



KtorとNuxt.jsで作る Webアプリケーション入門

crescent



Ktor と Nuxt.js で作る Web アプリケーション入門

chichi1091 著

2020-09-12 版 crescent 発行

本書について

対象読者

本書は対象読者として次のような方を想定しています。

- Web アプリケーションの開発を行ったことがある
- サーバサイド Kotlin や Nuxt.js に興味があり、Web アプリケーションの作成に挑戦してみたい方
- ハンズオン形式で Web アプリケーションの開発を進めていきたい方

本書では Kotlin や Nuxt.js の基本文法や使い方については多くは触れませんので、言語仕様については公式ドキュメントを参照していただけると幸いです。

本書の執筆環境と動作環境

執筆時点（2020.9.12）で LTS となるバージョンを採用するようにしています。本書では次の開発環境を前提にしております。

- Docker 19.03.12
- Docker Compose 1.26.2
- Java 11.0.8.j9-adpt
- Kotlin 1.3.72
- Node.js v12.18.3
- Yarn 1.19.2

環境構築については記載は行いませんが、Java や Kotlin、Node.js は次のツールを利用していますので参考にしてみてください。

- SDKMAN (<https://sdkman.io/>)
- nodebrew (<https://github.com/hokaccha/nodebrew>)

サンプルコード

本書のサンプルコードは GitHub に置いてあります。次のリポジトリからコードをダウンロードするか Git でチェックアウトしてご利用ください。

- ktor-todo-exmaple (<https://github.com/chichi1091/ktor-todo-exmaple>)
- nuxt-todo-exmaple (<https://github.com/chichi1091/nuxt-todo-exmaple>)

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

目次

本書について	2
対象読者	2
本書の執筆環境と動作環境	2
サンプルコード	3
免責事項	3
第 1 章 フレームワークの紹介	6
1.1 Ktor	6
1.2 Nuxt.js	6
第 2 章 プロジェクトを作ろう	8
2.1 Ktor プロジェクトを作る	8
2.2 Nuxt プロジェクトを作る	10
第 3 章 データベースの準備	12
3.1 データベースへの接続	12
3.2 DB マイグレーション	14
第 4 章 API を作ろう	16
4.1 Entity と Dao	16
4.2 Service クラス	17
4.3 CRUD 用 API	18
4.4 ルーティングの登録	19
第 5 章 UI を作ろう	21
5.1 ダミーデータ	21
5.2 ナビゲーションバー	22
5.3 TODO 一覧	23

5.4	追加・編集・削除	24
第 6 章	サーバとフロントを連携しよう	28
6.1	Axios と Proxy の設定	28
6.2	検索	29
6.3	新規登録・更新・削除	29
第 7 章	Heroku に公開しよう	30
7.1	Heroku とは?	30
7.2	Heroku CLI の導入	30
7.3	サーバサイドのデプロイ	31
7.3.1	サーバ Heroku プロジェクトの作成	31
7.3.2	PostgreSQL の導入	32
7.3.3	app.json の作成	33
7.3.4	Procfile と system.properties の作成	33
7.3.5	実行可能 jar を作成する Gradle タスク	33
7.3.6	データベース参照設定	34
7.4	フロントエンドのデプロイ	35
7.4.1	フロント Heroku プロジェクトの作成	35
7.4.2	nuxt.config.js の修正	36
7.4.3	package.json の設定	36
おわりに		37
長野 Java ユーザーグループ	37

第 1 章

フレームワークの紹介

本書で利用するフレームワークについて紹介します。

1.1 Ktor

Ktor (<https://jp.ktor.work/>) とは IntelliJ IDEA などの IDE や Kotlin を開発している JetBrains 社が作成した軽量 Web フレームワークです。

All Kotlin で開発されており、特徴として「薄い」フレームワークで必要な機能 (feature) を追加していくことで新たな機能を利用することができます。

サーバサイドで Kotlin といえば Spring Boot を選択することが多いと思いますが、Ktor は 2018 年 11 月に v1.0.0 がリリースされ、2020 年 3 月現在 v1.3.2 までアップデートされており、本格的に利用しても良さそうに感じます。次は追加できる主な機能 (feature) になります。

- ktor-thymeleaf：テンプレートエンジン
- ktor-websockets：WebSocket
- ktor-jackson：JSON
- ktor-auth：認証

1.2 Nuxt.js

Nuxt.js (<https://ja.nuxtjs.org/>) は、Vue.js の公式ガイドラインにそって強力なアーキテクチャを提供するように設計されたフレームワークです。

サーバサイドレンダリングを行うアプリケーションの開発に必要な設定が、あらかじめ行われているのが特徴で次のパッケージが含まれています。

- Vue2：Vue.js 本体
- Vue-Router：ルーティング管理
- Vuex：状態管理
- Vue Server Renderer：サーバサイドレンダリング
- Vue-Meta：メタ情報管理

第2章

プロジェクトを作ろう

Ktor プロジェクトと Nuxt.js プロジェクトを作成していきます。

バックエンドフレームワークは Kotlin + Ktor、フロントエンドは Nuxt.js を利用して RESTful API + SPA 構成で簡単な TODO アプリケーションを作成していきます。

また本書では定番の TODO アプリを例に基本的な RESTful API と SPA を開発する流れを示すことを目的とするため、本格的な開発では当然検討するであろう入力バリデーション、エラーハンドリング、ログ出力、UI/UX 設計、自動テスト、静的解析、CI/CD などについては省略します。

2.1 Ktor プロジェクトを作る

Ktor Project Generator (<https://start.ktor.io/>) を使ってプロジェクトを作成します。Spring Initializr 同様に初期プロジェクトを作成することができるサイトです。まずは Configuration を次の様に、その他はデフォルトのまま作成しましょう。

- Configuration
 - Gradle project
 - Server Engine: Tomcat
 - Ktor 1.3.2
 - Group : com.todo.exmaple
 - Name : todo-api
 - Version : 0.0.1

▲ 2.1 Ktor Project Generator

Build ボタンをクリックすると Gradle プロジェクトとしてソースコード一式がダウンロードされますのでお好みのエディタや IDE に取り込みを行ってください。筆者は IntelliJ IDEA を利用しています。

Application.kt を修正して Hello World を出力してみましょう。

▼ src/Application.kt

```
fun main(args: Array<String>): Unit = io.ktor.server.tomcat.EngineMain.main(args)

@Suppress("unused") // Referenced in application.conf
@kotlin.jvm.JvmOverloads
fun Application.module(testing: Boolean = false) {
    routing {
        get("/") {
            call.respondText("Hello World!", ContentType.Text.Plain)
        }
    }
}
```

Gradle で Tomcat を起動させて、http://localhost:8080 にアクセスすると Hello World!が表示されます。これで準備万端です。

```
$ ./gradlew run
$ curl http://localhost:8080
Hello World!
```

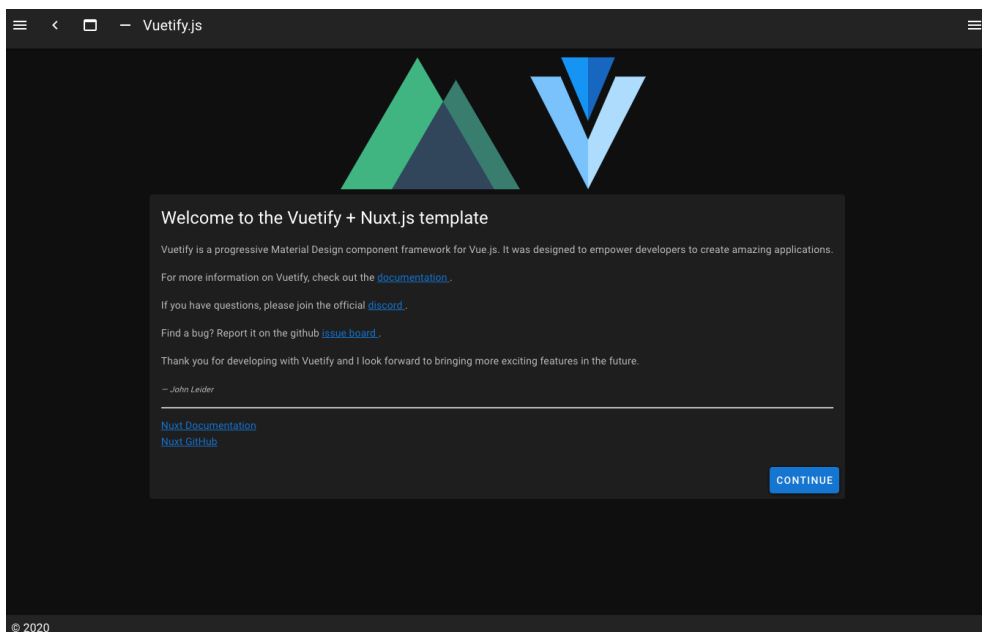
2.2 Nuxt プロジェクトを作る

Yarn を使って Nuxt.js プロジェクトを作成します。10 項目ほどの回答で Nuxt.js プロジェクトができて上がります。

```
$ yarn create nuxt-app todo-ui
yarn create v1.19.2
create-nuxt-app v3.2.0
Generating Nuxt.js project in todo-ui
? Project name: todo-ui // プロジェクト名
? Programming language: JavaScript // 開発言語
? Package manager: Yarn // パッケージマネージャー
? UI framework: Vuetify.js // UIフレームワーク
? Nuxt.js modules: (Press <space> to select, <a> to toggle all
, <i> to invert selection) // nuxt Communityが提供するモジュール
? Linting tools: ESLint, StyleLint // ソースルール
? Testing framework: Jest // ユニットテスト
? Rendering mode: Single Page App // レンダリングモード
? Deployment target: Server (Node.js hosting) // 動作モード
? Development tools: jsconfig.json (Recommended for VS Code if you're not using type
発ツール
```

Yarn コマンドで起動させて、ブラウザで <http://localhost:3000> にアクセスすると Nuxt.js の画面が表示されます。これでフロント側の準備も万全です。

```
$ yarn dev
$ open http://localhost:3000
```



▲図 2.2 Nuxt.js 初期表示

第 3 章

データベースの準備

データベースは PostgreSQL を使っていきます。ローカル環境では Docker Compose を利用してデータベース環境を用意します。docker-compose.yml ファイルを作成して次を記載してください。

▼ src/docker-compose.yml

```
version: "3"
services:
  postgresql:
    image: postgres:latest
    ports:
      - "5432:5432"
    environment:
      POSTGRES_USER: ktoruser
      POSTGRES_PASSWORD: ktorpass
      POSTGRES_DB: todo
      POSTGRES_INITDB_ARGS: "--encoding=UTF-8 --locale=C"
      TZ: "Asia/Tokyo"
```

```
$ docker-compose up -d
```

PostgreSQL の Docker Image のダウンロードとデータベースが起動します。

3.1 データベースへの接続

build.gradle に次の dependencies を追加します。

▼ build.gradle

```
dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8:$kotlin_version"
    implementation "io.ktor:ktor-server-tomcat:$ktor_version"
    implementation "ch.qos.logback:logback-classic:$logback_version"

    + implementation "org.jetbrains.exposed:exposed:0.17.7"
    + implementation 'com.zaxxer:HikariCP:3.4.5'
    + implementation "org.postgresql:postgresql:42.2.14"

    testImplementation "io.ktor:ktor-server-tests:$ktor_version"
}
```

- exposed: JetBrains 製の OR マッパー
- HikariCP: JDBC コネクションプール
- postgresql: PostgreSQL 用 JDBC ドライバー

続いて、application.conf に接続情報を追加します。接続先は Docker で起動した PostgreSQL となります。

▼ resources/application.conf

```
db {
    jdbcUrl = "jdbc:postgresql://localhost:5432/todo"
    dbUser = ktoruser
    dbPassword = ktorpass
}
```

HikariCP を利用したコネクションプールを実装します。コネクションプールについて細かく説明しませんが、接続時に複数の DB 接続を確保し使い回すことで同時に要求があった場合でも捌けるようにしようという仕組みです。

▼ factory/DatabaseFactory.kt

```
object DatabaseFactory {
    // application.confからの読み込むは省略
    ...

    fun init() {
        Database.connect(hikari())
    }

    private fun hikari(): HikariDataSource {
        val config = HikariConfig()
        config.driverClassName = "org.postgresql.Driver"
        config.jdbcUrl = dbUrl
        config.username = dbUser
        config.password = dbPassword
        config.maximumPoolSize = 3
        config.isAutoCommit = false
        config.transactionIsolation = "TRANSACTION_REPEATABLE_READ"
    }
}
```

```
        config.validate()
        return HikariDataSource(config)
    }

    suspend fun <T> dbQuery(
        block: suspend () -> T): T =
        newSuspendedTransaction { block() }
    }
```

DatabaseFactory.init メソッドを Application.kt から呼び出すことで起動時にデータベースとの接続を確立することができます。

▼ src/Application.kt

```
fun Application.module(testing: Boolean = false) {
    + DatabaseFactory.init()

    routing {
        get("/") {
            call.respondText("Hello World!", ContentType.Text.Plain)
        }
    }
}
```

3.2 DB マイグレーション

Exposed はテーブルを作成する方法を提供しますが、スキーマ/データに変更を加えるための機能を提供しないため、Flyway を使用してデータベースの移行を管理します。

build.gradle に次を dependencies を追加します。

▼ build.gradle

```
dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8:$kotlin_version"
    implementation "io.ktor:ktor-server-tomcat:$ktor_version"
    implementation "ch.qos.logback:logback-classic:$logback_version"

    implementation "org.jetbrains.exposed:exposed:0.17.7"
    implementation 'com.zaxxer:HikariCP:3.4.5'
    implementation "org.postgresql:postgresql:42.2.14"

    + implementation 'org.flywaydb:flyway-core:6.5.3'

    testImplementation "io.ktor:ktor-server-tests:$ktor_version"
}
```

マイグレーション用の DDL は src/resources/db/migration に配置し V1__create_todos_table.sql ファイルを作成して CREATE 文を追記します。

▼ resources/db/migration/V1__create_todo_table.sql

```
CREATE TABLE todos (  
    id SERIAL NOT NULL,  
    task TEXT NOT NULL,  
    PRIMARY KEY (id)  
);
```

DatabaseFactory.init メソッドの DB 接続が完了した後に Flyway の実行を行います。これにより、アプリケーションの起動時に適用されていない DDL が自動的に実行されます。

▼ factory/DatabaseFactory.kt

```
fun init() {  
    Database.connect(hikari())  
    +   val flyway = Flyway.configure().dataSource(dbUrl, dbUser, dbPassword).load()  
    +   flyway.migrate()  
}
```

データベースへの変更が起きた場合には、V2、V3 というファイル名でファイルを用意することで、起動時に DDL を実行することができます。

第4章

API を作ろう

続いてデータベースとやり取りを行う API サーバを作っていきます。
Web アプリケーションを開発したことがある方であれば特に複雑なことはありません。

4.1 Entity と Dao

TODOS テーブルに対応する Entity クラスと Dao クラスを作成していきます。
Table クラスを継承することで Entity クラスと Dao クラスとして利用することができ、SELECT や INSERT メソッドを利用することができます。Todo クラスと NewTodo クラスはフロントエンドとの Request と Response を定義するクラスとして作成します。

▼ src/model/Todos.kt

```
object Todos: Table() {
    val id: Column<Int> = integer("id").autoIncrement().primaryKey()
    val task: Column<String> = varchar("task", 4000)
}

data class Todo (
    val id: Int,
    val task: String
)

data class NewTodo (
    val id: Int?,
    val task: String
)
```

4.2 Service クラス

DB の検索・登録・更新・削除を行う Service クラスを作成します。前章で作成した DatabaseFactory.dbQuery メソッド経由で SQL を実行していきます。

▼ service/ToDoService.kt

```
class ToDoService {
    suspend fun getAllTodos(): List<Todo> = dbQuery {
        Todos.selectAll().map { convertTodo(it) }
    }

    suspend fun getTodo(id: Int): Todo? = dbQuery {
        Todos.select {
            (Todos.id eq id)
        }.mapNotNull { convertTodo(it) }
            .singleOrNull()
    }

    suspend fun addTodo(todo: NewTodo): Todo {
        var key = 0
        dbQuery {
            key = (Todos.insert {
                it[task] = todo.task
            } get Todos.id)
        }
        return getTodo(key)!!
    }

    suspend fun updateTodo(todo: NewTodo): Todo? {
        val id = todo.id
        return if (id == null) {
            addTodo(todo)
        } else {
            dbQuery {
                Todos.update({ Todos.id eq id }) {
                    it[task] = todo.task
                }
            }
            getTodo(id)
        }
    }

    private fun convertTodo(row: ResultRow): Todo =
        Todo(
            id = row[Todos.id],
            task = row[Todos.task]
        )
}
```

convertTodo メソッドは得られた検索結果を Todo クラスに変換をします。

4.3 CRUD 用 API

DB との接続準備が整いましたので、検索・登録・更新・削除の API を作っていきます。エンドポイントは次とします。

- GET /todos
- GET /todos/{id}
- POST /todos
- PUT /todos/{id}
- DELETE /todos/{id}

エンドポイントの受け口となる Controller を作成します。Route 関数でパスセグメントを定義し、HTTP Method に対応する関数を定義します。

▼ web/ToDoController.kt

```
fun Route.todos(todoService: TodoService) {
    route("todos") {
        get("/") {
            call.respond(todoService.getAllTodos())
        }

        get("/{id}") {
            val id = call.parameters["id"]?.toInt()
            ?: throw IllegalStateException("Must To id")
            val todo = todoService.getTodo(id)
            if (todo == null) call.respond(HttpStatusCode.NotFound)
            else call.respond(todo)
        }

        post("/") {
            val newTodo = call.receive<NewTodo>()
            call.respond(HttpStatusCode.Created, todoService.addTodo(newTodo))
        }

        put("/{id}") {
            val todo = call.receive<NewTodo>()
            val updated = todoService.updateTodo(todo)
            if (updated == null) call.respond(HttpStatusCode.NotFound)
            else call.respond(HttpStatusCode.OK, updated)
        }

        delete("/{id}") {
            val id = call.parameters["id"]?.toInt()
            ?: throw IllegalStateException("Must To id");
            val removed = todoService.deleteTodo(id)
            if (removed) call.respond(HttpStatusCode.OK)
            else call.respond(HttpStatusCode.NotFound)
        }
    }
}
```

```
}
```

call.parameters はクエリーパラメータを受け取るため、call.receive は POST、PUT のリクエストボディ（payload）を受け取るためのメソッドになります。

4.4 ルーティングの登録

このままだと作成したエンドポイントが登録されておらず、呼び出すことができませんので、Application.kt にルーティングを install する必要があります。あと、API の返却結果を JSON 形式にしたいため、Jackson コンテンツ機能も合わせて install します。Jackson コンテンツ機能を install することで API の返却結果がすべて JSON 形式にすることができます。

build.gradle に Jackson を dependencies を追加します。

▼ build.gradle

```
dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8:$kotlin_version"
    implementation "io.ktor:ktor-server-tomcat:$ktor_version"
    implementation "ch.qos.logback:logback-classic:$logback_version"
+   implementation "io.ktor:ktor-jackson:$ktor_version"

    implementation "org.jetbrains.exposed:exposed:0.17.7"
    implementation 'com.zaxxer:HikariCP:3.4.5'
    implementation "org.postgresql:postgresql:42.2.14"

    implementation 'org.flywaydb:flyway-core:6.5.3'

    testImplementation "io.ktor:ktor-server-tests:$ktor_version"
}
```

▼ Application.kt

```
fun Application.module(testing: Boolean = false) {
+   install(ContentNegotiation) {
+       jackson {
+           configure(SerializationFeature.INDENT_OUTPUT, true)
+       }
+   }

    DatabaseFactory.init()

+   val todoService = TodoService()
+   install(Routing) {
+       todos(todoService)
+   }
}
```

第 4 章 API を作ろう

install を増やすことで Jackson のように機能（feature）を追加することが簡単に行えます。

第 5 章

UI を作ろう

TODO を一覧表示する画面と追加・更新するモーダル画面、削除機能を Nuxt.js と Vuetify のデフォルト画面をベースに作っていきます。

いきなり API 連携は行わずにダミーデータを表示する形にして実装イメージを固めていきましょう。

5.1 ダミーデータ

index.js ファイルを作成し、ダミーデータと TODO を操作するメソッドを実装します。

▼ store/index.js

```
export const state = () => ({
  todos: [
    {
      id: 1,
      task: 'タスク 1'
    },
    ...
    {
      id: 10,
      task: 'タスク 10'
    }
  ]
})

export const getters = {
  getTodos (state) {
    return state.todos
  }
}

export const mutations = {
  addTodo (state, payload) {
    state.todos.push(payload.todo)
  }
}
```

```
    },
    updateTodo (state, payload) {
      state.todos.forEach((todo, index) => {
        if (todo.id === payload.todo.id) {
          state.todos.splice(index, 1, payload.todo)
        }
      })
    },
    removeTodo (state, payload) {
      state.todos.forEach((todo, index) => {
        if (todo.id === payload.todo.id) {
          state.todos.splice(index, 1)
        }
      })
    }
  }
}
```

5.2 ナビゲーションバー

デフォルトで用意されているナビゲーションバーに TODO メニューを追加します。アイコンは Material Design Icon から選ぶことができます。今回はチェックリストっぽい mdi-format-list-checks を使っていきます。items を増やすことでメニューを増やすことができますので、色々試してみてください。

▼ layouts/default.vue

```
<script>
export default {
  data () {
    return {
      clipped: false,
      drawer: false,
      fixed: false,
      items: [
        {
          icon: 'mdi-format-list-checks',
          title: 'TODO',
          to: '/'
        }
      ],
      miniVariant: false,
      right: true,
      rightDrawer: false,
      title: 'TODO'
    }
  }
}
</script>
```

5.3 TODO 一覧

一覧は vuetify の v-data-table コンポーネントを使っていきます。headers に text と value をセットして items にリストを渡すと一覧が表示され、ソートとページングも自動で動作してくれます。items-per-page を変更すると 1 ページに表示される件数が変更できます。

▼ pages/index.vue

```
<template>
  <v-layout
    column
    justify-center
    align-center
  >
    <v-card v-if="todos">
      <v-card-title>
        TODO一覧
      </v-card-title>
      <v-spacer />
      <v-text-field
        v-model="search"
        append-icon="mdi-magnify"
        label="検索"
        single-line
      />
      </v-card-title>
      <v-data-table
        :headers="headers"
        :items="todos"
        :items-per-page="5"
        search="search"
        sort-by="id"
        :sort-desc="true"
        class="elevation-1"
      >
      </v-data-table>
    </v-card>
  </v-layout>
</template>

<script>
export default {
  data () {
    return {
      search: '',
      headers: [
        { text: 'ID', value: 'id' },
        { text: 'タスク', value: 'task' },
        { text: '操作', value: 'actions' }
      ],
      todo: {}
    }
  },
}
```



```
computed: {
  todos () {
    return this.$store.getters.getTodos
  }
}
}
</script>
```

5.4 追加・編集・削除

続いて TODO を追加するための＋アイコンを一覧表の上に設置し、一覧の各行にも編集用の鉛筆アイコンと削除用のゴミ箱アイコンを設置していきます。追加と編集時にはタスクを入力する必要があるので、v-dialog コンポーネントでモーダル画面も作成します。

▼ pages/index.vue

```
<!-- 新規追加 -->
<v-btn
  fab
  dark
  small
  color="dark"
  class="mb-2"
  @click="add"
>
  <v-icon dark>
    mdi-plus
  </v-icon>
</v-btn>

<v-data-table>
  <!-- モーダル -->
  <template v-slot:top>
    <v-dialog v-model="dialog" max-width="500px">
      <v-card>
        <v-card-title>
          <span class="headline">{{ formTitle }}</span>
        </v-card-title>
        <v-card-text>
          <v-container>
            <v-row>
              <v-col cols="12">
                <v-text-field v-model="todo.task" label="タスク" />
              </v-col>
            </v-row>
          </v-container>
        </v-card-text>
        <v-card-actions>
          <v-spacer />
          <v-btn @click="close">閉じる</v-btn>
          <v-btn v-if="isPersistedTodo" class="primary" @click="update">更新する
        </v-card-actions>
      </v-card>
    </v-dialog>
  </template>
</v-data-table>
```

```

        <v-btn v-else class="primary" @click="create">追加する</v-btn>
        <v-spacer />
      </v-card-actions>
    </v-card>
  </v-dialog>
</template>

<!-- 編集と削除 -->
<template v-slot:[`item.actions`]="{ item }">
  <v-icon
    small
    @click="edit(item)"
  >
    mdi-pencil
  </v-icon>
  <v-icon
    small
    @click="remove(item)"
  >
    mdi-delete
  </v-icon>
</template>
</v-data-table>

```

次は各イベント操作を行うための JavaScript になります。まだ API 連携は行っていないので store/index.js を呼び出すことになります。

▼ pages/index.vue

```

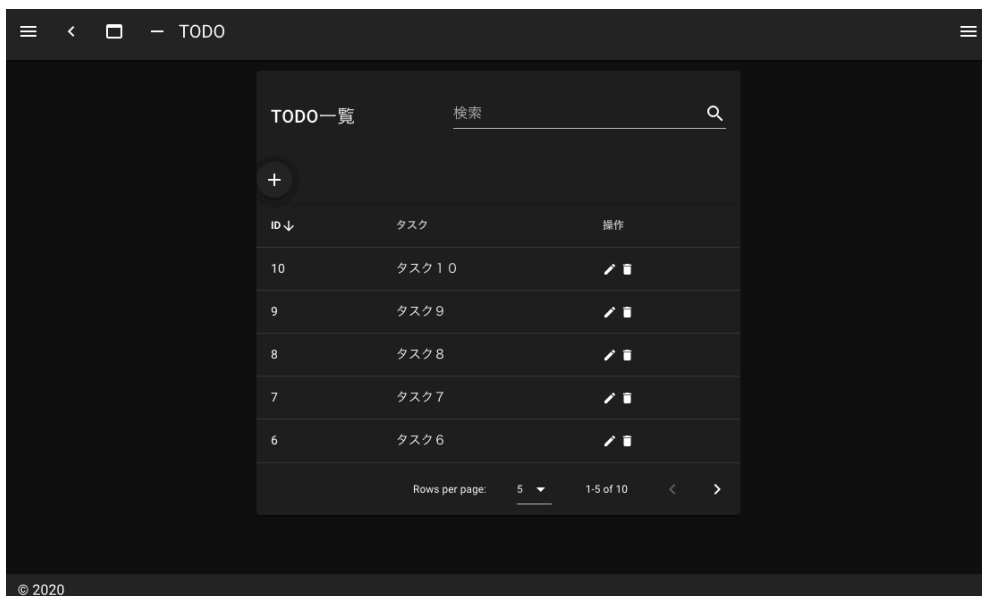
<script>
export default {
  computed: {
    isPersistedTodo () {
      return !!this.todo.id
    },
    formTitle () {
      return this.isPersistedTodo ? 'TODO編集' : 'TODO追加'
    }
  },
  methods: {
    add (todo) {
      this.todo = {}
      this.dialog = true
    },
    create () {
      const payload = { todo: this.todo }
      this.$store.commit('addTodo', payload)
      this.close()
    },
    edit (todo) {
      this.todo = Object.assign({}, todo)
      this.dialog = true
    },
    update () {
      const payload = { todo: this.todo }
      this.$store.commit('updateTodo', payload)
    }
  }
}

```

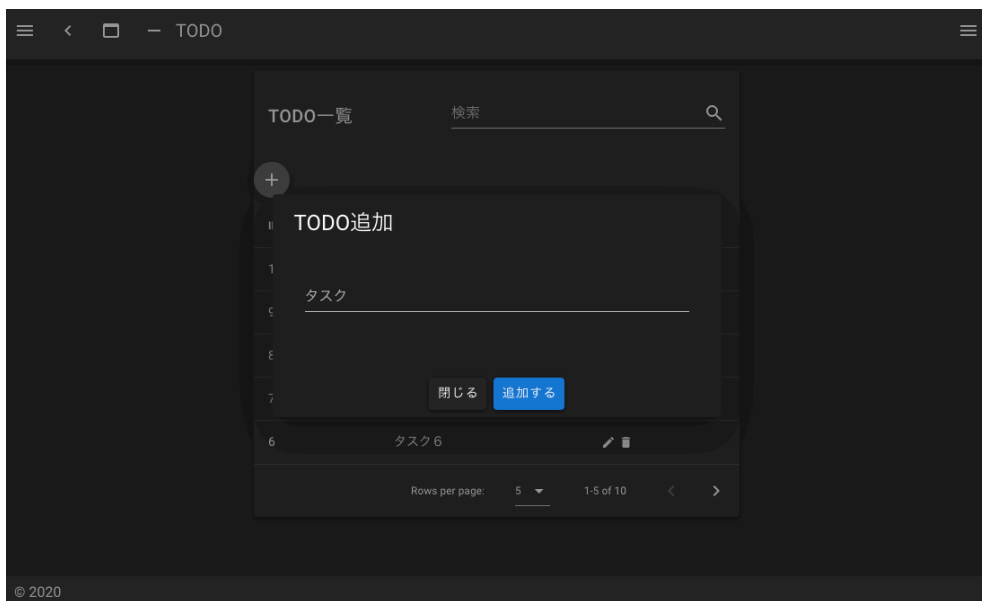
第5章 UI を作ろう

```
      this.close()
    },
    remove (todo) {
      const payload = { todo }
      this.$store.commit('removeTodo', payload)
    },
    close () {
      this.dialog = false
      this.todo = {}
    }
  }
}
</script>
```

完成した画面はこちらになります。Vuetify のコンポーネントを利用していることで少量のソースコードでも多機能な一覧画面を作ることができました。



▲図 5.1 TODO 一覧画面



▲図 5.2 TODO 新規追加モーダル

第 6 章

サーバとフロントを連携しよう

サーバとフロントの準備が整いましたのでサーバとフロントの繋ぎこみを行います。フロントから API を呼び出すのに Axios を使っていきます。

まずはモジュールをインストールします。フロントとサーバが別サーバで起動しているのでオリジン間リソース共有 (CORS) 制約に引っかかってしまいます。対処方法として Proxy モジュールを利用しますので合わせてインストールします。

```
$ yarn add @nuxtjs/axios
$ yarn add @nuxtjs/proxy
```

データは API サーバから取得するため、前章で作成した store/index.js はもう必要ありませんので削除してしまいましょう。

6.1 Axios と Proxy の設定

Axios と Proxy の設定を行います。nuxt.config.js に次を追加します。

▼ nuxt.config.js

```
modules: [
  '@nuxtjs/axios',
  '@nuxtjs/proxy'
],
axios: {
},
proxy: {
  '/api/': { target: 'http://localhost:8080', pathRewrite: { '^/api/': '' } }
},
```

proxy モジュールを利用して /api/ へのアクセスがされた場合、API サーバへ接続する

ようにしました。また、/api/はエンドポイントとして必要ないので削除するようにしています。

6.2 検索

computed で async を使うことができませんので asyncData で一覧検索を行います。

▼ pages/index.vue

```
async asyncData ({ app }) {  
  const response = await axios.get('http://localhost:3000/api/todos')  
  return { todos: response.data }  
},
```

検索した結果を、data/todos に格納してあげることで一覧に表示することができます。

6.3 新規登録・更新・削除

methods で TODO データを操作している箇所も Axios に変えていきます。処理が成功すると再表示して一覧を更新するように reload も行います。

▼ pages/index.vue

```
methods: {  
  async create () {  
    await axios.post('/api/todos', this.todo)  
    .then(() => { this.$router.app.refresh() })  
    this.close()  
  },  
  async update () {  
    await axios.put('/api/todos/' + this.todo.id, this.todo)  
    .then(() => { this.$router.app.refresh() })  
    this.close()  
  },  
  async remove (todo) {  
    await axios.delete('/api/todos/' + todo.id, todo)  
    .then(() => { this.$router.app.refresh() })  
  },  
}
```

入力チェックやエラー処理を省いているものの、だいぶすっきりしたコードになりました。API の繋ぎこみが完成しましたので、TODO を登録したり更新すると API 経由でデータベースに登録されることを確認することができます。

第 7 章

Heroku に公開しよう

作成したアプリケーションをデプロイしてインターネットに公開します。今回デプロイ先は Heroku を使います。

7.1 Heroku とは？

Heroku とは開発した Web アプリケーションを 10 分程度で公開することができるホスティングサービスです。

アカウント未認証でも利用可能ですが、クレジットカード登録をすることで無料利用できる時間や作成できるプロジェクト数が増えますのでクレジットカードを登録することをお勧めします。有料サービスを利用しなければ請求されることもありません。詳しくは Heroku (<https://jp.heroku.com/>) をご覧ください。

7.2 Heroku CLI の導入

Heroku CLI をインストールします。主にアプリケーションデプロイで利用しますが、次のような便利な機能が豊富にあります。

- アプリケーションログの参照
- プロジェクトの作成・削除
- サーバのステータス確認・再起動

```
$ brew tap heroku/brew && brew install heroku
$ heroku login
heroku: Press any key to open up the browser to login or q to exit:
```

```
Logging in... done
Logged in as dummy@dummy.com
```

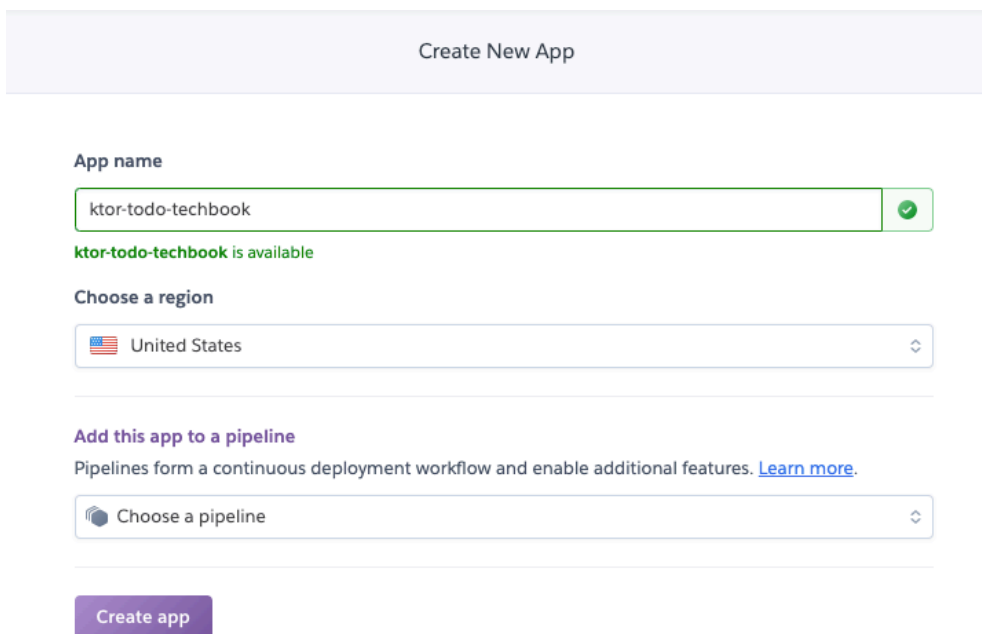
ログインコマンドを実行するとブラウザが起動しますので認証すれば CLI 側も認証することができます。認証が行われればインストールは無事終了です。

7.3 サーバサイドのデプロイ

まずはじめに API サーバをデプロイしていきます。本書ではソースコードを GitHub でソースを管理しているので、master への Push でデプロイするように構築していきます。

7.3.1 サーバ Heroku プロジェクトの作成

Ktor プロジェクトを Heroku にデプロイしていきます。名前は **ktor-todo-techbook** とします。Heroku の管理画面よりアプリケーション名を入力し Create app ボタンをクリックしてください。



Create New App

App name

ktor-todo-techbook

ktor-todo-techbook is available

Choose a region

United States

Add this app to a pipeline

Pipelines form a continuous deployment workflow and enable additional features. [Learn more.](#)

Choose a pipeline

Create app

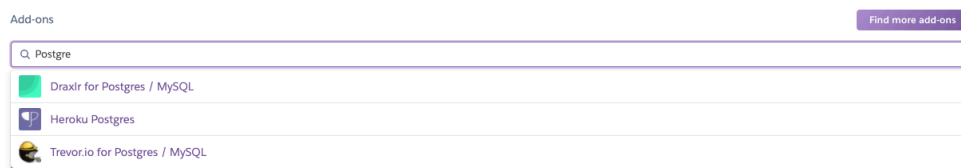
▲図 7.1 サーバプロジェクトの作成

作成したプロジェクトと GitHub を連携し、Automatic deploys を設定しておくと GitHub への Push と同時にデプロイされるので連携しておきましょう。

7.3.2 PostgreSQL の導入

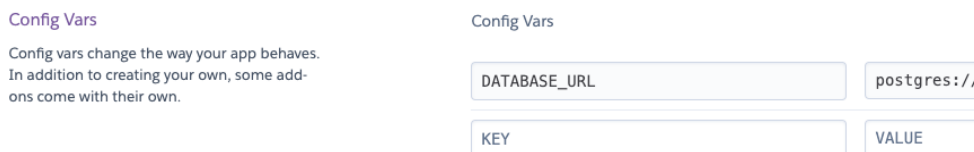
Heroku では PostgreSQL を公式にサポートしており、無料で利用することができます。他のデータベースも利用することができますが、有料ですので **Databases & Data Management** (<https://devcenter.heroku.com/categories/data-management>) で確認してからご利用ください。

作成したプロジェクトに PostgreSQL の Add-ons を追加します。管理画面の Resources > Add-ons で「Postgre」と入力して検索します。絞り込まれた「Heroku Postgres」を選択して追加します。



▲図 7.2 Add-ons の追加

データベース URL は管理画面の Settings > Config Variables で確認することができます。環境変数としてアプリケーションから参照することが可能です。



▲図 7.3 データベースの URL

アプリケーションの設定では利用しませんが、ユーザー ID とパスワードは Datastores で確認することができます。

1. 管理画面の Resources > Add-ons の「Heroku Postgres」をクリック
2. Datastores の Settings > Database Credentials

7.3.3 app.json の作成

Ktor プロジェクトと Heroku プロジェクトとの依存関係を定義するため、app.json ファイルを Ktor プロジェクトに作成します。name と description はアプリケーションの定義を記入し、image と addons は実行用 Docker イメージと Heroku に追加した add-ons を記載します。

▼ app.json

```
{
  "name": "todo api server with Ktor",
  "description": "todo application api server.",
  "image": "heroku/java",
  "addons": [ "heroku-postgresql" ]
}
```

7.3.4 Procfile と system.properties の作成

アプリケーションの実行方法を定義します。

▼ Procfile

```
web:    java -jar build/libs/todo-api-0.0.1.jar
```

Java のバージョンを定義するため system.properties ファイルも作成します。

▼ system.properties

```
java.runtime.version=1.8
```

7.3.5 実行可能 jar を作成する Gradle タスク

Procfile に追記した jar ファイルを作成するための Gradle タスクを追加します。デプロイ時に Gradle の stage タスクが実行されるのでタスクを追加します。

▼ build.gradle

```
jar {
    manifest {
        attributes 'Main-Class': "com.todo.exmaple.ApplicationKt"
    }
    from {
```

```
        configurations.compile.collect {
            it.isDirectory() ? it : zipTree(it)
        }
    }
}

task stage {
    dependsOn installDist
}
```

7.3.6 データベース参照設定

Heroku で動作している PostgreSQL に接続するように URL を変更します。Heroku でアプリケーションが開始されると動的環境変数で JDBC 用の URL とユーザ情報設定が追加されます。動的環境変数を利用してアプリケーションから接続を行います。

▼ build.gradle

```
def JDBC_DATABASE_URL = System.getenv()['JDBC_DATABASE_URL'] == null
    ? "jdbc:postgresql://localhost:5432/todo"
    : System.getenv()['JDBC_DATABASE_URL']
def JDBC_DATABASE_USERNAME = System.getenv()['JDBC_DATABASE_USERNAME'] == null
    ? "ktoruser"
    : System.getenv()['JDBC_DATABASE_USERNAME']
def JDBC_DATABASE_PASSWORD = System.getenv()['JDBC_DATABASE_PASSWORD'] == null
    ? "ktorpass"
    : System.getenv()['JDBC_DATABASE_PASSWORD']

flyway {
    url = JDBC_DATABASE_URL
    user = JDBC_DATABASE_USERNAME
    password = JDBC_DATABASE_PASSWORD
    baselineOnMigrate=true
    locations = ["filesystem:resources/db/migration"]
}
```

▼ resources/application.conf

```
db {
    jdbcUrl = "jdbc:postgresql://localhost:5432/todo"
    +   jdbcUrl = ${?JDBC_DATABASE_URL}
    dbUser = ktoruser
    +   dbUser = ${?JDBC_DATABASE_USERNAME}
    dbPassword = ktorpass
    +   dbPassword = ${?JDBC_DATABASE_PASSWORD}
}
```

デプロイが行われ Ktor アプリケーションが起動すると Flyway により TODOS テーブルが自動で作成されます。

7.4 フロントエンドのデプロイ

続いて Nuxt.js プロジェクトを Heroku にデプロイを行います。こちらも GitHub でソースを管理しているので、master へ Push されると自動でデプロイされるように構築していきます。

7.4.1 フロント Heroku プロジェクトの作成

Heroku にプロジェクトを作成します。**nuxt-todo-techbook** という名前でプロジェクトを作成します。

App name

nuxt-todo-techbook

nuxt-todo-techbook is available

Choose a region

United States

Add this app to a pipeline

Pipelines form a continuous deployment workflow and enable additional features. [Learn more.](#)

Choose a pipeline

Create app

▲図 7.4 フロントプロジェクトの作成

次に Nuxt.js アプリケーションを Production モードで起動するように行うのと API サーバの URL を環境変数に設定します。

```
$ heroku config:set HOST=0.0.0.0 --app nuxt-todo-techbook
$ heroku config:set NODE_ENV=production --app nuxt-todo-techbook
$ heroku config:set API_URL=https://ktor-todo-techbook.herokuapp.com/ \
```

```
--app nuxt-todo-techbook
$ heroku config --app nuxt-todo-techbook
--- nuxt-todo-techbook Config Vars
API_URL:  https://ktor-todo-techbook.herokuapp.com/
HOST:     0.0.0.0
NODE_ENV: production
```

環境変数の確認は `heroku config` コマンドで行います。

7.4.2 nuxt.config.js の修正

Proxy で API サーバへ接続を行っていますので、環境変数で設定した URL を参照するように修正を行います。次の記述で `API_URL` が定義されていれば環境変数で設定した URL を設定し、定義がされていない場合にはローカル環境用の URL を設定することができます。

▼ package.json

```
proxy: {
  '/api/': {
    target: process.env.API_URL || 'http://localhost:8080'
    , pathRewrite: { '~'/api/': '' }
  }
},
```

7.4.3 package.json の設定

`package.json` に次を追加します。後は Commit して Push すれば反映されます。

▼ package.json

```
"scripts": {
  "dev": "nuxt",
  "build": "nuxt build",
  "start": "nuxt start",
+  "heroku-postbuild": "npm run build"
}
```

<https://nuxt-todo-techbook.herokuapp.com/> にアクセスしてください。API サーバに接続できた TODO アプリケーションが表示できました。

おわりに

はじめまして、てっしーです。この度は本書をお読みいただきありがとうございます。長野県在住のサーバサイドエンジニアで、普段は Java や Kotlin、Groovy といった JVM 言語と SpringBoot で開発をしています。実は本書を執筆中に尿管結石が再発してしまい、痛みと戦いながらの執筆でした。皆さんが本書を手にとっていただけたということは無事に入稿できたということでホッとします。

Nuxt.js は仕事でちょっとだけ触れたことがある程度で調べながらの実装でしたので、経験者からすると「もっとこうしたほうが」や「これは間違いで」などあるかもしれません。ぜひ優しくフィードバックをしてもらえるととても嬉しいです。Ktor については私が運営を行っている「長野 Java ユーザーグループ」で登壇させていただいたことがあり、また触りたいなーという思いがあったので使わせていただきました。Kotlin の機能をフルに活かせるフレームワークなのでぜひ本書をきっかけに触れてもらえたらと思います。

認証や DI、ユニットテストなどについてもご紹介したいと思っているので次回の技術書典ではさらに実践的な内容で執筆を目指したいと思います。

長野 Java ユーザーグループ

長野県の東北信を中心に JVM 言語に関する勉強会を開催する**長野 Java ユーザーグループ**を運営しております。

2020 年 1 月に発足したばかりで一緒に運営してくれるメンバーを募集しています。ご興味がありましたらぜひ Twitter (@chichi1091) でお声がけください。

Ktor と Nuxt.js で作る Web アプリケーション入門

2020 年 9 月 12 日 技術書典 9 版 v1.0.0

著 者 chichi1091

編 集 chichi1091

発行所 crescent

(C) 2020 crescent