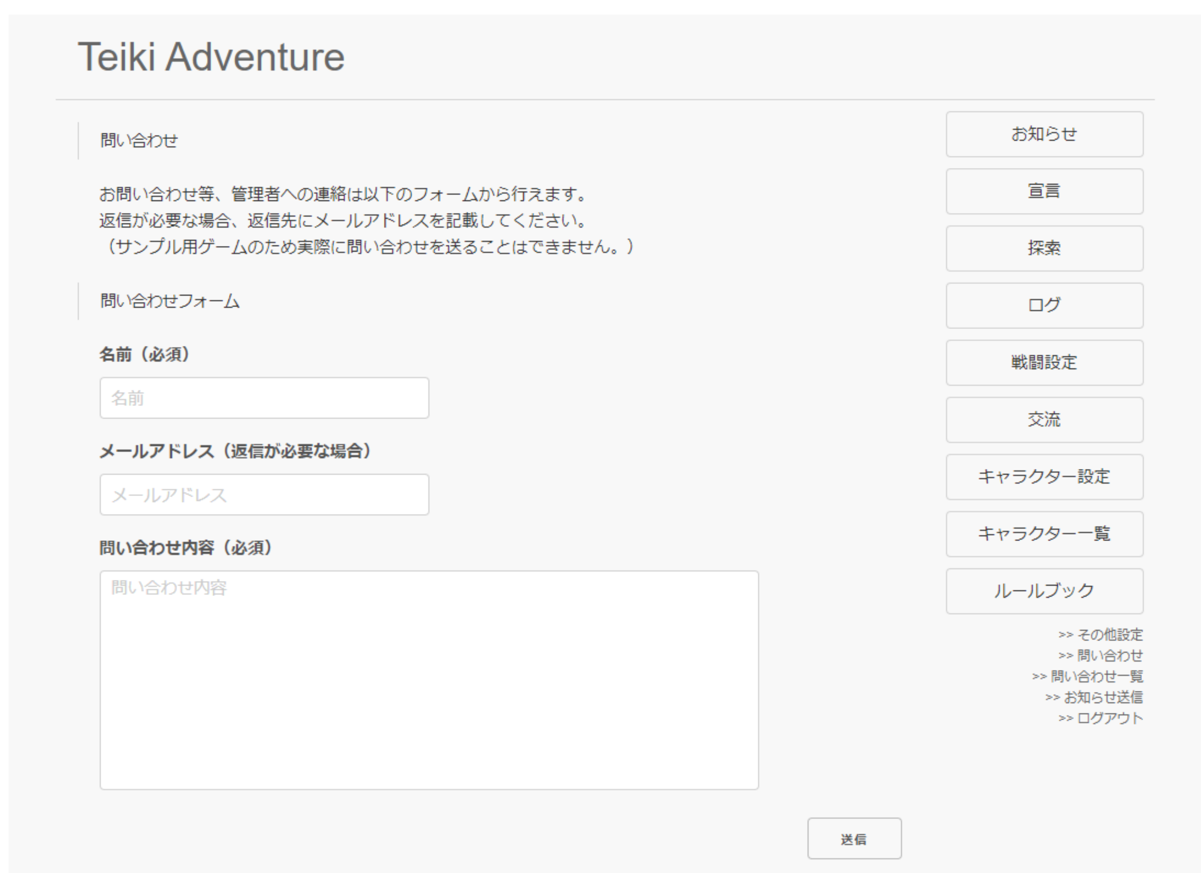
作成したら以下の内容を入力して保存してください。

frontend/pages/inquiry/index.vue

```
<template>
  <section>
    <section>
      <sub-heading>問い合わせ</sub-heading>
      <div class="inquiry-description">
        お問い合わせ等、管理者への連絡は以下のフォームから行えます。<br>
        返信が必要な場合、返信先にメールアドレスを記載してください。<br>
        （サンプル用ゲームのため実際に問い合わせを送ることはできません。）
      </div>
    </section>
    <section>
      <sub-heading>問い合わせフォーム</sub-heading>
      <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
      <section class="form">
        <div class="form-title">名前（必須）</div>
        <input class="form-input" type="text" v-model="name" placeholder="名前">
      </section>
      <section class="form">
        <div class="form-title">メールアドレス（返信が必要な場合）</div>
        <input class="form-input" type="text" v-model="address" placeholder="メールアドレス">
      </section>
      <section class="form">
        <div class="form-title">問い合わせ内容（必須）</div>
        <textarea class="form-textarea" v-model="content" placeholder="問い合わせ内容"></textarea>
      </section>
      <div class="button-wrapper">
        <button class="button" @click="send">送信</button>
      </div>
    </section>
  </section>
</template>

<script>
import SubHeading    from '~/components/SubHeading.vue'
import MessageBanner from '~/components/MessageBanner.vue'

export default {
  components: {
    SubHeading,
    MessageBanner
  },
  head() {
    return {
      title: '問い合わせ'
    };
  },
  data() {
```

```
    return {
      name:    '', // 名前
      address: '', // メールアドレス
      content: '', // 問い合わせ内容
      errorMessage:    '', // エラーメッセージ
      waitingResponse: false // 通信待ち中かどうか
    }
  },
  methods: {
    send: async function() {
      if (this.waitingResponse) {
        // 接続待ち状態であればアラートを表示しreturn、以降の処理を行わない
        return alert('しばらくお待ち下さい');
      }

      // 入力内容の検証を行う、問題があればエラーメッセージを表示、returnして処理を中断
      if (!this.name) { // フルネームが入力されていない
        return this.errorMessage = '名前が入力されていません';
      }
      if (!this.content) { // 短縮名が入力されていない
        return this.errorMessage = '問い合わせ内容が入力されていません'
      }

      // 問題がなければハッシュ化と接続に入る、接続待ち状態をON(true)に
      this.waitingResponse = true;

      try {
        // 問い合わせを送信
        const response = await this.$axios.post('/api/inquiry', {
          name:    this.name,
          address: this.address,
          content: this.content
        });

        // 送信できたらトップページへリダイレクト
        this.$router.push('/');
      } catch (e) {
        // エラーが発生した場合接続待ち状態をOFF(false)に
        this.waitingResponse = false;
        this.errorMessage = '送信中にエラーが発生しました'; // エラーメッセージを表示
      }
    }
  }
}
</script>

<style lang="scss" scoped>
.inquiry-description {
  margin: 20px;
}
</style>
```

こちらも入力内容を検証してAPIに送信するだけのシンプルな作りになっています。ページにアクセスして以下のような感じになっていればOKです。この先のテスト用に問い合わせを何件か送信しておきましょう。



4.16.2 管理者しかアクセスできないAPI

いちいちツールなどを起動しなくてもWeb上で問い合わせを表示できると便利なので今度は送られた問い合わせを表示できるようにしましょう。そのためには管理者のみアクセスできるAPIを用意する必要があるため、作っていきましょう。「backend/server/api.js」を開き、checkCsrfを定義しているところの次の箇所に以下のコードを追記してください。

backend/server/api.js

```
const checkAdministrator = (req, res, next) => {
  // 管理者かどうか確認するミドルウェア関数
  if (req.user.eno == 0) { // 管理者であれば次へ
    next();
  } else { // 管理者でなければ401を返して処理を中断する
    res.status(401).send();
  }
};
```

これは管理者のみアクセスできるAPIを実現するミドルウェア関数でcheckAuthenticationの後に呼び出して使います。アクセスしているユーザーのENoが0かどうかで管理者かどうかを判定しています。

4.16.3 問い合わせの表示

それではこれを利用して問い合わせ表示APIを作っていきましょう。まずは問い合わせを表示する際に1ページ何件表示するかを設定に記述しておきます。「backend/config/default.json」を開き、以下の設定を追記して保存してください。

backend/config/default.json

```
"inquiriesPerPage": 20
```

次に「backend/server/api.js」を開き、以下のAPIを追記してください。問い合わせの一覧を表示するAPIです。

backend/server/api.js

```
router.get('/inquiry', checkAuthentication, checkAdministrator, async (req, res) => {
  try {
    const query = {};
    if (req.query.unresolved) { // 未解決の問い合わせのみ表示するモードなら
      query.resolved = false;   // 「未解決の問い合わせ」を検索条件に追加
    }

    const currentPage = Number(req.query.page);
    let inquiries = await Inquiry.find(query, {
      name:      1,
      content:   1,
      address:   1,
      timestamp: 1,
      resolved:  1
    }).sort({timestamp: -1}).skip(currentPage * config.inquiriesPerPage).limit(config.inquiriesPer
Page + 1).exec();

    let isContinuing = false;
    if (config.inquiriesPerPage < inquiries.length) {
      isContinuing = true;
      inquiries = inquiries.slice(0, config.inquiriesPerPage);
    }

    return res.status(200).send({
      list:         inquiries,
      isContinuing: isContinuing
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

このAPIでは問い合わせをunresolvedというURLパラメーターの有無で問い合わせを全件表示するか未解決のもののみ表示するかを選ぶことができるようになっています。また、ページング機能がついています。仕組みとしてはcheckAdministratorによって管理者以外アクセスできなくなっていること以外今まで作ってきたログ検索

APIなどと変わりません。
　問い合わせの解決済みステータスを変更するためのAPIも作っておきましょう。以下のAPIも追記して保存してください。

backend/server/api.js

```
router.put('/inquiry/:inquiryId', checkAuthentication, checkCsrf, checkAdministrator, async (req
, res) => {
  try {
    // 型がおかしければ400を返して中断
    if (typeof req.body.resolved != 'boolean') {
      return res.status(400).send();
    }

    const inquiry = await Inquiry.findById(req.params.inquiryId);

    // 指定のIDの問い合わせドキュメントが存在しなければ404を返して中断
    if (!inquiry) {
      return res.status(404).send();
    }

    // 問題なければresolvedを指定の値に設定
    await inquiry.update({
      resolved: req.body.resolved
    });

    return res.status(200).send();
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

　このAPIでは指定のIDのInquiryのresolvedステータスを指定のものに変更します。このAPIにアクセスすることによって問い合わせの解決済みステータスを書き換えます。
　それではページを作りましょう。まずはフロントエンド側にも管理者のみ表示できるようにするミドルウェアを用意する必要があります。「frontend/middleware」の中に「administrator.js」を作成し、以下の内容を入力して保存してください。

frontend/middleware/administrator.js

```
export default function ({ store, redirect }) {
  if (!store.getters['auth/isAuthenticated'] || store.getters['auth/loginCharacter'].eno != 0) {
    return redirect('/');
  }
}
```

　このミドルウェアを指定するとログインしていなかったり、ログインしているキャラクターのENoが0（管理者）でなかったりする場合にトップページにリダイレクトするようになります。
　ミドルウェアができたらページを作ります。「frontend/pages/inquiry」の中に「list.vue」を作成し、以下の内容

を入力して保存してください。

```
frontend/pages/inquiry/list.vue
<template>
  <section>
    <section>
      <sub-heading>問い合わせ一覧</sub-heading>
      <div class="search">
        <div class="search-form">
          <label>未解決の問い合わせのみ表示 <input type="checkbox" v-model="unresolved"></label>
        </div>
      </div>
      <div class="button-wrapper">
        <nuxt-link :to="unresolved ? '?unresolved=t' : ''"><button class="button">検索</button></n
uxt-link>
      </div>
    </section>
    <section>
      <sub-heading>問い合わせリスト</sub-heading>
      <div v-if="list.length">
        <div class="inquiry" v-for="(inquiry, index) in list" :key="index">
          <div class="inquiry-row">
            <div class="inquiry-prop">名前</div>
            <div class="inquiry-val">{{ inquiry.name }}</div>
          </div>
          <div class="inquiry-row">
            <div class="inquiry-prop">アドレス</div>
            <div class="inquiry-val">{{ inquiry.address }}</div>
          </div>
          <div class="inquiry-row">
            <div class="inquiry-prop">日時</div>
            <div class="inquiry-val">{{ inquiry.timestamp | date }}</div>
          </div>
          <div class="inquiry-row">
            <div class="inquiry-prop">内容</div>
            <div class="inquiry-val inquiry-content">{{ inquiry.content }}</div>
          </div>
          <div class="resolved-status-wrapper" @click="toggleResolved(inquiry)">
            <span class="resolved-status resolved-status-resolved" v-if="inquiry.resolved">解決済</
span>
            <span class="resolved-status" v-else>未解決</span>
          </div>
        </div>
        <div class="pagelink-wrapper">
          <nuxt-link
            class="pagelink"
            v-if="Number($route.query.page)"
            :to="`${$route.path}${queryWithoutPage}&page=${Number($route.query.page) - 1}`">
            前のページへ
          </nuxt-link>
          <nuxt-link
            class="pagelink"
            v-if="isContinuing"
            :to="`${$route.path}${queryWithoutPage}&page=${Number($route.query.page || 0) + 1}`">
            次のページへ
          </nuxt-link>
```

```
        </div>
      </div>
      <div v-else>
        問い合わせはありません。
      </div>
    </section>
  </section>
</template>

<script>
import SubHeading from '~/components/SubHeading.vue'
import LogList    from '~/components/LogList.vue'

export default {
  components: {
    SubHeading,
    LogList
  },
  middleware: 'administrator',
  watchQuery: true,
  head() {
    return {
      title: 'ログリスト'
    }
  },
  data() {
    return {
      unresolved:      this.$route.query.unresolved,
      waitingResponse: false
    };
  },
  asyncData: async function(context) {
    const queries  = context.route.fullPath.slice(context.route.path.length);
    const response = await context.$axios.get(`/api/inquiry${queries}`);

    return {
      list:         response.data.list,
      isContinuing: response.data.isContinuing
    }
  },
  filters: {
    date: (str) => {
      const date = new Date(str);
        return (
          ('0' + (date.getMonth() + 1)).slice(-2) + '/' +
          ('0' + (date.getDate()     )).slice(-2) + '(' +
          ['日','月','火','水','木','金','土'][date.getDay()] + ') ' +
          ('0' + (date.getHours()    )).slice(-2) + ':' +
          ('0' + (date.getMinutes()  )).slice(-2)
        );
    }
  },
  computed: {
    queryWithoutPage: {
      get() {
        const queries = [];
        for (const prop in this.$route.query) {
```

```
          if (prop != 'page') {
            queries.push(`${prop}=${this.$route.query[prop]}`);
          }
        }
        return '?' + queries.join('&');
      }
    }
  },
  methods: {
    toggleResolved: async function(inquiry) {
      // 接続に入る、接続待ち状態をON(true)に
      this.waitingResponse = true;

      try {
        // inquiryのステータスを更新
        await this.$axios.put(`/api/inquiry/${inquiry._id}`, {
          resolved: !inquiry.resolved,
          csrf:     this.$store.getters['auth/loginCharacter'].csrf
        });

        // 問い合わせ情報を再取得して情報を更新
        const queries  = this.$route.fullPath.slice(this.$route.path.length);
        const response = await this.$axios.get(`/api/inquiry${queries}`);
        this.list         = response.data.list;
        this.isContinuing = response.data.isContinuing;

        // 接続待ち状態をOFFに
        this.waitingResponse = false;
      } catch (e) {
        console.log(e);
        // エラーが発生した場合接続待ち状態をOFF(false)に
        this.waitingResponse = false;
      }
    }
  }
};
</script>

<style lang="scss" scoped>
.search {
  display: flex;
  align-items: flex-end;

  .search-form {
    margin: 0 20px;
  }
}

.inquiry {
  margin: 20px 0;
  padding: 10px 20px;
  box-sizing: border-box;
  border: 1px solid gray;
  border-radius: 4px;

  .inquiry-row {
    margin: 4px 0;
```

```
    display: flex;

    .inquiry-prop {
      width: 80px;
      flex-shrink: 0;
      font-weight: bold;
      color: #555;
    }

    .inquiry-val {
      white-space: pre-wrap;
    }
  }

  .resolved-status-wrapper {
    display: flex;
    justify-content: flex-end;

    .resolved-status {
      padding: 4px 20px;
      box-sizing: border-box;
      border: 1px solid gray;
      border-radius: 4px;
      font-weight: bold;
      color: #555;
      cursor: pointer;
    }

    .resolved-status-resolved {
      background-color: #666;
      color: #EEE;
    }
  }
}

.pagelink-wrapper {
  padding: 20px;
  display: flex;
  justify-content: space-around;

  .pagelink {
    display: inline-flex;
    justify-content: center;
    min-width: 120px;
    padding: 4px 12px;
    border: 1px solid #333;
    border-radius: 4px;
    text-decoration: none;
    font-weight: bold;
    color: #666;
  }
}
</style>
```

保存したら管理者としてログインし、「http://dev.siroisakana.com/ta/inquiry/list」にアクセスしてみましょう。

(管理者はENo.0で、パスワードは「backend/config/default.json」に"administratorPassword"として記載されています)
以下のようなページが表示されれば成功です。

使われている技術としては特に新しいものはありません。まずAPIにアクセスし問い合わせの一覧を受け取ってリスト表示しています。未解決/解決済のボタンをクリックするとtoggleResolvedが呼び出されるようになっており、APIにアクセスして現在と逆の解決済みステータスをセットするようになっています。また未解決のものに絞って検索もできるようになっており、他の検索ページ同様にページと検索ステータスをAPIに渡すようになっています。

4.16.4 お知らせAPI

次はお知らせページを作っていきます。いちいちお知らせがあるたびにページを更新するのは面倒なので、こちらもWeb上から更新できるようにしましょう。ではAPIを作っていきます。まずはお知らせをデータベースに保管できるようにします。「backend/server/model.js」を開き、以下のスキーマを追加してください。

backend/server/model.js
```
const NoticeSchema = new Schema({
  content:   {type: String, required: true, index: false}, // お知らせ内容
  timestamp: {type: Date,    default: Date.now}             // お知らせ日時
});
```

内容とお知らせ日時だけのシンプルなデータです。追加したら外部から参照できるようにしましょう。末尾部を以下のように書き換えて保存してください。

backend/server/model.js
```
const Character     = mongoose.model('Character',    CharacterSchema    );
const Room          = mongoose.model('Room',         RoomSchema         );
const Message       = mongoose.model('Message',      MessageSchema      );
const Log           = mongoose.model('Log',          LogSchema          );
```

```
const Declaration  = mongoose.model('Declaration',  DeclarationSchema );
const UpdateStatus = mongoose.model('UpdateStatus', UpdateStatusSchema);
const Inquiry      = mongoose.model('Inquiry',      InquirySchema     );
const Notice       = mongoose.model('Notice',       NoticeSchema      );

  (省略)

module.exports = {
  Character:    Character,
  Room:         Room,
  Message:      Message,
  Log:          Log,
  Declaration:  Declaration,
  UpdateStatus: UpdateStatus,
  Inquiry:      Inquiry,
  Notice:       Notice
};
```

次にお知らせを受け取るAPIを作ります。まずはAPIでNoticeを参照できるようにします。「backend/server/api.js」を開き、冒頭を以下のように書き換えてください。

backend/server/api.js

```
  (省略)

const Message     = require('./model.js').Message;
const Log         = require('./model.js').Log;
const Declaration = require('./model.js').Declaration;
const Inquiry     = require('./model.js').Inquiry;
const Notice      = require('./model.js').Notice;
const router      = express.Router();
```

書き換えたら以下のAPIを追記して保存してください。お知らせを更新するためのAPIです。管理者のみアクセスするAPIのためcheckAdministratorを指定します。

backend/server/api.js

```
router.post('/notice', checkAuthentication, checkCsrf, checkAdministrator, async (req, res) => {
  try {
    // 型がおかしければ400を返して中断
    if (typeof req.body.content != 'string') {
      return res.status(400).send();
    }

    // 問題なければ問い合わせ内容をDBに登録
    const notice = new Notice({
      content: req.body.content
    });

    await notice.save();

    return res.status(200).send();
  } catch (e) {
```

```
    console.log(e);
    return res.status(500).send();
  }
});
```

このAPIも値を受け取って問題なければデータベースに登録するだけのシンプルなものです。続けてお知らせ内容を受け取るAPIを作りましょう。ここでは最新のお知らせを表示するAPI(latest)と今までのすべてのお知らせを表示するAPI(archive)の2つを作成します。以下の2つのAPIを追記して保存してください。

backend/server/api.js

```
router.get('/notice/latest', async (req, res) => {
  try {
    let notices = await Notice.find({}, {
      _id:       0,
      content:   1,
      timestamp: 1
    }).sort({timestamp: -1}).limit(1).exec();

    return res.status(200).send({
      notices: notices
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});

router.get('/notice/archive', async (req, res) => {
  try {
    let notices = await Notice.find({}, {
      _id:       0,
      content:   1,
      timestamp: 1
    }).sort({timestamp: -1}).exec();

    return res.status(200).send({
      notices: notices
    });
  } catch (e) {
    console.log(e);
    return res.status(500).send();
  }
});
```

こちらも値を取得して返すだけのAPIです。どちらも値が配列として送信されるという点に注意してください。(お知らせが1件も存在しない場合も含む)

4.16.5 お知らせページ

APIができたのでお知らせページを作成します。まずはお知らせを表示するためのコンポーネントを作成しま

す。「frontend/components」内に「NoticeContent.vue」を作成し、以下の内容を入力して保存してください。

frontend/components/NoticeContent.vue

```
<template>
  <section class="notice">
    <div class="date">
      <div class="date-date">{{ timestamp | month }}/{{ timestamp | date }}</div>
      <div class="date-day">{{ timestamp | day }}</div>
    </div>
    <div class="content" v-html="content.replace(/\r?\n/g, '<br>')">
    </div>
  </section>
</template>

<script>
export default {
  props: {
    timestamp: String,
    content:   String
  },
  filters: {
    month: (str) => {
      const date = new Date(str);
      return date.getMonth() + 1;
    },
    date: (str) => {
      const date = new Date(str);
      return date.getDate();
    },
    day: (str) => {
      const date = new Date(str);
      return ['日','月','火','水','木','金','土'][date.getDay()];
    },
  }
}
</script>

<style lang="scss" scoped>
$font: 'BIZ UDPGothic', 'Hiragino Kaku Gothic Pro', 'ヒラギノ角ゴ Pro W3', 'メイリオ', Meiryo, 'M
S Pゴシック', sans-serif;

.notice {
  margin: 20px;
  display: flex;

  .date {
    width: 100px;
    padding: 10px 0;
    flex-shrink: 0;
    font-family: $font;

    .date-date {
      font-size: 24px;
      font-weight: bold;
      text-align: right;
      padding-right: 20px;
```

```
    }

    .date-day {
      font-weight: bold;
      text-align: right;
      font-size: 16px;
      padding-right: 22px;
      color: #888;
    }
  }

  .content {
    border-left: 4px solid #666;
    padding: 10px 0 10px 20px;
    flex-grow: 1;
  }
}
</style>
```

お知らせ日時(timestamp)とお知らせ内容(content)を受け取って表示するだけのコンポーネントになっており、時刻を整形して表示するため時刻を表すStringを月、日、曜日に変換するfilterが搭載されています。また、ここで表示する内容についてはXSSのリスクを考える必要がないため(記述するのは管理者のみのため)お知らせ内容の表示にはv-htmlを使い、タグなどを自由に使えるようにしています。ただし改行の入力は面倒なので自動的に改行が
となるようになっています。

それではこのコンポーネントを使ってまずはお知らせを送信するページを作成します。「frontend/pages」内に「notice」ディレクトリを作成し、さらにその中に「inform.vue」を作成してください。ファイル構成は以下のようになります。

```
∨ pages
  > inquiry
  ∨ notice
    inform.vue
```

作成したら以下の内容を入力して保存します。

frontend/pages/notice/inform.vue

```
<template>
  <section>
    <sub-heading>お知らせ送信</sub-heading>
    <message-banner type="error" v-if="errorMessage">{{ errorMessage }}</message-banner>
    <section class="form">
      <div class="form-title">内容</div>
      <textarea class="form-textarea" v-model="content" placeholder="内容"></textarea>
    </section>
    <notice-content
      :timestamp="'' + (new Date())"
      :content="content"
      />
```

```
    <div class="button-wrapper">
      <button class="button" @click="send">送信</button>
    </div>
  </section>
</template>

<script>
import SubHeading    from '~/components/SubHeading.vue'
import MessageBanner from '~/components/MessageBanner.vue'
import NoticeContent from '~/components/NoticeContent.vue'

export default {
  components: {
    SubHeading,
    MessageBanner,
    NoticeContent
  },
  middleware: 'administrator',
  head() {
    return {
      title: 'お知らせ送信'
    };
  },
  data() {
    return {
      content:         '', // 問い合わせ内容
      errorMessage:    '', // エラーメッセージ
      waitingResponse: false // 通信待ち中かどうか
    }
  },
  methods: {
    send: async function() {
      if (this.waitingResponse) {
        // 接続待ち状態であればアラートを表示しreturn、以降の処理を行わない
        return alert('しばらくお待ち下さい');
      }

      // 入力内容の検証を行う、問題があればエラーメッセージを表示、returnして処理を中断
      if (!this.content) { // 短縮名が入力されていない
        return this.errorMessage = '内容が入力されていません'
      }

      // 問題がなければ接続に入る、接続待ち状態をON(true)に
      this.waitingResponse = true;

      try {
        // 問い合わせを送信
        const response = await this.$axios.post('/api/notice', {
          csrf:    this.$store.getters['auth/loginCharacter'].csrf,
          content: this.content
        });

        // 送信できたらお知らせページへリダイレクト
        this.$router.push('/notice');
      } catch (e) {
        // エラーが発生した場合接続待ち状態をOFF(false)に
        this.waitingResponse = false;
```

```
        this.errorMessage = '送信中にエラーが発生しました'; // エラーメッセージを表示
      }
    }
  }
}
</script>

<style lang="scss" scoped>
.inquiry-description {
  margin: 20px;
}
</style>
```

保存してから管理者キャラクターで「http://dev.siroisakana.com/ta/notice/inform」にアクセスすると以下のように入力欄とプレビューが表示されるはずです。

特に技術的に複雑な部分はありません。お知らせ内容を入力するtextareaとそれを送信する機能にちょっとしたプレビューがついているだけです。プレビューを行っているNoticeContentにわたす時刻のデータは文字列である必要があるため、「:timestamp="" + (new Date())"」というように指定していることに注意してください。動作が確認できたらこの先のテストのため適当なお知らせを2、3件ほどテスト入力して送信しておきましょう。お知らせ表示ページを作っていないため送信後404ページに飛ばされますが気にしなくても構いません。

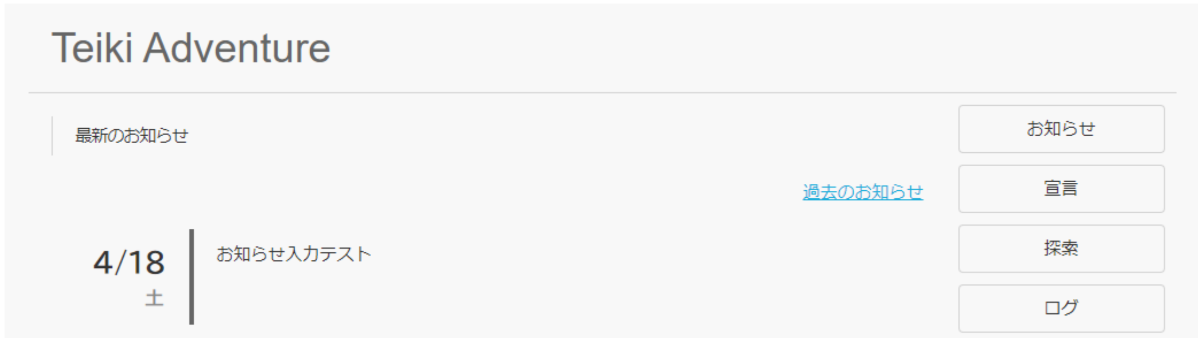
次にお知らせ表示ページを作ります。「/notice」で最新のお知らせが、「/notice/archive」で今までのすべてのお知らせが表示できるようにしましょう。「frontend/pages/notice」内に「index.vue」と「archive.vue」を作成し、それぞれ以下の内容を入力して保存してください。

```
frontend/pages/notice/index.vue
<template>
  <section>
    <sub-heading>最新のお知らせ</sub-heading>
    <div class="archive-link-wrapper">
      <nuxt-link to="/notice/archive">過去のお知らせ</nuxt-link>
    </div>
    <section v-if="notices.length">
      <notice-content
        v-for="(notice, index) in notices"
        :key="index"
        :timestamp="notice.timestamp"
        :content="notice.content"
      />
    </section>
    <section v-else>
      お知らせはありません。
    </section>
  </section>
</template>

<script>
import SubHeading    from '~/components/SubHeading.vue'
import NoticeContent from '~/components/NoticeContent.vue'

export default {
  components: {
    SubHeading,
    NoticeContent
  },
  head() {
    return {
      title: 'お知らせ'
    };
  },
  asyncData: async function(context) {
    const response = await context.$axios.get('/api/notice/latest');

    return {
      notices: response.data.notices
    };
  }
}
</script>

<style scoped>
.archive-link-wrapper {
  display: flex;
  justify-content: flex-end;
}
</style>
```

```
frontend/pages/notice/archive.vue
<template>
  <section>
    <sub-heading>過去のお知らせ</sub-heading>
```

```
    <div class="archive-link-wrapper">
      <nuxt-link to="/notice">最新のお知らせ</nuxt-link>
    </div>
    <section v-if="notices.length">
      <notice-content
        v-for="(notice, index) in notices"
        :key="index"
        :timestamp="notice.timestamp"
        :content="notice.content"
      />
    </section>
    <section v-else>
      お知らせはありません。
    </section>
  </section>
</template>

<script>
import SubHeading    from '~/components/SubHeading.vue'
import NoticeContent from '~/components/NoticeContent.vue'

export default {
  components: {
    SubHeading,
    NoticeContent
  },
  head() {
    return {
      title: 'お知らせ'
    };
  },
  asyncData: async function(context) {
    const response = await context.$axios.get('/api/notice/archive');

    return {
      notices: response.data.notices
    };
  }
}
</script>

<style scoped>
.archive-link-wrapper {
  display: flex;
  justify-content: flex-end;
}
</style>
```

どちらもAPIからデータを受け取って表示しているだけで内容はほぼ同じです。お知らせにアクセスしてみてページが以下のような感じになっていればOKです。



最後に、問い合わせ表示とお知らせ送信をよりやりやすくするため管理者でログインしている場合はサイドメニューにリンクが表示されるようにしましょう。「frontend/components/SideMenu.vue」を開き、<template>内の<div class="mini-link-wrapper">部分と<script>内のcomputed部分をそれぞれ以下のように変更して保存してください。

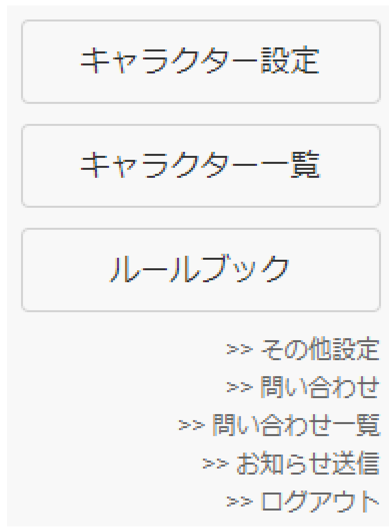
frontend/components/SideMenu.vue

```
    <div class="mini-link-wrapper">
      <nuxt-link v-if="auth" class="mini-link" to="/config">&gt;&gt; その他設定</nuxt-link>
      <nuxt-link class="mini-link" to="/inquiry">&gt;&gt; 問い合わせ</nuxt-link>
      <nuxt-link v-if="admin" class="mini-link" to="/inquiry/list">&gt;&gt; 問い合わせ一覧</nuxt-link>
      <nuxt-link v-if="admin" class="mini-link" to="/notice/inform">&gt;&gt; お知らせ送信</nuxt-link>
      <a v-if="auth" class="mini-link" :href="`${$router.options.base}api/logout`">&gt;&gt; ログアウト</a>
    </div>

  （省略）

  computed: {
    auth: {
      get() {
        return this.$store.getters['auth/isAuthenticated'];
      }
    },
    admin: {
      get() {
        return this.$store.getters['auth/isAuthenticated'] && this.$store.getters['auth/loginCharacter'].eno == 0;
      }
    }
  }
```

管理者としてログインしているかを表すadminが追加され、それに基づいてリンクが表示されるようになっています。adminの算出の際、ログインしていないときthis.$store.getters['auth/loginCharacter']がnullになるためログインしているか判定してから「this.$store.getters['auth/loginCharacter'].eno == 0」とする必要がある点に注意してください。

さて、最後にチェックを行います。管理者でログインした際にサイドメニューが以下のようになっていれば成功で

す。管理者以外でログインした際にこれらの項目が消えていることも確認しておいてください。



ここまで実装すればおおよそ機能的な部分は実装完了です。完成まであと少し、がんばりましょう。

4.17　公開前にするべきこと

4.17.1 ルールブックの作成

ルールブックページを作成しておきましょう。「frontend/pages」内に「rulebook.vue」を作成し、以下の内容を入力して保存してください。

frontend/pages/rulebook.vue

```
<template>
  <section>
    <sub-heading>ルールブック</sub-heading>
    <p>
      これはルールブックページです。
    </p>
  </section>
</template>

<script>
import SubHeading from '~/components/SubHeading.vue'

export default {
  components: {
    SubHeading
  },
  head() {
    return {
      title: 'ルールブック'
    };
  }
}
</script>
```

これはサンプル用のゲームなのでほとんど内容は書いてありませんが、実際には世界観説明や画像規格、戦闘ルール説明などを書いておくといいでしょう。

4.17.2 アイコンの作成

Webページを開くと多くの場合タブのタイトルの横にそのWebページを表すアイコンが表示されます。このアイコンのことを**ファビコン**(favicon)と呼びます。Nuxt.jsではデフォルトで緑の三角形が2つ重なったようなファビコンが設定されていますが、これを独自のアイコンに変えていきましょう。

ファビコンはicoという、あまり聞き慣れない拡張子の画像ファイルで設定することができます。icoは複数の解像度の画像が詰まったアイコン用のファイルになっており、PNG画像などからオンラインツールなどで変換することで作成できます。適当なアイコンを用意しておきましょう。

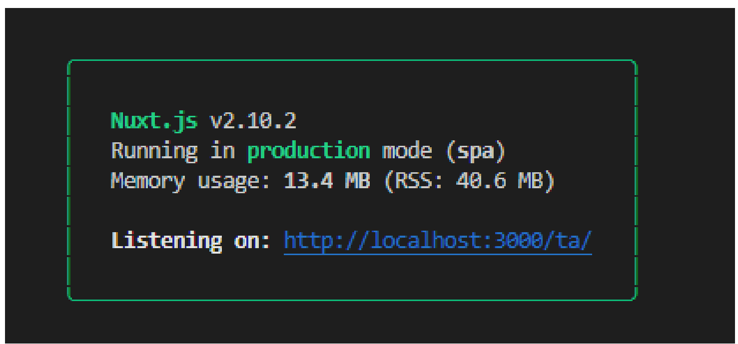
Nuxt.jsではfavicon.icoなどの圧縮や処理などを経ずにそのまま配信されるファイルはstaticディレクトリに入れます。作成したファビコンをfavicon.icoという名前にリネームし、「frontend/static」ディレクトリに保存しましょう。適当なページにアクセスし、アイコンが変わっていれば成功です。(キャッシュが残っている場合はアイコンが変わらないことがあります。その場合スーパーリロードなどを試してみましょう。スーパーリロードはChromeではShift+F5

から実行できます。）

4.17.3 ビルドとプロダクションモード

今までフロントエンドを実行するときは「npm run dev」で実行していましたが、これは開発モードで実行するという意味であり、簡単に開発ができるもののやや動作が重かったりデバッグ用のログが表示されたりしてしまいます。そのため、実際に公開するときは公開用データを構築する**ビルド**を行ってから**プロダクションモード**で公開する必要があります。

ビルドは「npm run build」から実行することができます。フロントエンドを実行している場合はCtrl+Cなどで停止し、「npm run build」を実行しましょう。少し待つといろいろ表示が出てきて処理が完了します。完了したら「npm run start」を実行してみてください。このような感じでRunning in production modeと出ていればOKです。いろいろページにアクセスしてみましょう。



わずかに動作が早くなっていたり、Vue.js devtoolsがNuxt.jsを検出するものの解析は使えなくなっていたり、コンソールにデバッグ用ログが出なくなっていたりするはずです。公開する際はビルドしてプロダクションモードで公開する、ということを忘れずに行いましょう。

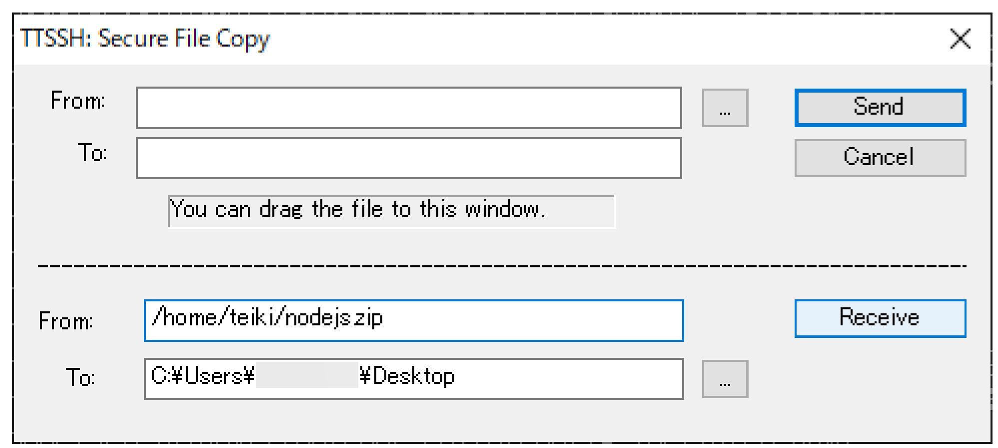
4.17.4 公開用サーバーのレンタルとセットアップ

次に実際にゲームを公開するためのサーバーをレンタルしセットアップしましょう。まずは開発用サーバーをセットアップしたときと同様に初期設定手順を行っていきます。その際の手順は「2.7 手順書」にまとめているので参考にしてください。手順書の処理が完了したら「3.4.4 環境を整える」に従ってVisual Studio Codeから公開用サーバーに接続できるようにします。

環境が整ったら開発用サーバーから公開用サーバーにファイルをコピーしましょう。まずはTera Termで開発用サーバーにログインします。ログインした時点でホームディレクトリだとは思われますが一応「cd /home/teiki/」を実行しホームディレクトリに移動します。

次にnodejsディレクトリを圧縮します。「zip -r nodejs.zip nodejs -x ¥*/node_modules/¥*」を実行してください。圧縮したファイルが肥大化してしまうためnpm installからインストールされるモジュールについてはコピーしないようにしています。実行が完了するとnodejs.zipというファイルが生成されます。

これをPCにダウンロードします。Tera Termの上部メニューのファイル(F)からSSH SCPを選択し、下側の入力欄のFromに「/home/teiki/nodejs.zip」、Toに適当なダウンロード先フォルダを指定し「Receive」をクリックしましょう。



nodejs.zipがダウンロードできたことを確認したら今度は公開用サーバーにnodejs.zipをアップロードします。Tera Termで公開用サーバーにログインし「cd /home/teiki/」を実行したらSSH SCPを実行し、今度は上側の入力欄のFromで先程ダウンロードしたnodejs.zipを選択しToに「/home/teiki」を入力します。入力が完了したら「Send」をクリックしましょう。

TTSSH: Secure File Copy
From: C:¥Users¥ ¥Desktop¥nodejs.zip
To: /home/teiki
Send
Cancel
You can drag the file to this window.
From:
Receive
To: C:¥Program Files (x86)¥teraterm

これで公開用サーバーにアップロードされたので、「unzip nodejs.zip」を実行しファイルを解凍します。nodejsディレクトリ内に開発用サーバーで作成したファイルが展開されているはずなのでVisual Studio Codeなどから確認してみてください。

次はnode_modulesディレクトリを復元します。「cd /home/teiki/nodejs/teikiadv/frontend」を実行してください。node_modulesディレクトリを復元するには単に「npm install」を実行します。これで自動的に作成されたインストール設定から復元できます。しばらく待つと処理が完了するはずです。このとき「found *** vulnerabilities」(***が1以上)と表示されていれば「npm audit fix」も実行しておきましょう。

バックエンド側も同様に行います。「cd /home/teiki/nodejs/teikiadv/backend」を実行し「npm install」を実行。「found *** vulnerabilities」(***が1以上)と表示されていればさらに「npm audit fix」を実行。これでバックエンド側のnode_modulesの復元も完了です。

それが終わったらMongoDBにデータベースを作成します。「4.1.5 データベースの作成」に従って適宜データーベースを作成してください。

次に設定を公開用サーバーに合わせ編集しましょう。Visual Studio Codeから「frontend/nuxt.config.js」を

開きます。「axios」の「browserBaseURL」や「env」の「logDirectory」「resultDirectory」は開発用サーバーのURLが設定されていると思われるので、ここを公開用サーバーに合わせて修正します。

バックエンド側も同様に行います。「backend/config/default.json」を開き、「dbName」「dbPassword」「dbPassword」を適宜公開用サーバー用のものに書き換えてください。「sessionSecretKey」も適当なものに変えておきましょう。「logoutRedirect」も公開用サーバーのURLに合わせて書き換えてください。また、現時点ではAPを配布しないと思われるので「givingAp」もfalseにしておきます。

次に探索戦ログや物語戦ログの保存場所を作成します。Tera Termのコンソールから「su」を実行しrootに昇格し以下のコマンドを順番に実行してください。

「cd /var」
「mkdir -p www/teikiadv/explore/log」
「mkdir -p www/teikiadv/result」
「cd www」
「chown -R teiki:teiki teikiadv」

次に初期化処理を呼び出します。Visual Studio Codeのコンソールから「cd /home/teiki/nodejs/teikiadv/backend」を実行し、「npm run init」から初期化を行います。さて、この先公開用サーバーで初期化処理を呼び出すことはほぼないと思われるので開発用サーバーと間違えて初期化してしまうミスを防ぐため「npm run init」をしても初期化処理が呼び出されないようにします。

「backend/package.json」を開き、「scripts」の「init」の行を消してしまいましょう。やや強引ですがこれで誤操作でユーザーのデータをすべて消してしまうミスを防げます。公開終了後などに初期化したい場合は開発用サーバーからinitの記述を持ってきて復元してから初期化しましょう。

次にAPの配布処理をcronに記述しておきます。Visual Studio Codeのコンソールから「crontab -e」を実行し、以下の内容を入力して保存してください。

crontab -e
```
0 20 * * * cd /home/teiki/nodejs/teikiadv/backend; npm run ap
```

次にnginxの設定を変更します。Tera Termのコンソールからrootユーザーで「vi /etc/nginx/conf.d/default.conf」を実行し以下の内容を「location /」となっているセクションの次の箇所に追記します。

/etc/nginx/conf.d/default.conf
```
    location /ta/ {
        proxy_pass http://127.0.0.1:3000;
    }

    location /ta/api/ {
        proxy_pass http://127.0.0.1:4000;
    }

    location /ta/log/ {
        alias  /var/www/teikiadv/explore/log/;
```

```
    }

    location /ta/result/ {
        alias   /var/www/teikiadv/result/;
    }
```

追記したら「nginx -s reload」を実行し設定をリロードしましょう。これで環境を移すことができました。テストとしてフロントエンドとバックエンドを実行し、新しいサーバーからもTeiki Adventureにアクセスできることを確認してください。

もしフロントエンド実行時に以下のようなエラーが出る場合はバージョンが低い方のモジュールをアップデートしてください。例えば前者の画像のようになっていれば「npm install vue」を、後者のようになっていれば「npm install vue-template-compiler」を実行します。（両方出る場合もあります。その場合は両方行います。）

```
Vue packages version mismatch:

- vue@2.6.10
- vue-server-renderer@2.6.11

This may cause things to work incorrectly. Make sure to use the same version for both.
```

```
Module Error (from ./node_modules/vue-loader/lib/index.js):            friendly-errors 16:10:10


Vue packages version mismatch:

- vue@2.6.11 (/home/teiki/nodejs/teikiadv/frontend/node_modules/vue/dist/vue.runtime.common.js)
- vue-template-compiler@2.6.10 (/home/teiki/nodejs/teikiadv/frontend/node_modules/vue-template-comp
iler/package.json)
```

また、以下のようなエラーが出る場合は「4.1.8 ENOSPCエラーが出る場合」の手順に従ってファイルウォッチャー数を増やしてください。

```
 ERROR  ENOSPC: System limit for number of file watchers reached, watch '/home/teiki/nodejs/teikiad
v/frontend/node_modules/@nuxt/vue-app/template/views/loading'
```

最後にフロントエンド側のビルドを再構築しておきましょう。フロントエンド側で「npm run build」を実行しておいてください。Generated ... というような表示が出ればOKです。

4.17.5 PM2

Node.jsは万が一異常終了したりVPSが突然停止してしまった際に動作が停止してしまいます。そこで利用するのが**PM2**です。PM2は何らかの原因でNode.jsアプリケーションが終了してしまった際に自動で再起動してくれます。このようにすることを**永続化**と呼びます。

また、多くのVPSではマルチコアのCPUを利用できますが、Node.jsは基本シングルスレッドで動作するため使わないCPUが生まれてしまいせっかくのマルチコアが無駄になってしまうことがあります。そのため、アプリケーショ

ンを複数起動して並列して処理を行わせるのが一般的です。これを**クラスタリング**と呼び、この機能もPM2に備わっています。クラスタリングにより起動されたそれぞれのアプリケーションを**インスタンス**と呼びます。

まずはPM2を公開用サーバーにインストールしましょう。PM2はnpmからインストールすることができます。このとき特定のプロジェクトに対してインストールするのではなくシステム全体にインストールします。

Tera Termで公開用サーバーにログインし、rootに昇格して「npm install -g pm2」を実行してください。-gを指定することでシステム全体にインストールできます。「pm2 -version」を実行してバージョン情報が帰ってくればインストールできています。

さて、PM2のクラスタリング機能ですが単純にクラスタリングするだけではうまくいきません。例えば元のアプリケーションが3000番ポートを利用して動作する場合、複数起動した際に3000番ポートの取り合いになってしまってエラーが発生してしまいます。そこでインスタンスごとに動作するポートを変更する必要があります。それに対応するため、PM2のforkモードでクラスタリングした際にインスタンス側ではprocess.env.NODE_APP_INSTANCEから自身のインスタンスIDを知ることができます。インスタンスIDは0から始まる連番になっていて、例えば利用できるCPUが8スレッドの場合インスタンスIDは0〜7の値を取ります。これを利用して動作ポートを変更しましょう。

まずはNuxt.js側から変更します。公開用サーバーの「frontend/nuxt.config.js」の冒頭の「server」の「port」の部分を以下のように書き換えて保存してください。

frontend/nuxt.config.js

```
  server: {
    port: 3000 + parseInt(process.env.NODE_APP_INSTANCE || 0)
  },
```

次にバックエンド側を変更しましょう。公開用サーバーの「backend/server/index.js」の冒頭部分の「const port = ...」の部分を以下のように書き換えて保存してください。

backend/server/index.js

```
const port          = config.port + parseInt(process.env.NODE_APP_INSTANCE || 0);
```

次にNginxの設定を変更しましょう。アプリケーションが動作するポートが変わったため、それに対応して設定を書き換えます。公開用サーバーにログインしてrootユーザーになり「vi /etc/nginx/conf.d/default.conf」を実行して先頭(一行目)に以下の内容を書き加えてください。

このとき、利用しているVPSのCPUのスレッド数に応じてserver 127.0.0.1:XXXX;の行数は適宜書き換えてください。以下はCPUのスレッド数が4スレッドの場合の例となっています。(例えばスレッド数が2スレッドなのであればそれぞれ先頭2行のみを残します。)CPUのスレッド数はVPS契約時にどこかに記載されているかと思われますが、「grep processor /proc/cpuinfo | wc -l」を実行することでも調べることができます。

/etc/nginx/conf.d/default.conf

```
upstream teikiadv_frontend {
    server  127.0.0.1:3000;
    server  127.0.0.1:3001;
    server  127.0.0.1:3002;
    server  127.0.0.1:3003;
```

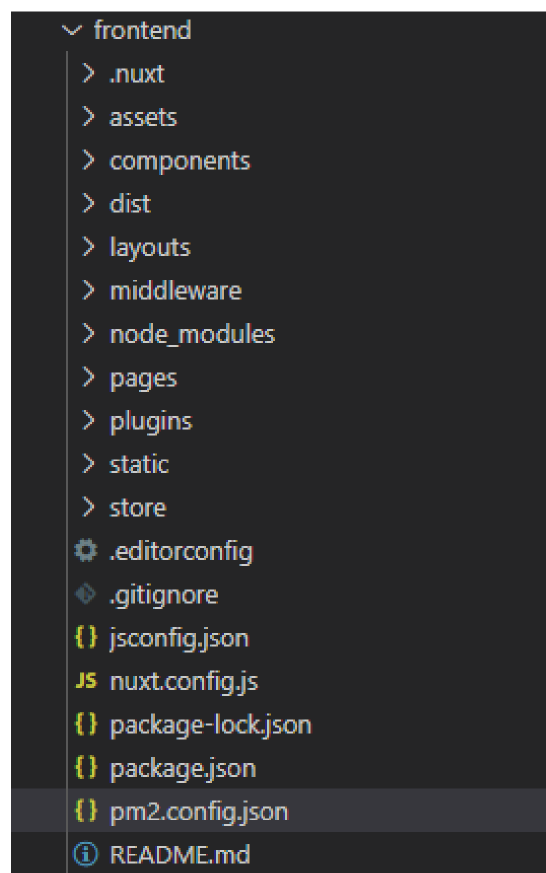
```
}

upstream teikiadv_backend {
    server  127.0.0.1:4000;
    server  127.0.0.1:4001;
    server  127.0.0.1:4002;
    server  127.0.0.1:4003;
}
```

次にlocationの/ta/と/ta/api/の部分を以下のように書き換えてください。書き換え終わったら保存し「nginx -s reload」を実行して変更を反映します。

/etc/nginx/conf.d/default.conf

```
    location /ta/ {
        proxy_pass  http://teikiadv_frontend;
    }

    location /ta/api/ {
        proxy_pass  http://teikiadv_backend;
    }
```

次にPM2の設定ファイルをフロントエンドとバックエンドそれぞれに記述します。まずは公開用サーバーの「frontend」ディレクトリに「pm2.config.json」ファイルを作成してください。ファイル構成は以下のようになります。



作成したら以下の内容を入力して保存してください。

frontend/pm2.config.json

```
{
  "name": "teikiadv_frontend",
  "script": "npm",
  "args": "run start",
  "exec_mode": "fork",
  "instances": 0
}
```

それぞれの設定項目は以下のような指定になっています。

name	アプリケーション名
script	実行するコマンド
args	実行するコマンドに渡す引数
exec_mode	実行モード
instances	実行するインスタンス数 （0であればCPUのスレッド数と同数起動）

記述したらバックエンド側も同様に行いましょう。公開用サーバーの「backend」内に「pm2.config.json」を作成してください。ファイル構成は以下のようになります。

```
backend
  > config
  > node_modules
  > server
  > template
  {} package-lock.json
  {} package.json
  {} pm2.config.json
```

作成したら以下の内容を入力して保存します。

backend/pm2.config.json

```
{
  "name": "teikiadv_backend",
  "script": "node server/index.js",
  "exec_mode": "fork",
  "instances": 0
}
```

これでPM2に関する設定は完了です。Visual Studio Codeのコンソールから公開用サーバーのフロントエンドとバックエンドディレクトリに移動しそれぞれ「pm2 start pm2.config.json」を実行しましょう。以下のような感じになればOKです。(id、cpu、memoryの部分は例から多少変わることがあります)

```
[teiki@            frontend]$ pm2 start pm2.config.json
[PM2][WARN] Applications teikiadv_frontend not running, starting...
[PM2] App [teikiadv_frontend] launched (2 instances)
```

id	name	mode	↺	status	cpu	memory
2	teikiadv_backend	fork	0	online	0%	33.8mb
3	teikiadv_backend	fork	0	online	0%	33.6mb
4	teikiadv_frontend	fork	0	online	0%	11.5mb
5	teikiadv_frontend	fork	0	online	0%	11.6mb

次に再起動時にPM2が自動起動するようにしましょう。フロントエンドとバックエンドどちらのコンソールでも構いませんので「pm2 startup」を実行します。すると以下のような表示になります。

```
[teiki@            frontend]$ pm2 startup
[PM2] Init System found: systemd
[PM2] To setup the Startup Script, copy/paste the following command:
sudo env PATH=$PATH:/usr/bin /usr/lib/node_modules/pm2/bin/pm2 startup systemd -u teiki --hp /home/teiki
```

内容としては「自動起動用スクリプトを生成したので以下のコマンドをコピー＆ペーストで実行して自動起動をセットアップしてください」というような感じです。ただしsudoというコマンド(rootで続くコマンドを実行するためのコマンド)を使うための手順を本書では行っていないので、ここではTera Termで開発用サーバーにログインしてrootに昇格し、sudo以降のコマンドを実行してください。この画像の例では以下のようになります。表示されたコマンドによって適宜読み替えて実行してください。

「env PATH=$PATH:/usr/bin /usr/lib/node_modules/pm2/bin/pm2 startup systemd -u teiki --hp /home/teiki」

「Command successfully executed.」というような表示が出たらTera Termを閉じてVisual Studio Code側のコンソールに戻り「pm2 save」を実行します。以下のような表示になれば自動起動設定は完了です。

```
[teiki@            frontend]$ pm2 save
[PM2] Saving current process list...
[PM2] Successfully saved in /home/teiki/.pm2/dump.pm2
```

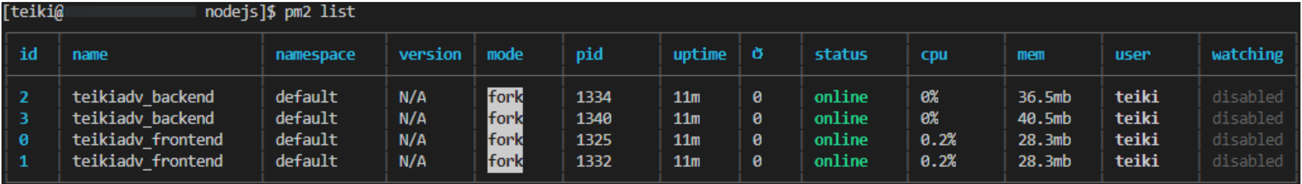
試しに再起動を行ってみましょう。「reboot」を実行します。rootのパスワードが求められるので入力して再起動を実行します。しばらく待ってからブラウザから公開用サーバーのTeiki Adventureにアクセスしてみてください。特に「npm run …」などを行っていないのに起動できているはずです。なお、このときVisual Studio Codeの接続が切れるのでウィンドウの再読み込みを実行しておきましょう。

PM2の設定はこれで完了ですが管理する上で使うであろうコマンドについても記載しておきます。PM2では実行するユーザーごとに操作対象となるアプリケーションが分けられているので以下のコマンドを実行する際はteiki

ユーザーで実行してください。

pm2 list

PM2で実行しているアプリケーションのリストと詳細情確認することができます。実行する以下のような表示になります。



それぞれの項目は以下のような意味です。

id	アプリケーションID
name	アプリケーション名
namespace	名前空間
version	アプリケーションのバージョン
mode	実行モード
pid	アプリケーションの
uptime	アプリケーションの起動時間(再起動でリセットされる)
円形矢印アイコン	再起動された回数
status	実行状況(onlineで実行中、stoppedで停止中)
cpu	CPU使用率
mem	メモリ使用量
user	実行しているユーザー
watching	ファイルを監視して再起動するかどうか

pm2 logs (アプリケーション名/ID)

実行ログを確認するコマンドです。「pm2 logs teikiadv_frontend」や「pm2 logs 3」のようにアプリケーション名やIDで確認する対象を選べます。終了する場合はCtrl+Cから離脱できます。

pm2 stop (アプリケーション名/ID)

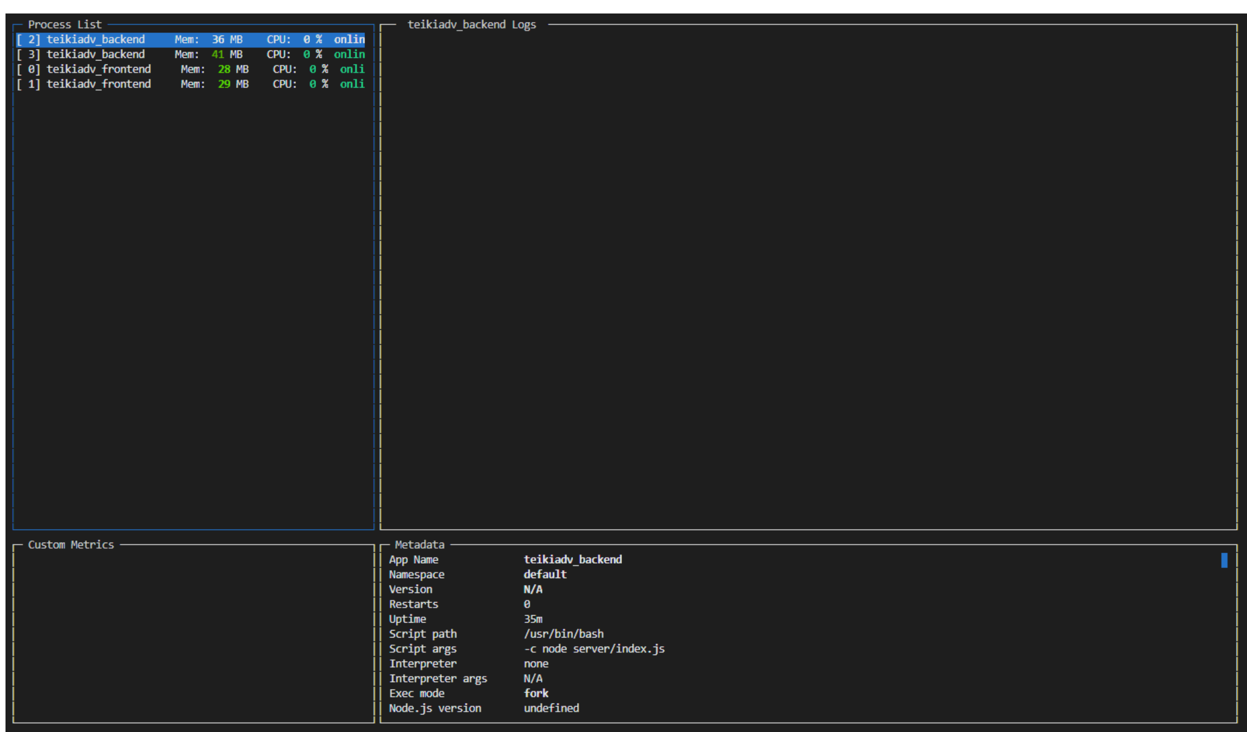
対象のアプリケーションを停止するコマンドです。

pm2 restart (アプリケーション名/ID)

対象のアプリケーションを再起動するコマンドです。

pm2 monit

　GUIライクなモニターを表示し、管理しているアプリケーションの状態を一括で確認できるコマンドです。実行すると以下のようになります。こちらも離脱はCtrl+Cから行えます。



4.17.6 バックアップの定期実行

　データベースは万が一のことがあってもデータが壊れないように耐障害性が高く設計されていることがほとんどですが、それでも操作ミスやコーディングミスなど様々な理由によってデータが破損してしまう可能性があります。そのためそのような事態になってもユーザーのデータが失われないようにデータベースのバックアップを定期的に実行するようにしましょう。

　これには**バッチ**という仕組みを利用します。バッチとは予め実行するコマンドをファイルなどに記述しておき、指定した一連の処理をひとまとめに行えるようにする仕組みのことを指し、多くの場合簡易的な変数なども扱えたりします。CentOSにおいてバッチは.shという拡張子のテキストファイルに記述します。バックアップを作成するバッチを作成してcronで定期実行することでバックアップを定期的に作成するというわけです。

　それではバッチを作成していきます。公開用サーバーにTera Termでログインしてください。teikiユーザーのままで構いません。ホームディレクトリの中に「teikiadv_db_backup.sh」というファイル名でバッチを作成していきましょう。「vi /home/teiki/teikiadv_db_backup.sh」を実行します。エディターが立ち上がるので以下の内容を入力して保存してください。

/home/teiki/teikiadv_db_backup.sh

```
#!/bin/bash

readonly PORT="17089" # MongoDBのポート
readonly DB_NAME="teikiadv" # バックアップするデータベースの名前
```

```
readonly USERNAME="teikiadv" # バックアップするデータベースのユーザーの名前
readonly PASSWORD="6+1Q(N&si-&z" # バックアップするデータベースのユーザーのパスワード
readonly BACKUP_DIRECTORY="/home/teiki/teikiadv_db_backup" # バックアップ先のディレクトリ
readonly KEEP_TIME=10080 #バックアップが有効な期間（分）

readonly current_date=`date +%Y%m%d%H%M` # 現在時刻

# バックアップを作成
mongodump --port ${PORT} -u ${USERNAME} -p ${PASSWORD} --db ${DB_NAME} --out ${BACKUP_DIRECTORY}/backup${current_date}

# 作成したバックアップを圧縮
zip -r ${BACKUP_DIRECTORY}/backup${current_date}.zip ${BACKUP_DIRECTORY}/backup${current_date}

# 圧縮元を削除
rm -rf ${BACKUP_DIRECTORY}/backup${current_date}

# 古くなったバックアップを削除
find ${BACKUP_DIRECTORY} -name "backup????????????.zip" -mmin +${KEEP_TIME} -delete
```

バッチは上から実行されていきます。一行目の「#!/bin/bash」についてはおまじないのようなものと思って構いません。「readonly … = …」の部分は使う設定を記述しています。環境に応じて適宜設定は書き換えてください。BACKUP_DIRECTORYはバックアップの保存先の指定になっていて、「/home/teiki/teikiadv_db_backup」ディレクトリに保存されるようにしています。また、KEEP_TIMEはバックアップが有効な期間の指定で、ここでは10080分（1週間）を指定しています。

次にcurrent_date = `date +%Y%m%d%H%M`でファイル名につけるための現在時刻を取得します。例えば今が2020年4月1日10:00だとするとcurrent_dateは"202004011000"となります。

以降は変数の内容を利用してコマンドを実行するだけです。mongodumpはバックアップを作成するためのコマンドで、これによりバックアップがディレクトリとともにいくつかのファイルとして生成されます。バックアップとしては圧縮して1つのファイルにまとまっていたほうが便利なので、zipコマンドを利用して生成されたバックアップを圧縮し圧縮する前のディレクトリは削除します。最後にバックアップが増えすぎて容量を圧迫するのを防ぐためKEEP_TIME以上に作成されてから時間が過ぎているファイルについては削除します。

バッチができたら設定に従って保存先ディレクトリを作成します。「mkdir /home/teiki/teikiadv_db_backup」を実行してください。

次にcronでバッチを定期実行するようにしましょう。「crontab -e」を実行し、以下の行をAP配布処理の下に追記してください。以下の例では毎日午前4:00にバックアップを実行します。

crontab -e

```
0 4 * * * sh /home/teiki/teikiadv_db_backup.sh
```

4.17.7 その他設定

あとは公開に向けて細かな設定等を行っていきましょう。まずはフロントエンドでは検索に載らないように設定を記述していたのですがその記述を外し検索に載るようにします。公開用サーバーの「frontend/nuxt.config.js」を開き、「head」内部の「meta」の「hid: 'robots'」と書いてある行を削除して保存してください。削除後のmeta部は

以下のようになります。

frontend/nuxt.config.js

```
  meta: [
    { charset: 'utf-8' },
    { name: 'viewport', content: 'width=device-width, initial-scale=1' },
    { hid: 'description', name: 'description', content: '定期・APゲームのサンプル' }
  ],
```

記述したら変更を反映します。Visual Studio Codeのコンソールから公開用サーバーのフロントエンドのディレクトリに移動し「pm2 stop teikiadv_frontend」を実行してフロントエンドを停止、「npm run build」から再度ビルドを行い「pm2 start teikiadv_frontend」を実行して再度フロントエンドを起動します。これで設定が反映されます。

次にトップページ以外が検索に引っかからないようにします。これには**robots.txt**を利用します。robots.txtにアクセス制御の内容を記述することで検索エンジンはその内容を元にどのページは検索結果に載せてよくてどのページは載せてはいけないのかを判断します。それではrobots.txtを記述します。公開用サーバーにrootでログインして「vi /usr/share/nginx/html/robots.txt」を実行し、以下の内容を記述して保存してください。

/usr/share/nginx/html/robots.txt

```
User-agent: *
Disallow: /ta/
Allow: /ta/$
```

上記の設定では/ta/ディレクトリ以下で一律で検索エンジンロボットからのアクセス拒否を行い、例外的に/ta/ディレクトリに完全一致する場合のみ許可をしています。これによりトップページのみ検索結果に載せてそれ以外は載せないようにすることができます。

4.17.8 開発用サーバーのアクセス制限

現状開発用サーバーで作っている定期ゲームはどこからでもアクセスできるようになっていますが、ゲームを公開する上ではネタバレ等防止のためにアクセス制御をしたほうがよいでしょう。アクセス制御の手順は開発環境が固定IP環境かそうでないかで分かれるためそれぞれ自身の環境に合ったセクションに進んでください。

固定IP環境

もし自宅のインターネット環境が固定IPであり、自宅でしか開発を行わない場合は簡単にアクセス制御することができます。単に自身以外のIPの接続をブロックすればよいだけです。

開発サーバーにログインしてrootに昇格し「vi /etc/nginx/conf.d/default.conf」を実行、location /ta/ ～ /ta/result/の箇所を以下のように書き換えて保存してください。「allow」のIP部分（123.456.789.123）に関してはご自身の環境の固定IPに合わせて変更してください。

```
/etc/nginx/conf.d/default.conf
location /ta/ {
    proxy_pass  http://127.0.0.1:3000;
    allow       123.456.789.123;
    deny        all;
}

location /ta/api/ {
    proxy_pass  http://127.0.0.1:4000;
    allow       123.456.789.123;
    deny        all;
}

location /ta/log/ {
    alias  /var/www/teikiadv/explore/log/;
    allow  123.456.789.123;
    deny   all;
}

location /ta/result/ {
    alias  /var/www/teikiadv/result/;
    allow  123.456.789.123;
    deny   all;
}
```

保存したら「nginx -s reload」を実行してください。アクセス制御が行われるようになります。開発環境からはアクセスできてそれ以外の環境からはアクセスできないことを確認してください（Wi-Fiではなくモバイルデータで接続している状態のスマートフォンからアクセスしてみるなど）。

非固定IP環境

IPが固定されていない環境ではアクセス制御をするために若干の手順を踏む必要があります。いくつか方法はあるのですが、ここではcookieをパスワードとしてアクセス制御をする方法を取ります。まず使用するパスワードを決めてください。ここでは「4CEAc|5.~x%(」とします。このパスワードについてはユーザーに開発用サーバーが見られないようにする目的でそこまでセキュリティについて考えているわけではないので他のパスワードとは別個にしてください。

それでは設定していきましょう。開発サーバにログインしてrootに昇格し「vi /etc/nginx/conf.d/default.conf」を実行、location /ta/ ～ /ta/result/の箇所を以下のように書き換えて保存してください。「4CEAc|5.~x%(」の部分は適宜書き換えてください。

```
/etc/nginx/conf.d/default.conf
location /ta/ {
    set $interrupt true;

    if ($cookie_secret = "4CEAc|5.~x%(") {
        set $interrupt false;
    }

    if ($interrupt = true) {
        return 403;
```

```
        }

        proxy_pass  http://127.0.0.1:3000;
    }

    location /ta/api/ {
        set $interrupt true;

        if ($cookie_secret = "4CEAc|5.~x%(") {
            set $interrupt false;
        }

        if ($interrupt = true) {
            return 403;
        }

        proxy_pass  http://127.0.0.1:4000;
    }

    location /ta/log/ {
        set $interrupt true;

        if ($cookie_secret = "4CEAc|5.~x%(") {
            set $interrupt false;
        }

        if ($interrupt = true) {
            return 403;
        }

        alias  /var/www/teikiadv/explore/log/;
    }

    location /ta/result/ {
        set $interrupt true;

        if ($cookie_secret = "4CEAc|5.~x%(") {
            set $interrupt false;
        }

        if ($interrupt = true) {
            return 403;
        }

        alias  /var/www/teikiadv/result/;
    }
```

保存したら「nginx -s reload」を実行してください。アクセス制御が行われるようになります。アクセスすると以下のように「403 Forbidden」が表示されるはずです。

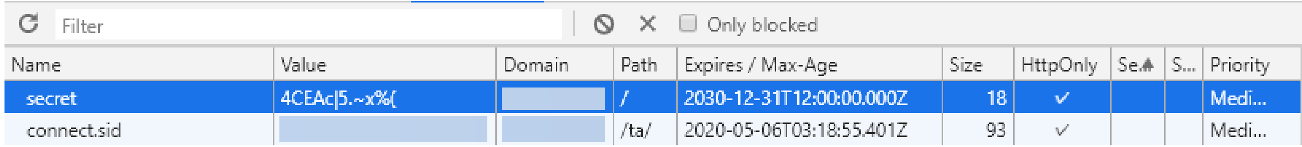
403 Forbidden

---

nginx/1.16.1

ここからアクセスを行いましょう。以下はChromeでの手順です。F12をクリックし開発者ツールを表示し「Application」タブから「Storage」内の「Cookies」の「http://dev.siroisakana.com」を選択します。Cookieの編集パネルが開くので空白の箇所をダブルクリックしてCookie編集モードに移り以下のようにCookieを記述してください。特に記述のない箇所はデフォルトのままで構いません。

Name	secret
Value	設定したパスワード
Expires / Max-Age	2030-12-31T12:00:00.000Z（適当な未来の時刻）
HttpOnly	チェックを有効化



設定の記述が終わったらリロードしてみましょう。「403 Forbidden」ではなくページが表示できればOKです。

4.17.9 更新の反映手順

定期・APゲームでは更新やバグ修正などでコードを変更することがよくあります。ここでは公開用サーバーにおいて更新を反映するための手順を解説しておきます。

まず開発用サーバーでのファイルの変更を公開用サーバーにも反映します。PHPではこれで反映は終了ですがNode.jsはアクセスごとに起動される方式ではないのでプロセスを再起動する手順を踏む必要があります。

バックエンドの更新を行う場合

teikiユーザーでログインし、「pm2 restart teikiadv_backend」を実行します。なお、実行の際のディレクトリはバックエンドディレクトリでなくても構いません。

フロントエンドの更新を行う場合

フロントエンド側ではビルドをしてから再起動という手順を踏む必要があります。まずteikiユーザーでログインしフロントエンドディレクトリに移動して「npm run build」を実行しビルドが完了したら「pm2 restart teikiadv_frontend」を実行します。

Chapter 5
後書き・おまけ

後書きの他、本編中では書ききれなかった
定期・APゲームの開発に使えそうな
技術の紹介などを行っていきます。

5.1 あとがき

この度は「Node.jsで作る定期・APゲーム」を読んで頂きありがとうございます。冒頭でも少し触れましたが、本書は「定期ゲ Advent Calendar 2019」の1記事として寄稿される予定のものでした。（結果としては詰め込もうとしすぎて期日までに完成させることはできませんでしたが……）

普段は定期ゲーの世界の隅っこの方で空気を吸って楽しむというような生活をしているのですが、いただいた楽しみを還元したいという想いと本のようなものを書いてみたいという想いが前々からあり、せっかくの機会ということで寄稿させていただきました。このような機会を作ってくださった25氏さん、ゆうさんにこの場を借りてお礼します。ありがとうございました。

本書は定期・APゲームを作りたいけど作り方が分からない！という方に向けてなるべく丁寧に、本書だけである程度のものは作れるようにというコンセプトで書かれました。その分、文量はすごいことになってしまいましたが。最初は100pぐらいで解説できるかなと思っていたのですが、定期ゲーは例えるならば総合芸術のようなもので本当に様々な技術の上に成り立っているものだと実感しました。技術を開発された方々、定期・APゲームGMの方々の努力の結果の上に定期・APゲームは成り立っているのだなと感じます。ありがとうございます。本書もそのような世界の中で一つの手助けになることができればこれ以上の喜びはありません。

筆者は普段Node.jsをメインで使っている関係で本書はNode.jsで定期・APゲームを作る方法を解説しましたが、定期ゲーの運営環境はレンタルサーバー（PHP）が大多数を占めているため「レンタルサーバー（PHP）で作る定期・APゲーム」も書きたいですね。いずれ機会があれば。

続く「おまけ」では本文中では紹介しきれなかった技術や開発のヒントになりそうなものをいくつか紹介しています。本書の内容も残りわずかとなりましたがお楽しみいただければ幸いです。ここまで読んで頂き、本当にありがとうございました。

2020年04月23日

('ω')<) @siroi_sakana

5.2 おまけ

5.2.1　PageSpeed Insights

PageSpeed InsightsはGoogleが公開しているサイトの高速化を図れるツールです。以下のURLからアクセスすることができ、分析したいサイトのURLを入力すればサイトがどれだけ高速化できているかの点数と改善点を表示してくれます。

PageSpeed Insights
https://developers.google.com/speed/pagespeed/insights/

Nuxt.jsなどを利用している場合数多くの最適化を自動的に行ってくれますが、より快適なウェブサイトを目指す場合参考になるヒントが提案されるかもしれません。

5.2.2　Can I use...

各ブラウザはW3CやWHATWGなどが定めた仕様に従って機能を実装していますが、機能の実装の速さやバグ、独自仕様や更新停止などの理由によりブラウザによって使えるタグやCSSのスタイル、JavaScriptの構文が異なったりします。これを**ブラウザ間差異**などと呼びます。
ある機能に対してどのブラウザが対応しているかを1つ1つ調べるのは非常に面倒です。そこで利用するのが対応状況を簡単に調べることができるサイトである**Can I use...**です。Can I use...は以下のURLよりアクセスすることができます。

Can I use... Support tables for HTML5, CSS3, etc
https://caniuse.com/

英語ですが、わかりやすいUIなので直感的に対応状況を把握できると思います。Can I useと書かれた入力欄に対応状況を調べたいタグやスタイルなどを入力することで対応状況を一覧で見ることができます。緑色で表示されているものが完全対応、黄緑色は部分的対応（試験的に一部のみ実装されている場合やバグなどが存在する場合など）、赤色が非対応となっています。

5.2.3　metaタグ

ここでは様々なmetaタグを紹介しています。なお、Nuxt.jsにおいてはmetaは「nuxt.config.js」のhead内のmetaにて指定することができます。

format-detection

iPhoneなどにはWebページ内に電話番号やメールアドレスらしき文字列があった場合に自動でリンクにするという機能が備わっています。format-detectionはその機能を制御するためのmetaタグです。定期・APゲームでは与ダメージなどの桁数が大きい場合勝手にそれが電話番号として表示されてしまうことがあるので、それを防ぐためにこのmetaタグはつけておいたほうがいいでしょう。以下のように指定します。この例ではメールアドレス、電話

番号、住所の自動リンクを無効化しています。

```
<meta name="format-detection" content="email=no,telephone=no,address=no">
```

keywords

keywordsはそのサイトがどのようなサイトなのかを表すキーワードを指定することができるmetaタグです。さて、このkeywordsですが検索エンジンがまだ未熟だった時代に検索エンジンに情報を伝えることができるタグとして重宝されていましたが、現在Googleではこのタグを全く参考にせずサイトの内容からキーワードをピックアップしています。なので現在では設定する必要はありません。しばらくサイト作成に触れていなかった方のために紹介しておきます。

Google Webmaster Central Blog: Google does not use the keywords meta tag in web ranking
https://webmasters.googleblog.com/2009/09/google-does-not-use-keywords-meta-tag.html

Twitterカード

Twitterにおいてリンクを含んだツイートを送信すると自動的に記事の内容が小さく表示されることがあります。それを指定するものが**Twitterカード**です。Facebookなど他のSNSにも同様の機能があり、これら仕組みのことを**OGP**（Open Graph Protocol）と呼びます。なお、Nuxt.jsにおいてOGPを利用する場合SSRを有効化する必要があるので注意が必要です。

Twitterカードの使い方についてはこちらの記事が非常に詳しいので紹介しておきます。

【2020年版】Twitterカードとは？使い方と設定方法まとめ
https://saruwakakun.com/html-css/reference/twitter-card

また、公式によるTwitterカードのガイドは以下のURLからアクセスできます。

カードの利用開始 — Twitter Developers
https://developer.twitter.com/ja/docs/tweets/optimize-with-cards/guides/getting-started

theme-color

theme-colorはAndroidのChromeにおいてタブや上部のアドレスバーなどの色を変更するためのmetaタグです。AndroidのChrome以外ほとんど対応していない珍しいタグになりますが、設定すると周りと違う個性を出せるかもしれません。以下は黄色を設定している例です。

```
<meta name="theme-color" content="#ffff00">
```

5.2.4 フォント

フォント(font)とは一般的にコンピュータで表示されたり印刷されたりする書体のことを指します。見た目の印象においてフォントの占める割合は大きく、逆に言えばフォントを適切に選択することで見た目の印象をぐっと変えることが可能です。試しに本書で作成した『Teiki Adventure』のタイトルをいろいろなフォントで表示してみましょう。

Teiki Adventure
Goudy Old Style(URW Type Foundry)

Teiki Adventure
Mason Serif OT Regular (Jonathen Barnbrook)

Teiki Adventure
スキップ M(フォントワークス)

Teiki Adventure
Orbitron (Matt McInerney)

白背景に黒字だけの表示ではありますが、例えばMason Serifであれば西洋風ファンタジー世界観のゲームに、Orbitronであれば近未来的あるいは電脳的な世界観のゲームに感じるのではないでしょうか。フォントの選び方一つでこのように印象を大きく変えることができます。実際のロゴ制作ではこれに装飾などを施すことでさらに世界観などを表現することができるでしょう。

さて、フォントはそのデザインの特徴によって大まかに分類することが可能です。まずは日本語フォントの分類から紹介していきます。

明朝体

横線に対して縦線が太く、止めやはねの部分に**ウロコ**と呼ばれる装飾が施されていることが特徴です。
楷書から派生した書体であり、可読性が高く本の本文などに非常によく利用されます。
ただし、Webなどスクリーン上に表示する場合は解像度の関係により字が潰れてしまいがちです。
毛筆が元となっているため、「優雅」「和風」といった印象を与えます。

ゴシック体

横線と縦線の太さがほぼ同じで、明朝体とは違いウロコはほとんどありません。
そのデザインのシンプルさにより視認性が高く、見出しやタイトルなどによく使われます。
また、字を小さくしても潰れにくいためスクリーン上での本文フォントとしてもよく利用されます。

太さによって印象が大きく変わることも特徴で、細いと「モダン」、太いと「力強い」といった印象を与えます。

筆書体

毛筆が元となった書体です。

非常にデザイン的で、読ませるというよりはロゴなど見せる目的で利用することととなるでしょう。

これ以上なく和風な雰囲気を表現できますが、同時に力強さや格式高さを感じることが多いため必ずしも和風な雰囲気に合致するとは限らない点に注意してください。

デザイン書体

書体のデザインを通してイメージや世界観をより伝えることを目的として作られた書体です。

こちらも読ませるというよりは見せる目的で利用されることが多いです。(例外もあります)

エレガントやホラー、スタイリッシュなど様々な雰囲気を持ったデザイン書体が数多く存在します。

英字フォントは主に以下のように分類できます。英字フォントにもデザイン書体が存在しますが、デザイン書体についてては日本語フォントで紹介したため省かせていただきます。

Serif (セリフ体・ローマン体)

文字の線の端っこに**セリフ**と呼ばれる小さな飾りがついていることが特徴の書体です。

明朝体と類似した特徴を持ちます。

Sans-serif (サンセリフ体)

セリフがついていないことが特徴の書体です。「Sans-」はフランス語で「無い」を意味します。

ゴシック体と類似した特徴を持ちます。

Script (スクリプト体)

筆記体が元になったフォントで、流れるようなエレガントなデザインが特徴です。

日本人にとっては読むことは非常に難しく、デザインのワンポイントとして装飾的に使うことになるでしょう。

おしゃれで高級感のある雰囲気を出すことができます。

また、フォントはその頒布形態によって主に2つに分けることができます。**フリーフォント**と**有償フォント**です。前者は無償で頒布されているフォントを指し、反対に後者は有償で頒布されているフォントを指します。

フリーフォントは当然のことながら無償で利用することができることがメリットで、無料でも数多くの優秀なフリーフォントが存在します。反面、日本語フォントではサポートされていない漢字があったり規約にあいまいな部分があっ

たりと少し注意しなければならない点があるのも事実です。また、商用利用が不可となっているものも多く存在するため「商用」の定義(広告収入も含めて1円も取ってはいけないのか、同人程度ならOKなのかなど)も含めてライセンスをしっかり確認することが重要です。

　有償フォントはその名の通り利用するのにお金がかかるフォントで、ほとんどはフォントのメーカー企業が制作しています。企業が制作しているので非常に高品質でありサポートされている漢字も幅広いです。利用方法によってライセンスが細かく決められてることがほとんどのため、利用する場合はライセンスをしっかり確認しましょう。

　さて、ここでは有償フォントを提供しているサイトのうち定期・APゲーム制作をはじめとした同人活動でも使いやすい安価で有償フォントを利用できるサイトを紹介しておきます。

　一つはAdobe Fontsです。Adobe FontsはAdobe Creative Cloudというサブスクリプションサービスに付帯したサービスで、Creative Cloudを何か契約していれば追加料金なしでAdobe Fontsの豊富なフォントを利用することができ、そのフォント数は2020年4月現在15,000以上にも及びます。もちろん日本語フォントも数多く存在します。

　Creative Cloudの最安プランは980円/月(フォトプラン 個人向け、2020年4月現在)のため利用したいフォントがあれば契約してみるのも悪くないでしょう。Adobe Fontsは以下のURLよりアクセスできます。

　Adobe Fonts
　https://fonts.adobe.com/

　もう一つはmojimoです。mojimoはフォントワークスというフォント制作会社が提供しているサブスクリプション型サービスで、漫画用やゲーム制作用、かわいいフォントやきれいなフォントなど特定の用途や雰囲気ごとにフォントをまとめたフォントパックを年契約で利用することができるサービスです。

　用途が少し制限される代わりに従来では考えられないほど低価格で非常に高品質なフォントが入手できるようになっており、例えばゲーム制作向けのフォントパックmojimo-gameでは有名なゲームでも採用されているフォントがいくつか入って4800円(税別)/年で利用することができます。

　mojimoは以下のURLよりアクセスできます。なお、mojimoのライセンスはフォントパックごとに多少異なるのでライセンスをしっかり確認するようにしましょう。

　mojimo - ちょうどいい文字を、ちょうどいい価格で
　https://mojimo.jp/

　次にフォントをきれいに表示するためのワンポイントアドバイスを紹介します。フォントはただ書き並べればそれで終わりではありません。字体そのものの印象の他にもう一つ印象を左右する要素として**字間**というものがあります。字間はその名の通り字の間隔のことで、どれぐらい字間を空けるかによって印象が変わります。以下は同じ文章、同じフォントで字間を変えた例です。

信実とは、決して空虚な妄想ではなかった。

信 実 と は、決 し て 空 虚 な 妄 想 で は な か っ た。

僅かな違いではありますが、前者は普通に読み進めるための文章のように感じるのに対し、後者はよりキャッチコピーの文章のような印象に感じるのではないでしょうか。字間をうまく使うことで味わい深い文章を表現することができます。ただし開けすぎると間延びした印象となってしまうため適切な字間を見極めることが重要です。

字間で気を配るべき点がもう一つあります。以下の単語を見てください。

AVOID

「AVOID」という英単語ですが、AとVの間が広がってしまっていて見方によっては「A VOID」というようにも感じられてしまうのではないでしょうか。実は上のAVOIDという単語は字間をいじっているわけではなく、字と字の間はすべて同じ字間になっています。にも関わらず字間がおかしく見えるのはAは右下の部分が文字の端っこ、Vは文字の左上の部分が文字の端っことなっており他の文字に比べて最初から見た目の字間が開いてしまっているためです。

これを修正するためには、1文字1文字字間を変えて綺麗に見えるように調整します。このように調整を行うことを**カーニング**といいます。上の例をカーニングすると以下のような感じになるでしょう。

AVOID

カーニングを常に意識して字間を調整するのは難しいですが、少なくともロゴなどでは意識しておくとデザインがもっと良くなるでしょう。

また、ロゴデザインなどで日本語のタイトルロゴを作る際に漢字はやや大きく、ひらがなはやや小さくするとバランスがよくなり見た目に動きも出てより印象的なロゴにすることができます。文字の大きさも意識してロゴをデザインするとよりよいデザインになるかもしれません。

葉桜と魔笛

5.2.5 Webフォント

Web上のテキストにおいては環境によってシステムにインストールされているフォント(**システムフォント**)は大きく異なるため画像以外の方法で個性的なフォントを利用することは難しく、画像を利用する場合でも全体で利用すると通信容量が肥大化してしまいますし、動的な内容はそもそも表示できません。

そこで利用されるのが**Webフォント**です。Webフォントを利用する場合、表示する際にサーバーからフォントデータをダウンロードしてそのダウンロードしたフォントを使って文字を表示します。これによりWebフォントに対応している環境であればWindowsやMac、スマートフォンなどの環境に関わらず同じフォントを表示することができます。

注意点としてダウンロードという方式を取る以上どうしても大きなファイルだとその分フォントの反映が遅れてしまいます。特にアルファベットなどに比べて非常に文字数の多い日本語フォントでこの問題は顕著なので注意する必要があるでしょう。

Webフォントを配信している有名なサービスに**Google Fonts**というものがあります。多彩な英字Webフォントを取り揃えており、サイトに指定のコードを貼り付けるだけで無料で利用することができます。数は少ないですが日本語のWebフォントも取り扱っています。Google Fontsは以下のURLからアクセスすることができます。

Google Fonts
https://fonts.google.com/

また、Adobe Cleative Cloudを契約している方であればAdobe FontsのフォントをWebフォントとして利用することもできます。こちらの機能についても追加料金は必要ありません。利用する方法は以下のドキュメントから参照することができます。

Webサイトへのフォントの追加
https://helpx.adobe.com/jp/fonts/using/add-fonts-website.html

WebフォントにはUI表示用にアイコンをまとめたフォントも存在します。そのようなWebフォントのことを**Webアイコンフォント**と呼びます。主要なWebアイコンフォントは**FontAwesome**と**Google Material Icons**の2つで、どちらも無料で利用することができます（ただしFontAwesomeは有料限定のアイコンあり）。

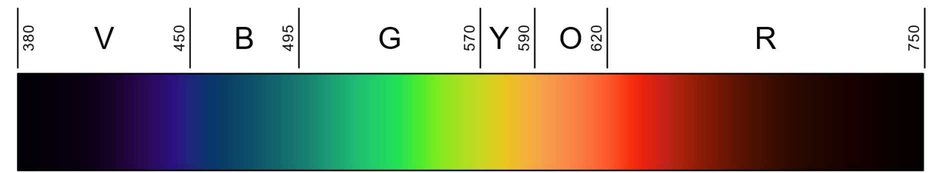
デザインの違いや対応しているアイコンの違いは若干ありますが、どちらもWebで利用されるアイコンが十分以上にまとまっているため利用する場合はお好みで選ぶと良いでしょう。それぞれ以下のサイトからチェックすることができます。

FontAwesome
https://fontawesome.com/

Google Material Icons
https://material.io/resources/icons/

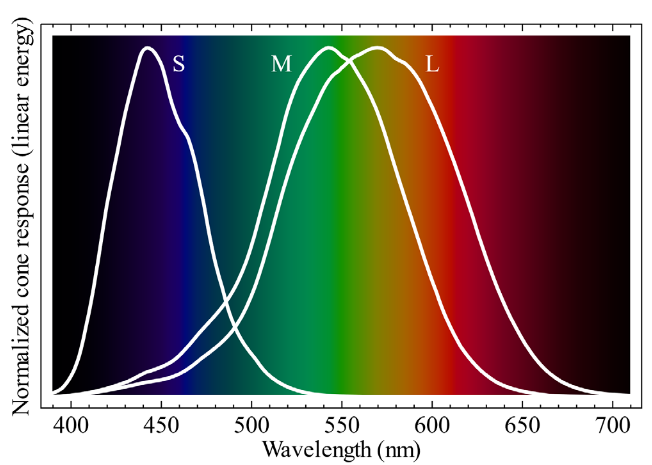
5.2.6 色

色について学ぶにあたってまずは光とか何なのかを知っておきましょう。光（可視光線）とは電磁波のうち人間の目に見えるものをさします。光を構成する**光子**は波であり粒子でもあるという性質をもっており、エネルギーが高いほど波長が短く（青紫っぽく）なりエネルギーが低いほど波長が長く（赤っぽく）なります。



人間の目は光を感じ取るために**錐体細胞**と**桿体細胞**という2種類の視細胞を持っています。錐体細胞は光の波長に対して反応する細胞であり、長波長（赤色光）に反応する**L錐体**、中波長（緑色光）に反応する**M錐体**、短波長（青色光）に反応する**S錐体**の3種類が存在します。それぞれ反応する波長（Long、Middle、Short）の頭文字を取って命名されています。桿体細胞は色覚にほぼ関与しないものの、錐体細胞より感度が高く暗いところでも光を認識することができます。暗いところで形は分かっても色がよくわからなくなるといった現象は日常生活の中でよくありますが、暗所で反応できるのが桿体細胞だけであり桿体細胞は光の強さはわかっても色がわからないためにこのような現象が起こります。

さて、LMSそれぞれの錐体細胞は光の波長に対して以下のような反応を示します。グラフが高いほど高い反応を示しているということです。



人間の目、正確には脳はこれらの錐体細胞の反応量のバランスにより色覚を行います。例えば光線の波長が470nmであればS錐体が強く反応しM、L錐体はほとんど反応しません。これにより脳はこれが青色であると認識します。また580nm付近であればS錐体はほとんど反応せず、L錐体は強く反応し、M錐体はL錐体より少し劣るものの強めに反応します。これによって脳はこれが黄色であると認識します。

これを利用したものが**光の三原色**です。L錐体、M錐体、S錐体にそれぞれ特異に反応を与えることができる赤、緑、青を原色とし、それを組み合わせることによって脳に様々な色を認識させることができます。例えば黄色を認識させたい場合、強い赤色光とやや強い緑色光を目に照射し青色光は照射しないようにします。これによりS錐体はほとんど反応せず、L錐体が強く反応し、M錐体がやや強く反応するという状況が生み出され、脳はこれが黄色光であると誤認します。

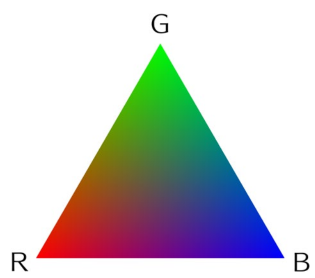
様々な色の光を際限なく混ぜ合わせる、もしくは光の三原色全てを混ぜ合わせると最終的に錐体細胞全てが強く反応するようになり、白く見えるようになります。自然界において白色光はほぼ全ての可視光線の波長を持った光

で錐体細胞全てを強く反応させるものであり、その状況が再現されるためにこれを白であると認識するようになるのです。このように色を混ぜ合わせるとどんどん白に近づいていくような混色のことを**加法混色**と呼びます。

実際には混ぜ合わせずとも非常に細かな精度で色を配置することで色が混ぜ合わさったように見えます。これを利用したものがディスプレイであり、ディスプレイを拡大して見てみると赤緑青の小さな画素が敷き詰められています。このような混色のことを**並置加法混色**（並置混色）と呼びます。（なお、これを利用することで減法混色であるはずの絵の具などで加法混色を表現することも可能です。興味のある方は調べてみてください。点描画やグラビア印刷などはこれを利用しています。）

色の表現方法

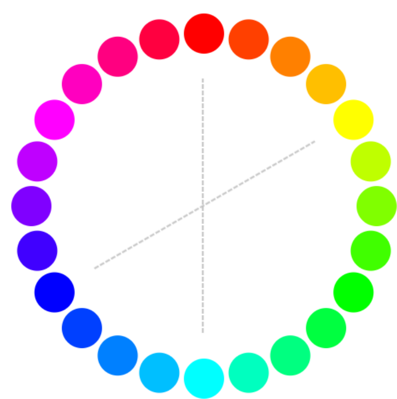
色の表現方法には主に2種類の方法があります。1つは光の三原色の強度によって色を表現する方法で、赤緑青の三原色の英語の頭文字を取って**RGBカラーモデル**（RGB）と呼ばれます。コンピューターで利用するときはこの値をそのまま画素の光のレベルに当てはめることができるのでコンピューターなどの分野ではよく使われます（CSSのカラーコードなど）。反面、人間にとっては直感的には分かりづらいものであり色について考える際にはあまりおすすめできない表現方法でもあります。



もう1つは**色相**（Hue）、**彩度**（Saturation）、**明度**（Value）の3つの要素によって色を表現する方法で、**HSVカラーモデル**（HSV）などがこれにあたります。色相でどんな色味なのか、彩度でその色の色鮮やかさ、明度でその色の明るさを指定します。デジタルでイラストを描いている人には親しみ深いカラーモデルかもしれません。人間にとって分かりやすいカラーモデルであり、色について考えるときは通常これもしくはこれに近いカラーモデルを使います。

色相

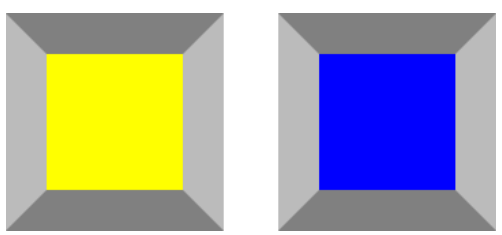
色相について考えるときには**色相環**という色相を輪の形に配置したものを用います。以下の例はHSV色相環を24等分したものです。（なおHSVなのか、マンセルなのか、PCCSなのかなどカラーシステムの違いでどのように並ぶのかは多少変わることがあります。）



色相環上で反対同士の色のことを**補色**(反対色)と呼びます。例えば赤の補色は水色、黄色の補色は青色になります。(分かりやすいように画像に補助線を引いてあります。)また、色相環上で隣同士もしくは近くに存在する色のことを**類似色**と呼びます。

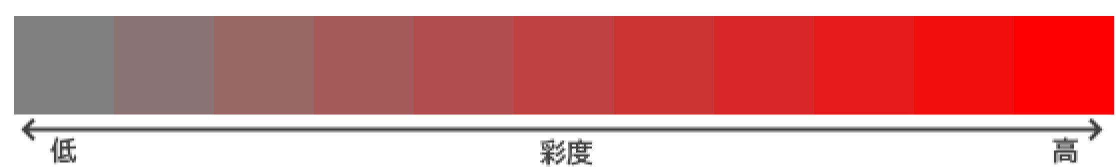
色のうち赤〜黄色のあたたかかみを感じる色のことを**暖色**、青緑〜青紫のつめたさを感じる色のことを**寒色**と呼びます。紫や緑など暖色でも寒色でもない色のことは**中性色**と呼びます。暖色は交感神経に作用するため人を興奮させたり食欲を高めたり時間を長く感じさせたりする色でもあります。それに対し、寒色は副交感神経に作用し興奮を鎮めたり食欲を減退させたり時間を短く感じさせたりします。

また、同じ距離・同じ大きさ・同じ形で色が違うだけの図形を見た場合でもその色によってその図形が近くに見えたり遠くに見えたりします。近くに見える色のことを**進出色**、遠くに見える色のことを**後退色**と呼びます。一般に暖色は進出色、寒色は後退色です。下の例では黄色が手前に青が奥に感じる方が多いです。



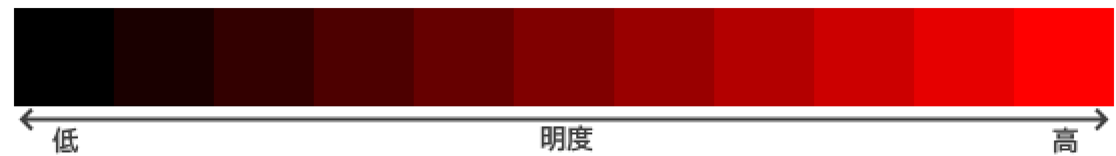
彩度

色の鮮やかさを決める要素です。彩度が高いほど鮮やかな色で、低いほどくすんだ色になります。彩度が最も高い色のことを**純色**といいます。また、彩度がない色(白、グレー、黒など)のことを**無彩色**といい、それに対し彩度がある色のことを**有彩色**といいます。

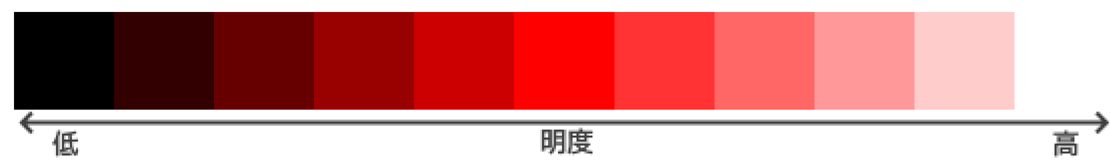


明度

色の明るさを決める要素です。明度が高くなるほど明るく、低いほど黒に近くなります。純色とその彩度を変更した色を最も明るいとするHSVカラーモデルでは以下のようになります。(HSVカラーモデルでは白は彩度0%、明度100%で表現される。)



白を最も明度の高い色とする**HSLカラーモデル**などでは以下のようになります。(HSLカラーモデルでは純色は明度50%で表現される。)

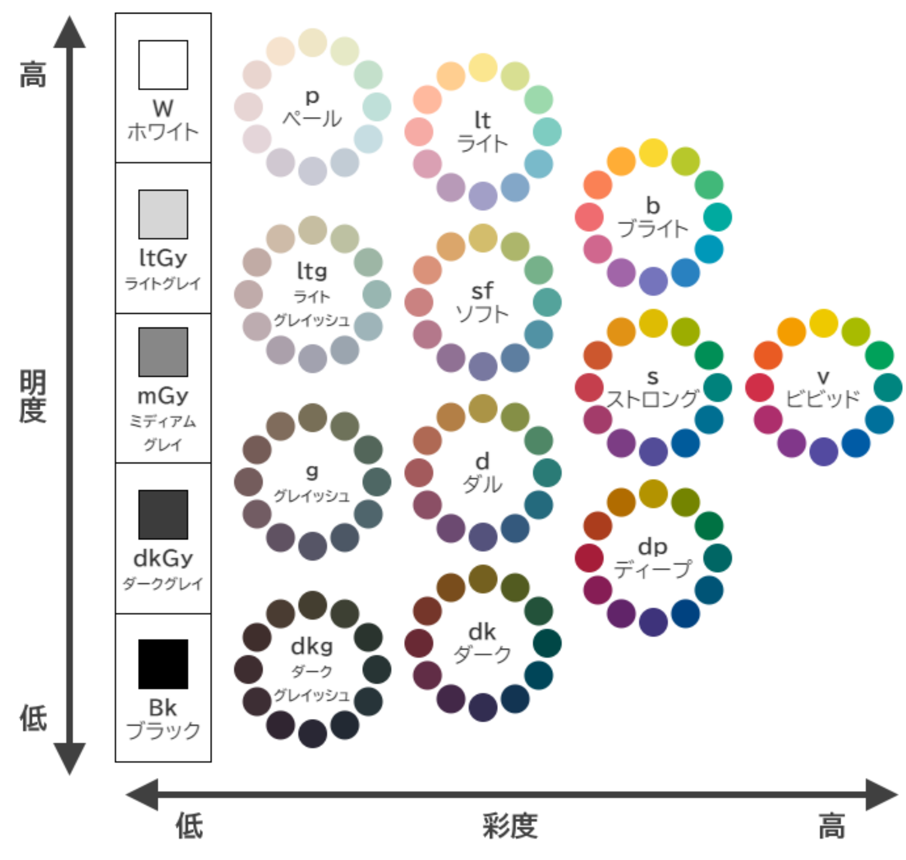


また、同じ距離・同じ大きさ・同じ形で色が違うだけの図形を見た場合でもその色によってその図形が大きく感じられたり小さく感じられたりします。大きく見える色のことを**膨張色**といい、暖色や明度の高い色は膨張色になります。それに対して小さく見える色のことを**収縮色**といい、寒色や明度の低い色は収縮色になります。下の画像の例では実際のサイズは一緒であるにも関わらず左のほうがやや大きく感じられるはずです。碁石などではこの効果に対応するために白を若干小さく作っているものもあります。

また、人間は明度が高い色ほど軽く、低い色であるほど重く感じます。見た目で感じる重さは真っ白の箱を100とすると薄茶色で約120、黒色では約190になります。最近の引っ越し用ダンボールには白色がベースのものが増えていますが、この心理的効果を利用したものです。

PCCS

色を体系立てて考えるときに役立つのが**PCCS**(Practical Color Co-ordinate System:日本色研配色体系)というカラーシステムです。PCCSの特徴として、色を色相と**トーン**に分けて考えるというものがあります。トーンとは彩度と明度の度合いによって色をまとめたものです。PCCSではトーンを以下のように区分しています。



有彩色トーンのうちブライト・ライト・ペールなどの純色に白を混ぜて作られる色のことを**明清色**と呼びます。逆に、ディープ・ダーク・ダークグレイッシュなどの純色に黒を混ぜて作られる色のことを**暗清色**と呼びます。そのどちらでもないストロング・ソフト・ダル・ライトグレイッシュ・グレイッシュなどの純色にグレーを混ぜて作られる色のことを**中間色**と呼びます。文化的背景によっても異なりますがそれぞれのトーンは以下のような印象を与えます。

ビビッド	鮮やかな、さえた、派手な、生き生きした、目立つ
ブライト	明るい、健康的な、陽気な、華やかな
ストロング	強い、くどい、動的な、情熱的な
ディープ	深い、濃い、充実した、伝統的な、和風の
ライト	浅い、澄んだ、子どもっぽい、爽やかな、楽しい
ソフト	柔らかい、穏やか、優しい、ぼんやりした
ダル	鈍い、穏やか、くすんだ、自然
ダーク	暗い、大人っぽい、渋い、丈夫な、円熟した
ペール	薄い、軽い、あっさりした、女性的、弱い、若々しい、かわいい、優しい
ライトグレイッシュ	明るい灰みの、落ち着いた、渋い、おとなしい
グレイッシュ	濁った、地味な、渋い、落ち着いた、都会的
ダークグレイッシュ	シック、陰気な、重い、固い、男性的

PCCSの色のRGB値については以下のサイトが参考になります。ただし、モニターによって表示される色は異なるため参考程度に見るようにしましょう。

PCCS相互変換表 - garakuta.net
http://www.garakuta.net/color/pccs/matrix.html

色相の印象

トーンだけではなく、色相にも印象効果があります。主な色の印象効果は以下のとおりです。こちらも文化的背景によって変わることがあります。

色	色名	印象
	赤	活動的、情熱、勇敢、力、積極的、外向性、興奮、勇気、活気、注意、革命、力、衝動
	橙	活力、行動、冒険、安定、陽気、賑やか、成功、慈善、好意的、美味しそう、親しみ、友情
	黄	楽しい、元気、明朗、快活、好奇心、希望、エネルギー、ユーモア、想像力、危険、幼さ
	黄緑	若々しい、フレッシュ、穏やか、順応、希望、勇気、知恵、調和、爽やか
	緑	安定、自然、静か、平和、秩序、健康、調和、信頼、安心、中立、平凡、苦い
	青緑	カジュアル、洗練、冷静、治癒
	青	知恵、知性、神秘、冷静、信頼、誠実、安定、清潔、落ち着き、悲しい
	紺	論理、知識、専門的、革新的、法則
	紫	気品、威厳、上品、高貴、気品、神秘、幻想的、直感、忍耐力、想像力、エキゾチック、憂鬱
	桃	可憐、女性的、幸福、愛情、自愛、好意、感謝、優しい、甘さ
	茶	現実、大地、忍耐力、平穏、親和、信頼、堅実、知性、調和、質素、地味
	灰	人工的、産業的、保守的、落ち着き、安定、迷い
	白	高貴、洗練、純粋、単純、清潔、平和、明快、素直、完璧、無垢、神聖、真実、冷たい、空虚
	黒	高貴、洗練、神秘、不安、優雅、おしゃれ、クール、知的、孤独、拒絶、不安、恐怖、終わり、死

避けるべき色使い

デザインにおいて完全な黒(#000000)の使用は避けるべきであると言われています。完全な黒は自然界にほとんど存在しない色のため、無意識的にそちらに意識が行ってしまい他の色を圧倒してしまうためです。そんなことをいいつつもこの文章の文字色は黒じゃないかと思われるかもしれませんが、この文章の色は#212121でありかなり黒に近いグレーになっています。完全な黒ではありません。私達が普段目にする色で黒だと思うようなものはほとんどの場合黒に近いグレーです。Twitterのデフォルトの文字色やGoogleの検索ページの文字色、Visual Studio Codeの背景色などもそうなっています。

また、黒背景白文字のデザインはよく見ますが黒背景の利用はあまり好ましいものではありません。人間の目は無意識的にサッケードという周辺視野を確認する動作を行っています。黒背景の暗い画面に慣れた目がサッケードによって蛍光灯で明るく照らされた壁を見てしまうと目がチカチカしたり疲労したりしてしまいます。定期・APゲームをプレイするユーザーの環境は明るい屋内や日中の野外のような周辺視野が明るい環境がほとんどだと思われるので、それに合わせて明るい色の背景色を使うようにしましょう。(逆に言えば映画館など周辺視野が暗いことが保証されている環境では黒背景は有用です。)なお黒背景白文字が全て駄目というわけではありません。黒背

景は何かを読ませるのにはあまり適していませんが画像などを魅せたいときには非常に有用です。

なお表現したい世界観の関係などでどうしても黒背景を使いたく、明るい背景での表現はどうやっても無理だといった場合、背景を黒っぽく見えるものの若干明るさのある色にするといいでしょう。黒寄りのグレーや紺色などです。このとき、文字色が真っ白（#FFFFFF）だと明度差で目がチカチカしてしまうので文字色は真っ白ではなく若干暗い色にしましょう。また、人間が感じる明るさというのは相対的なものなのでグラデーションや他要素の色などを上手く使うことで実際はある程度明るい色だとしても暗く感じさせることができます。

もし背景色が暗い感じになっている商業ゲームなどをプレイすることがあれば今度プレイするときは背景がどうなっているのか注意して見てみましょう。おそらく大抵の場合は暗いと感じていたものの実際はある程度明度があったり、グラデーションで暗く見せているだけだったりして全体としての明度は実はそんなに低くはないはずです。このようなデザインの組み方は背景色が暗いデザインの定期・APゲームを組むときに参考になるかもしれません。（とはいえ小手先のテクニックに頼るよりは最初から明るい背景にしたほうがよいので、どうしても暗い背景でないとダメといった場合でなければ明るい背景にしましょう。）

5.2.7　配色

『色』では単一の色に関する理論などを扱いましたがここでは複数の色のとりあわせについての方法論である**色彩調和論**について扱います。色彩調和論はシュブルール、ルード、オストワルトなど様々な人々が独自の理論を展開しているのですが、ここではそれら先人の色彩調和の原理をアメリカの色彩学者であるディーン・ブリュースター・ジャッドがまとめたものである『4つの色彩調和論』について扱います。

秩序の原理

秩序の原理は「規則的に選ばれた色同士は調和する」という原理です。こちらに関しては実際に見たほうが早いでしょう。秩序の原理に基づく配色には以下のようなものがあります。

このように色相環から幾何学的に一定の法則に基づいて色を選ぶと調和しやすいです。それぞれの配色は以下のように色を選択しています。

アイデンティティ	同一の色相のみで構成される配色です。
アナロジー	色相環上で隣り合った色で構成される配色です。 調和が取りやすく、自然で柔らかい印象となります。
インターミディエイト	色相が60°〜105°ほど離れた色で構成される配色です。 変化が中途半端となるため不調和になりやすく難しい配色です。
オポーネント	色相が120°〜150°ほど離れた色で構成される配色です。 変化が大きいためいきいきとしたダイナミックな印象となります。
ダイアード	補色同士を取り合わせた配色です。 コントラストが強すぎるため境界面で目がちかちかしてしまいやすく、 使うには一工夫が必要になるでしょう。
トライアド	色相が120°離れた色同士を取り合わせた配色です。
テトラード	色相が90°離れた色同士を取り合わせた配色です。
ヘクサード	色相が60°離れた色同士を取り合わせた配色です。

その他にも秩序の原理に基づく配色はいくつかあるので調べてみるとよいでしょう。

なじみの原理

なじみの原理は「いつも見慣れている色の配列は調和している」という原理です。自然界では明るい色は黄色がかって、暗い色は青みがかって見えるという**ルードの色相の自然連鎖**などはこれにあたります。ルードの色相の自然連鎖に従った配色のことを**ナチュラル・ハーモニー**（ナチュラル配色）と呼びます。イラストを描いている人であれば影の色の色相を青に寄せると影がそれらしくなるというテクニックを学んだことがあるかもしれませんが、これはこの理屈に基づくものです。

左のグラデーションは明度だけを下げたもの、右のグラデーションは明度を下げつつ色相を青に寄せていったものです。右のほうがより自然な色合いに感じるのではないでしょうか。

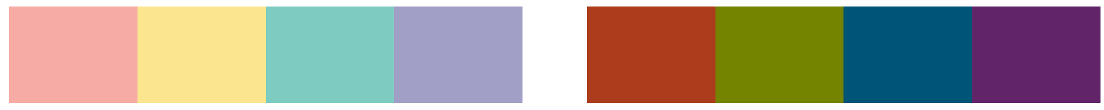
これを逆手に取った**コンプレックス・ハーモニー**（コンプレックス配色）というものもあります。ナチュラル・ハーモニーとは逆に明るい色を青みがからせて暗い色を黄色がからせるという配色で、上手く使うことができれば目を引いたり独創的な印象を与えることができたりします。身近なものでいうとチョコミントアイスなどはコンプレックス・ハーモニーを上手く使っています。アイス部分が明るい水色、チョコレート部分が暗い茶色になっておりコンプレックス・ハーモニーが成立しています。あの目を引く独特の印象はコンプレックス・ハーモニーによるものです。

類似性の原理

類似性の原理は「色の感じに何らかの共通性がある色同士は調和する」という原理です。よりわかりやすく言えば似た色相の色同士や似た明度の色同士やトーンが同じ色同士など似ている色は調和する、というものになります。

代表的なものに**ドミナントカラー配色**があります。ドミナントカラー配色は色相を統一、あるいは類似色相のみに色を絞り込みその中からトーンを選んで配色する方法になります。このとき明度差を大きく取ったものは**トーン・オン・トーン配色**といいます。ドミナントカラー配色を利用することで統一感が生まれ、色相のイメージをより強く与えることができます。

逆に使用するトーンを絞り込んでその中から色相を選んで配色する方法を**トーン・イン・トーン配色**もあります。この配色ではトーンのイメージをより強く与えることができ、にぎやかな印象を出しつつも統一感のある印象を与えることもできます。

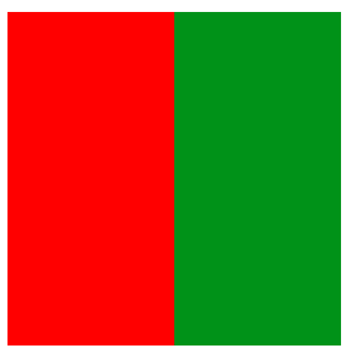


その他にも類似性の原理に基づく配色技法はいくつかあるので、調べてみるとよいでしょう。

明瞭性の原理

明瞭性の原理は「はっきりとした対比のある色同士は調和する」という原理です。具体的な例でいうとフランス国旗やドイツ国旗のような配色はこの原理に基づいたデザインとなっており、このようなはっきりと別れた三色配色のことを**トリコロール配色**と呼びます。トリコロール配色には彩度の高い色を使うとコントラスト感が明瞭になりより印象的になります。

また、二色による同様の配色のことを**ピコロール配色**といいます。さらにピコロール配色のうち色相、彩度、明度、トーンのいずれかが反対の色同士を取り合わせたものを**コントラスト配色**といいます。ただし色相を反対にした(補色)コントラスト配色は色どうしの境界線の部分で**ハレーション**という現象を起こしてしまうことがあります。次の画像を見てください。



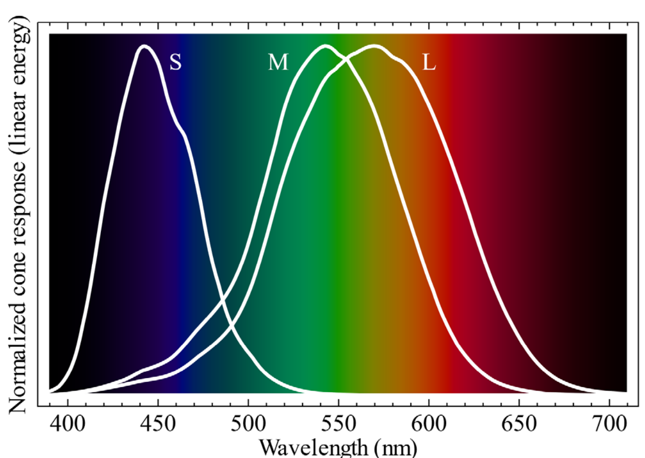
　境界線付近のコントラストが強すぎてギラギラとした印象になってしまっているのではないでしょうか。それぞれの色が互いの色をかき消そうとして境目部分がグレーに見えてしまうためこのようなことが起こります。白や黒などの色を境界線に挟むことでこれを解決することができます。これを**セパレーション効果**といいます。

5.2.8　カラーユニバーサルデザイン

　色の見え方が一般的な人とは異なることを**色覚異常**といいます。色覚多様性や色弱、色盲とも呼ばれます。色覚異常はまれなものではなく、日本人においては男性の約5%、女性の約0.2%が先天的に色覚異常を持っていると言われています。もし定期・APゲームの男性参加者が500人、女性参加者が500人いた場合、参加者のうちおよそ26人が色覚異常を持っているということになります。

　色覚異常を発現させる遺伝子はX染色体に存在しており、X染色体を2つ持っていてかつ片方が色覚異常を発現させる遺伝子を持っていない場合には発現しないという特性があります（伴性劣性遺伝）。この特性によりXXの性染色体を持つ女性にはあまり色覚異常は発現せず、XYの性染色体を持つ男性には多く発現します。（発現しないというだけで日本人女性のうち約10%は色覚異常の保因者です。）

　色覚異常者は錐体細胞が無かったり少なかったり、あるいは知覚する波長がずれていたりするために色の見え方が一般的な人と異なります。一般的な人の色覚タイプを**C型色覚**（正常色覚）といいます。それに対してL錐体に異常が存在する色覚タイプを**P型色覚**（1型色覚）、M錐体に異常が存在する色覚タイプを**D型色覚**（2型色覚）といい、色覚異常者のほとんどはこのどちらかのタイプです。L錐体とM錐体は反応する光の波長が近いのでP型とD型は似たような色覚特性を持っており、どちらも赤〜緑の判別が困難になります。



他にもS錐体に異常が存在する**T型色覚**(3型色覚)、錐体のうち2つもしくは全てに異常がある**A型色覚**(1色覚)も存在しますが数万人に1人レベルであり非常にまれです。

色覚異常を持つ方には左上の画像が例のように見えます。(モニターや印刷の発色状況による差や個人差があるため正確なものではなく、色覚診断に使えるようなものではありません。あくまで参考です。)

▲C型色覚

▲P型色覚

▲D型色覚

▲T型色覚

P型・D型色覚では以下のような色の識別が非常に困難になります。

例えば以下の画像は正常色覚者にはMerry Xmasと書いてあるように見えますが、D型の色覚異常を持つ方には右の画像のように見え、なにが書いてあるのか非常にわかりづらくなってしまいます。

Merry Xmas　　Merry Xmas

色覚異常の方も見やすいように配慮された色使いやデザインのことを**カラーユニバーサルデザイン**（CUD）といいます。カラーユニバーサルデザイン向けに配色セットやデザインのガイドラインが公開されているため、ここではそれを紹介しておきます。色覚異常を持つ方にも正常色覚の方にも見やすい色の取り合わせとなっているため有効活用すればよりよいデザインとなるでしょう。

カラーユニバーサルデザイン推奨配色セット
https://jfly.uni-koeln.de/colorset/

東京都カラーユニバーサルデザインガイドライン
https://www.fukushihoken.metro.tokyo.lg.jp/kiban/machizukuri/kanren/color.files/colorudguideline.pdf

5.2.9　論理ピクセル / 物理ピクセル

テクノロジーが進みスマートフォンやタブレットなどが高解像度化するにつれて1pxのサイズはどんどん小さくなっていき、1pxを1pxのまま表示してしまうと非常に画面が見づらいというような問題が起きてしまいました。そこで生まれたのが**CSSピクセル**という概念です。例えば実際の画面の2px×2pxの領域を1px扱いで表示するようにすることで1pxが小さくなりすぎて画面が見づらくなる問題に対処したりしています。実際のデバイスの画面の解像度のことを**デバイスピクセル**といい、CSSピクセルとデバイスピクセルの比のことを**デバイスピクセル比**といいます。先程の例ではデバイスピクセル比は2になります。簡単に言ってしまうと、デバイスピクセル比2では200%拡大、3では300%拡大、4では400%拡大……というように拡大表示されています。多くの携帯用端末ではデバイスピクセル比は2以上になっていて、例えばiPhone XであればDP比は3、iPad（第7世代）であればDP比は2、Xperia ZであればDP比は3となっています。

文字や枠線などなら拡大表示しても問題ないのですが、画像となると少し問題が出てきます。例えば60×60のアイコン枠に60×60pxの画像を設定しデバイスピクセル比2の端末で表示する場合、200%に拡大表示されてしまうのでアイコンが滲んでしまいます。

これに対応するためにはアイコンサイズ120×120や180×180などに対応する必要があるのですが、そうすると今度は通信負荷が高まってしまいます。デバイスピクセルが高い端末は多くの場合スマートフォンであり、通信量の制限があることが多いためあまりアイコンサイズの許容範囲を高めすぎるのもよくありません。アイコンサイズを決める際はそのあたりのバランスを取りながら設定するといいでしょう。

5.2.10　変数の命名規則

JavaScriptの変数の命名規則には主に2種類が存在します。1つは**キャメルケース**です。キャメルケースでは「EnemyCharacter」や「getEnemyCharacters」というように単語の区切りを大文字で表記します。前者のように

1単語目も含めて大文字にする命名規則のことを**アッパーキャメルケース**(パスカルケース)といいます。それに対し、1単語目は大文字にしない命名規則のことを**ローワーキャメルケース**(単にキャメルケースとも)と呼びます。JavaScriptではクラス名にアッパーキャメルケースを用い、変数名やプロパティ名、メソッド名などにはローワーキャメルケースを用いるのが一般的です。

もう1つの方法は正式な名称はないと思われるのですが全てを大文字にして単語の区切りを_(アンダースコア)で表現する方法で、「CHARACTERS_MAX」などというようにします。JavaScriptでは不変な値(定数)にこの命名規則を用いるのが一般的です。

また、メソッド名をつける際はより分かりやすくなるように以下のような命名をすることが多いです。

単語	例	意味
get	getEnemyCharacters	対象を取得する
set	setEnemyCharacters	対象の値をセットする
add	addEnemyCharacter	対象を追加する、もしくは加算する
remove	removeCharacter	対象を取り除く
find	findCharacter	対象を検索する
create	createCharacter	対象を作成する
to	toInteger	対象へと変換する
apply	applyChanges	対象を適用する
clear	clearMessage	対象を消去する
reset	resetBattle	対象を初期状態に戻す
post	postCharacter	対象のデータを送信する
is	isAlive	対象が特定の状態かどうか
has	hasMagicItem	対象が特定のプロパティや値をもっているかどうか
can	canAttack	対象が特定の動作を可能かどうか
on	onTurnStart	特定のイベントが起こったとき
before	beforeTurnStart	特定のイベントが起こる前
after	afterTurnEnd	特定のイベントが起こった後

命名規則には他にもHTMLやCSSなどで用いられる「character-list」というように-(ハイフン)を使って単語を区切る**ケバブケース**(チェインケース)や、PHPやRubyなどで用いられる「character_list」というように_(アンダースコア)を使って単語を区切る**スネークケース**というものもあります。

その他にも意味はないものの都合上名前が必要な変数に使われる**メタ構文変数**という命名もあります。本書にも何度か出てきましたが、メタ構文変数には「foobar」「hoge」「fuga」などがあります。他にも「foo」「bar」「baz」「piyo」などもあります。

5.2.11 CSSの設計手法

CSSはいろいろな作り方ができる分、さまざまな設計手法が存在します。これらは好みによって採用するか決めましょう。続く解説では有名な設計手法を簡単に解説しています。ほぼさわりの部分しか解説していないので厳密

に採用したい場合はそれぞれ調べてみてください。

OOCSS

OOCSS(Object-Oriented CSS)ではCSSをパーツの組み合わせとして記述します。例えば赤色で100px平方なボックスと青色で100px平方なボックスを作るとしましょう。素直に書くなら以下のようになります。

```
<div class="redbox"></div>
<div class="bluebox"></div>

<style>
  .redbox {
    width: 100px;
    height: 100px;
    background-color: red;
  }

  .bluebox {
    width: 100px;
    height: 100px;
    background-color: blue;
  }
</style>
```

この例だとwidthとheightの宣言が2回必要で冗長です。これをOOCSSでは以下のように表現します。赤色や青色といったパーツと、100px平方なボックスであるというパーツを分けるのです。

```
<div class="box red"></div>
<div class="box blue"></div>

<style>
  .box {
    width: 100px;
    height: 100px;
  }

  .red {
    background-color: red;
  }

  .blue {
    background-color: blue;
  }
</style>
```

これにより例えば色は変えなくてもいいけど全体的にサイズをちょっと小さくしたい、といった場合などに変更する必要のある箇所が少なくなり、メンテナンスが容易になります。

BEM

BEMはその要素がどのブロックに属するのか(Block)、そのブロックの中でどのような要素なのか(Element)、またその例外パターン(Modifier)によって要素を指定し「block--element__modifier」というようにクラス名をつけます。クラスを見ただけでこれが何であるのかが分かりやすいのが特徴ですが、この独特な命名規則(MindBEMding)などにより敬遠されることもあります。例外ごとにBEMのスタイルを一つ一つ書いていてはきりがないので、BEMを採用する場合SCSSなどプリプロセッサの利用はほぼ前提となるでしょう。

例えばキャラクターの死亡がありえる定期・APゲームでキャラクターリストを作りたいとしましょう。基本的にはキャラクター名は黒字で表示されますが死亡している場合は灰色でキャラクター名を表示するとします。BEMではこれを以下のように表現します。

```html
<ul class="character-list">
  <li class="character-list--character-name">
    太郎
  </li>
  <li class="character-list--character-name__dead">
    花子 (死亡)
  </li>
</ul>
```

```scss
.character-list {
  &--character-name {
    color: black;

    &__dead {
      color: gray;
    }
  }
}
```

SMACSS

SMACSS(Scalable and Modular Architecture for CSS)ではCSSのルールを以下の5種類に分けて考え、それぞれに考え方や記述ルールを取り決めます。

ベース	要素そのもののスタイル
レイアウト	ヘッダーやフッターなどセクションを分割するためのスタイル
モジュール	再利用可能なパーツのスタイル　見出しや吹き出し、キャラクターリストなど
状態(ステート)	マウスオーバーされている時など特定の状態におけるモジュールやレイアウトのスタイル
テーマ	見た目に影響する色や画像などのスタイル

SMACSSの公式による解説は以下からアクセスでき、「Get the Book」のPDFリンクから日本語訳を読むこと

が可能です。

Ja - Scalable and Modular Architecture for CSS
http://smacss.com/ja

5.2.12 TypeScript(AltJS)

JavaScriptは動的な型付けや比較的自由な構文が特徴で、プログラミングがはじめてという人でも非常に扱いやすい言語となっています。反面、コードが複雑化してしまったりこの変数はどんな型が入る変数なのかがわかりづらくなってメンテナンス性が低くなりがちです。

そのような問題を解決するために生まれたのが**AltJS**(Alternative JavaScript)です。AltJSにはいくつかの言語があるのですが、それぞれの言語の文法に従ってコードを書き、そのコードをJavaScriptに変換することによって利用されるというのが特徴です。

さて、AltJSの中でも最も人気のある言語が**TypeScript**です。TypeScriptはその名の通り「型を定義できるJavaScript」であり、JavaScriptの仕様を拡張したものとなっています。構文はJavaScriptとほぼ同じのためJavaScriptを知っていれば学習コストが低いのも人気の理由の一つでしょう。設計者がMicrosoftという大企業であるのも大きいです。TypeScriptを利用することで以下のようなメリットを享受できます。

- 変数に意図しない値が入ったときにエラーを表示できるためデバッグが容易になる
- より高機能なクラスを利用することができる
- 対応しているエディターでは型がおかしくなるコードを書いてしまったとき実行しなくてもエラーを表示できる

以下はTypeScriptでコードを記述した例です。JavaScriptとほとんど変わらないコードですが型を宣言できるようになっており、違う型の値を代入しようとするとエラーになります。

```
TypeScript
let language: string = "JavaScript";
language = "TypeScript";
console.log(`Hello, ${language} world!`);

language = 1; // 文字列型の変数に数値を代入しようとしているのでエラーになる
```

TypeScriptは以下のサイトから簡単に試すことが可能です。左のテキストエリアに実行したいTypeScriptのコードを入力し、上部メニューの「Run」をクリックすることで実行できます。試しに上記のコードを入力し、実行しなくても「language = 1;」のところでエラーが出ることを確認してみてください。

TypeScript Playground
https://www.typescriptlang.org/play/

実際にNode.jsなどでTypeScriptを利用する場合は設定等を行う必要がありますがそれに値するメリットがあ

るため利用できる環境では積極的に利用するとよいでしょう。TypeScriptを利用する方法は以下の記事が参考になります。

初心者がTypeScriptを始めるためのチュートリアル！｜侍エンジニア塾ブログ
https://www.sejuku.net/blog/82822

5.2.13 Webhook

WebhookとはWebアプリケーションなどで何らかのイベントが発生した際、外部サービスにHTTPなどで通知する仕組みのことを指します。オンラインコミュニケーションツールであり定期・APゲームのユーザーにも利用者の多い**Discord**などのサービスがWebhook APIを公開しています。

DiscordのWebhook APIに仕様に従ってHTTPリクエストを送るだけでユーザーに通知を送ることができ、これを利用することで例えば返信を受け取った際などに着信を受け取れるといったサービスを低コストで提供することが可能です。

『Stroll Green』という定期ゲームを運営されていたヒサギさんという方がこの機能の実装方法を解説されているので、ここではそれを紹介します。記事ではPHPを用いて解説されていますが、シンプルなコードで実現できるためNode.jsにも容易に移植できると思います。

Discordウェブフック連携を用いたチャット通知の実装｜ヒサギ｜note
https://note.com/_soraniwa/n/n073c674e2e19

なお、DiscordのWebhookに関するドキュメントは以下のURLよりアクセスできます。

Discord Developer Portal — Documentation — Webhook
https://discordapp.com/developers/docs/resources/webhook

5.2.14 正規表現

正規表現とは文字列を検証するためのパターンのことです。JavaScriptにおいてはスラッシュで囲まれた**正規表現リテラル**を使うことで正規表現を使用することができます。また、RegExpオブジェクトを呼び出すことでも使用できます。以下の2つはいずれも同じ正規表現を表しますが、前者の方がパフォーマンスがよいため正規表現のパターンが予め指定できないなど特別な理由がない限り前者を利用しましょう。

```JavaScript
const regexp1 = /ab+c/;
const regexp2 = new RegExp("ab+c");
```

ここではよく利用する正規表現のパターンの記法をいくつか紹介します。

.（ドット）

ドットは改行以外のどの1文字にもマッチします。例えば/r.n/であればrunやranやrenなどにマッチします。文字数が異なるためrnやrainにはマッチしません。

^（ハット）$（ドルマーク）

ハットは入力の先頭に、ドルマークは入力の末尾にマッチします。例えば/pen/であれば「This is a pen」にマッチしますが、/^pen/とした場合は「pen」は先頭に存在しないためマッチしなくなります。/pen$/であればマッチします。/^pen$/というように同時に指定することで「pen」に完全に一致する場合のみマッチさせることも可能です。

+（プラス）

プラスは直前の文字の繰り返しにマッチします。例えば/do+g/であれば「dog」や「doooog」にマッチさせることができます。1文字以上存在していることが条件になっており、「dg」にはマッチしません。

*（アスタリスク）

アスタリスクも直前の文字の繰り返しにマッチします。/do*g/であれば「dog」や「doooog」にマッチします。プラスとは違って「dg」のように直前の文字が存在しない場合もマッチします。

?（クエスチョンマーク）

クエスチョンマークを利用すると直前の文字が存在しない場合もマッチさせることができます。例えば/-?10/であれば「10」にも「-10」にもマッチさせることが可能です。

[xyz]

[]の中の文字のいずれか1文字にマッチします。例えば/r[au]n/であればrunやranにマッチしますが、renなどにはマッチしません。

[^xyz]

[]の中の文字以外のいずれか1文字にマッチします。例えば/r[^au]n/であればrenにはマッチしますがrun、ranにはマッチしません。

\（バックスラッシュ）

正規表現に用いられる文字をマッチさせる際に該当の文字の前に記述することで通常の文字としてマッチさせることができます。例えばMr.Taroにマッチさせる正規表現であれば/Mr\.Taro/というように表記します。フォントによっては¥で表示されることもあります。

正規表現のパターンの記法は非常に数が多く、ここで紹介した記法の他にも数々の記法が存在するためより詳しく知りたい場合は調べてみてください。以下のサイトに詳しく解説されています。

正規表現 - JavaScript | MDN

https://developer.mozilla.org/ja/docs/Web/JavaScript/Guide/Regular_Expressions

5.2.15 SELinux

SELinuxはLinux用のセキュリティソフトです。例えるなら自動車のシートベルトなどの安全装置のようなもので、シートベルトなどが事故そのものは防ぐことはできないものの事故が起きた際の被害を低減してくれるように、サーバーに不正に侵入されることは防げないものの侵入された際の被害を最低限に抑えてくれます。

私個人の意見としてはSELinuxは個人で開発しているような定期・APゲームのサーバーでは無効化していても構わないかと思います。(個人情報や金融情報などを扱う場合は使うべきです。)ですがやはり使ったほうがセキュリティ的によいのは確かなので、もしセキュリティにこだわりたいという方であれば難しいですが有効化に挑戦してみるといいでしょう。

ConoHa VPSではデフォルトでSELinuxは無効化されています。有効化するにはログインしてrootに昇格しまず「setenforce enforcing」を実行。次に「vi /etc/selinux/config」を実行し「SELINUX=disabled」となっているところを「SELINUX=enabled」に書き換えて保存します。これでSELinuxが有効になります。

実際にSELinuxが何をやっているのか、どうやって設定するかについてはそれだけで本が一冊書けてしまうので本書では取り扱いません。CentOS7の元となっているRed Hat Enterprise Linux 7のガイドにSELinuxの扱い方について詳しく解説しているものがあるので、ここではそれを紹介します。

SELinux ユーザーおよび管理者のガイド Red Hat Enterprise Linux 7 | Red Hat Customer Portal
https://access.redhat.com/documentation/ja-jp/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/index