KtorとNuxt.jsで作るWebアプリケーション入門 Vol2

crescent

Ktor と Nuxt.js で作る Web アプリケーション入門 Vol.2

てっしー　著

2020-12-26 版　　crescent 発行

本書について

対象読者

- 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
- より実践的なポートフォリオを作りたい方
- ドメイン駆動設計に挑戦したい方

本書の執筆環境と動作環境

- Docker 19.03.13
- Docker Compose 1.27.4
- Java 11.0.9.j9-adpt
- Kotlin 1.4.10
- Node.js v14.15.0
- Yarn 1.22.10
- Ktor 1.3.2
- Nuxt.js 2.14.0
- Vuetify 1.11.2

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-example-2（`https://github.com/chichi1091/ktor-todo-example-2`）
- nuxt-todo-example-2（`https://github.com/chichi1091/nuxt-todo-example-2`）

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

第 1 章

前書の振り返り

本書は「Ktor と Nuxt.js で作る Web アプリケーション入門（https://techbookfest.org/product/5755616553336832?productVariantID=4897939897974784）」の続編です。前書をお持ちでない方向けに振り返りを行います。ご興味をお持ちいただければぜひご購入いただけると幸いです。

1.1 フレームワーク

タイトルにも記載されている Ktor と Nuxt.js について簡単に説明します。

1.1.1 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。
2018 年 11 月に v1.0.0 がリリースされ、2020 年 12 月現在 v1.5.0 までアップデートされており、採用実績も増えています。

1.1.2 Nuxt.js

Nuxt.js（https://ja.nuxtjs.org/）は、Vue.js の公式ガイドラインに沿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になります。

1.2 ToDo アプリケーション

前書では次について説明しています。

1. プロジェクト作成
2. Ktor による REST ful API
3. Nuxt.js によるフロントエンド
4. フロントエンドとバックエンドの連携
5. Heroku へのデプロイ

サンプルコードも GitHub にて公開していますので、ぜひ参考にしてみてください。完成画面は次になります。

▲図 1.1　ToDo 一覧画面

▲図 1.2　ToDo 新規追加モーダル

第 2 章

認証

前書で作成した ToDo アプリケーションでは誰もが参照・登録ができてしまい、セキュリティ面に問題があります。実際の ToDo 管理では登録者や担当者などが登場することがあるので、ユーザーの概念が必要となります。
本章ではユーザーー登録と認証を新たに実装していきます。アカウントとパスワードを管理するテーブルを用意して認証し、JWT トークン発行し API 呼び出しが認証済みであるか評価します。

2.1 JWT トークン

JWT は JSON Web Tolen の略称で、属性情報（Claim）を JSON データ構造で表現したトークンの使用です。JWT は次のような特徴を持っています。

1. 署名されたデータのため改ざんをチェックできる
2. URL Safe なデータ（URL に組めることができる文字のみで構成）
3. JSON 文字列

デコードして JSON 文字列を得ることができ、鍵情報を持たなくても復元できます。

2.2 アカウントテーブル

アカウントを管理するテーブルを PostgreSQL に作成していきます。db/migration に V2___create_accounts_table.sql ファイルを作成します。db/migration に SQL ファイルを作成することで Flyaway（DB マイグレーション）が SQL を自動で実行してくれます。
アカウントテーブルは以下の成約を設けます。

1. プライマリキーはシーケンスを利用する
2. パスワードはハッシュ処理をする
3. 同一のメールアドレスは登録できない

▼ resources/db/migration/V2__create_accounts_table.sql

```
create table accounts (
    id SERIAL NOT NULL,
    password TEXT,
    name TEXT NOT NULL,
    email varchar(60) NOT NULL,
    PRIMARY KEY (id)
);
CREATE UNIQUE INDEX unq_accounts_email ON accounts (email);
```

アカウントテーブルの model クラスを作成します。
Login クラスはサインインするときの Payload クラスです。AuthUser クラスはログインしているアカウント情報を Principal に登録する際に利用します。

▼ model/Account.kt

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

data class Account (
    val id: Int,
    val password: String,
    val name: String,
    val email: String
)

data class Login (
    val email: String,
    val password: String
)

data class AuthUser(val id: Int): Principal {
    companion object {
        fun toAuthUser(principal: JWTPrincipal) = AuthUser(
            principal.payload.getClaim("id")?.asInt()
                ?: throw IllegalStateException("unauthorized")
        )
    }
}

data class NewAccount (
    val id: Int?,
    val password: String,
    val name: String,
```

```
    val email: String,
)
```

続いて Service クラスを作成し、アカウントの登録、ログイン認証を作成します。ユーザーが入力したパスワードを Java の標準クラスの MessageDigest クラスを利用してハッシュ化しています。MessageDigest クラスでダイジェスト値を求めた場合、戻り値が byte 配列になるので注意が必要です。

▼ service/AccountService.kt

```
class AccountService {
    suspend fun createAccount(account: NewAccount): Account {
        var key = 0
        dbQuery {
            key = (Accounts.insert {
                it[password] = createHash(account.password!!)
                it[name] = account.name
                it[email] = account.email
            } get Accounts.id)
        }
        return getAccount(key)!!
    }

    suspend fun authentication(email: String, password: String): Account? =
        dbQuery {
            Accounts.select {
                (Accounts.email eq email) and
                (Accounts.password eq createHash(password))
            }.mapNotNull { convertAccount(it) }
                .singleOrNull()
        }

    internal fun createHash(value: String): String {
        return MessageDigest.getInstance("SHA-256")
            .digest(value.toByteArray())
            .joinToString(separator = "") {
                "%02x".format(it)
            }
    }

    private fun convertAccount(row: ResultRow): Account =
        Account(
            id = row[Accounts.id],
            password = row[Accounts.password],
            name = row[Accounts.name],
            email = row[Accounts.email],
        )
}
```

2.3 サーバ側認証処理

Ktor では API リクエストの認証処理を簡単に行える Feature が用意されています。JWT トークンを使った認証をします。JWT トークンにはアカウント ID を含めます。

application.conf に JWT を利用するための情報を設定します。

▼ resources/application.conf

```
jwt {
    // 暗号化アルゴリズム
    algorithm = tech_book
    // 発行者
    issuer = issuer
    // JSONのユーザーID区分
    userId = user_id
    // 想定利用者
    audience = "jwt-audience"
    // JWTレルム名
    realm = "ktor-ToDo-example"
}
```

Application.module に Authentication をインストールします。

JWTConf クラスは application.conf で設定した JWT 設定情報を取得するクラスです。詳しい実装は GitHub を参照してください。

validate は認証した結果を取得できます。Payload から取得した認証情報を Principal に変換し router で call.principal 経由で取得できます。

▼ Application.kt

```
fun Application.module() {
    val jwtConfig = JWTConfig()
    install(Authentication) {
        jwt {
            realm = jwtConfig.realm
            verifier(jwtConfig.makeJwtVerifier())
            validate {credential ->
                if (credential.payload.audience.contains(jwtConfig.audience)) {
                    val principal = JWTPrincipal(credential.payload)
                    AuthUser.toAuthUser(principal)
                    principal
                } else null
            }
        }
    }
}
```

ToDo を操作する API はログイン認証が行えていないと利用できません。

そのために authenticate を配置します。逆にログインやアカウント作成は認証していな

くても利用できる必要があるため、authenticate を配置する必要はありません。また、authenticate 配置内であれば call.principal でログインしているアカウントの ID を取得ができます。

▼ web/ToDoController.kt

```
route("ToDos") {
    authenticate {
        get("/") {
            val id = call.principal<AuthUser>()!!.id
            call.respond(ToDoService.getAllToDos())
        }
    }
}
```

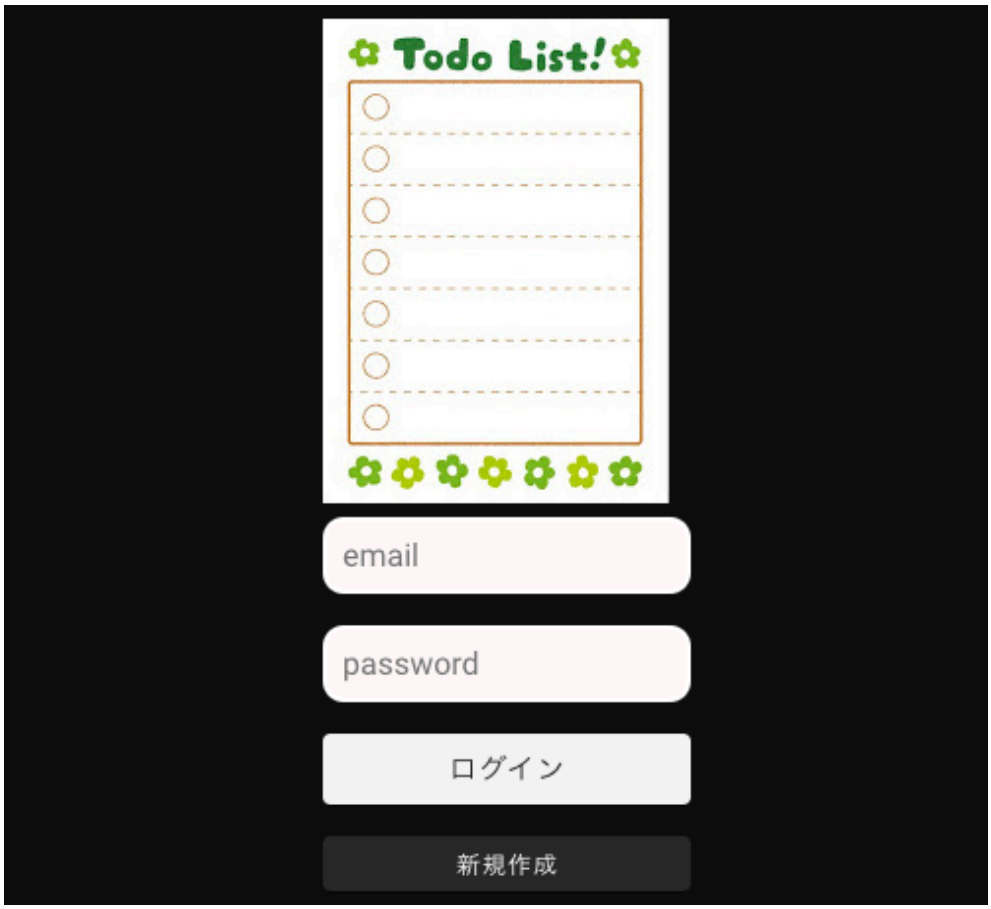
2.4 ログイン処理の実装

システムにログインするためのサインイン画面と、アカウントを作成するためのサインアップ画面を作成します。
サインイン画面では、メールアドレスとパスワードを入力することで認証を行えるようにします。

▼ pages/signin.vue

```
<template>
    <v-layout
        justify-center
        align-center
    >
        <div class="login-form">
            <form @submit.prevent="login">
                <p class="error" v-if="error">{{ error }}</p>
                <img src="logo.jpg">
                <p>
                    <input type="text"
                        v-model="email"
                        placeholder="email"
                        name="email"
                        class="textlines"/>
                </p>
                <p>
                    <input type="password"
                        v-model="password"
                        placeholder="password"
                        name="password"
                        class="textlines"/>
                </p>
                <p>
                    <v-btn
                        light
```

```
                        block
                        elevation="2"
                        type="submit"
                    >ログイン</v-btn>
                </p>
                <p>
                    <v-btn
                        block
                        small
                        @click="signup"
                    >新規作成</v-btn>
                </p>
            </form>
        </div>
    </v-layout>
</template>
```



▲図 2.1　サインイン画面

ログインボタンをクリックしたときのイベント操作をするための JavaScript で、入力されたメールアドレスとパスワードを Ktor の認証 API を呼び出し認証します。API の戻り値として JWT トークンを受け取りますので、Cookie に保存しています。
ToDo 画面を操作するときに Cookie から取得して header にセットします。

▼ pages/signin.vue

```
<script>
import axios from 'axios'

export default {
    layout(context) {
```

```
        return 'login'
    },
    data() {
      return {
        error: null,
        email: "",
        password: "",
      }
    },
    methods: {
      async login() {
        const request = {
            email: this.email,
            password: this.password
        }
        await axios.post('/api/accounts/auth', request)
        .then(response => {
            console.log(response);
            if(response.status === 200) {
                this.$cookies.set('token', response.data.bearer_token);
                this.$router.push("/ToDos");
                return;
            }
            this.error = 'サインインに失敗しました';
        }).catch(error => {
            console.log(error);
            this.error = 'サインインに失敗しました';
        });
      },
      signup() {
        this.$router.push("signup")
      }
    }
  }
</script>
```

続いてサインアップ画面を作成します。アカウント情報となるメールアドレス、名前、そしてパスワードを登録します。

▼ pages/signup.vue

```
<template>
    <v-layout
        justify-center
        align-center
    >
        <v-card>
            <v-card-title>
                アカウント新規作成
            </v-card-title>
            <div class="signup-form">
                <form @submit.prevent="signup">
                    <p v-if="errors.length">
                        <ul>
                            <li style="color: red;"
                                v-for="(error, key) in errors"
```

```
                            :key="key">{{ error }}
                        </li>
                    </ul>
                </p>

                <p v-if="infos.length">
                    <ul>
                        <li style="color: green;"
                            v-for="(info, key) in infos"
                            :key="key">{{ info }}
                        </li>
                    </ul>
                </p>

                <p>
                    <input type="text"
                        v-model="name"
                        placeholder="name"
                        name="name"
                        class="textlines"/>
                </p>
                <p>
                    <input type="text"
                        v-model="email"
                        placeholder="email"
                        name="email"
                        class="textlines"/>
                </p>
                <p>
                    <input type="password"
                        v-model="password"
                        placeholder="password"
                        name="password"
                        class="textlines"/>
                </p>
                <p>
                    <v-btn
                        light
                        block
                        elevation="2"
                        type="submit"
                        >新規作成</v-btn>
                </p>
            </form>
        </div>
    </v-card>
</v-layout>
</template>
```

▲図 2.2　サインアップ画面

新規作成ボタンをクリックすると同じようにアカウント作成 API を呼び出しています。アカウント作成が成功すると、あらためてサインインをするため 2 秒後にサインイン画面を表示しています
簡単な入力チェックも実装しているので参考にしてみてください。

▼ pages/signup.vue

```
<script>
import axios from 'axios'

export default {
    layout(context) {
        return 'login'
    },
    data() {
      return {
        errors: [],
        infos: [],
        name: "",
        email: "",
        password: "",
```

```
      }
    },
    methods: {
      async signup() {
        this.errors = [];
        if (!this.name) this.errors.push('Nameが未入力です');
        if (!this.email) this.errors.push('EMailが未入力です');
        else {
            const regexp = /^[A-Za-z0-9]{1}[A-Za-z0-9_.-]*@{1}[A-Za-z0-9_.-]{1,}\.[A-Za-z0-9]{1,}$/;
            if(!regexp.test(this.email))
                this.errors.push('EMail形式に誤りがあります');
        }
        if (!this.password) this.errors.push('Passwordが未入力です');
        if(this.errors.length > 0) return;

        const request = {
            name: this.name,
            email: this.email,
            password: this.password
        }
        await axios.post('/api/accounts', request)
        .then(response => {
            this.infos.push('新規作成しました。サインイン画面へ遷移します。');
            setTimeout(() => {this.$router.push("/signin")}, 2000);
        })
      }
    }
  }
</script>
```

2.5 JWT トークンを設定する

Cookie に JWT トークンが保存されているので、ToDo 画面で API を呼び出す際に取り出し、header にセットし Bearer 認証を利用します。
Bearer 認証とはトークンを利用した認証・認可に使用され、OAuth2.0 の仕様として定義されています。HTTP Header の Authorization に指定でき、‘Authorization: Bearer <token>‘の形式で指定します。

▼ pages/ToDos.vue

```
      await axios.post('/api/ToDos', this.ToDo, {
        headers: { Authorization: `Bearer ${this.$cookies.get('token')}` }
      }).then(() => {
        this.$router.app.refresh();
      })
```

第 3 章

DI コンテナ

DI コンテナとは、アプリケーションに依存性注入（Dependency Injection：以下 DI）機能を提供するフレームワークのことです。依存性注入とはソフトウェアの外部環境などに依存するデータベースへの接続や API などを切り離し、関係を疎結合にする考え方です。

現在の ToDo アプリケーションでは TodoController が TodoService を直接アクセスしていました。この構造の場合、データベースを PostgreSQL から MySQL に変えたいという場合に Controller にも影響が出てしまいます。DI コンテナを使用する場合は TodoService インタフェースを追加し、Controller はインタフェースに対してアクセスするようにします。DI コンテナは、インタフェースに対する実装機能を提供します。これにより、データベースの変更や直接実装クラスの参照がなくなり関係を疎結合にできます。

3.1 DI コンテナの導入

DI フレームワークには Koin（`https://insert-koin.io/`）を使います。Ktor には Application クラスに installKoin の拡張機能が用意されていますので簡単に導入できます。Koin とは Kotlin 用の軽量 DI フレームワークです。基本的には次のステップで容易に DI のしくみを提供してくれます。

- module の宣言
- Koin の開始
- 利用箇所で inject

▼ build.gradle

```
dependencies {
    implementation "org.koin:koin-ktor:2.1.6"
}
```

3.2 インタフェースと実装クラス

すでに実装されている TodoService は TodoServiceImpl という名前に変更し、TodoService インタフェースを新たに作成します。TodoService インタフェースには Controller クラスが呼び出しているメソッドを定義します。TodoServiceImpl クラスは TodoService インタフェースを継承し、対象メソッドをオーバーライドします。

▼ service/TodoService.kt

```
interface TodoService {
    suspend fun getAllTodos(): List<Todo>
    suspend fun getTodo(id: Int): Todo?
    suspend fun addTodo(todo: NewTodo): Todo
    suspend fun updateTodo(todo: NewTodo): Todo?
    suspend fun deleteTodo(id: Int): Boolean
}
```

▼ service/TodoServiceImpl.kt

```
class TodoServiceImpl: TodoService {
    override suspend fun getAllTodos(): List<Todo> = dbQuery {
        // 行数の都合上、処理は省略
    }

    override suspend fun getTodo(id: Int): Todo? = dbQuery {
        ・・・
    }

    override suspend fun addTodo(todo: NewTodo): Todo {
        ・・・
    }

    override suspend fun updateTodo(todo: NewTodo): Todo? {
        ・・・
    }

    override suspend fun deleteTodo(id: Int): Boolean {
        ・・・
    }
}
```

Koin は DI 定義を module 毎に分割できます。今回は ServiceModule を作成し module を定義していきます。

▼ service/ServiceModule.kt

```
val serviceModule = module {
    singleBy<TodoService, TodoServiceImpl>()
}
```

singleBy で単一インスタンスを保証するシングルトンを定義しました。サービスクラスが増えた場合は、この定義を追加するだけで DI コンテナに登録できます。

作成した ServiceModule を Ktor にインストールします。

▼ Application.kt

```
fun Application.module(testing: Boolean = false) {
    install(ContentNegotiation) {
        jackson {
            configure(SerializationFeature.INDENT_OUTPUT, true)
        }
    }
+    install(Koin) {
+        modules(serviceModule)
+    }
    DatabaseFactory.init()
    install(Routing) {
        todos()
    }
}
```

Controller で利用するには inject をするだけで利用できます。ServiceModule で定義した TodoServiceImpl のシングルトンインスタンスが inject によって代入されます。

▼ web/TodoController.kt

```
fun Route.todos() {
    val todoService by inject<TodoService>()

    route("todos") {
        get("/") {
            call.respond(todoService.getAllTodos())
        }
    }
}
```

第 4 章

ユニットテスト

ユニットテストは、プログラムを構成する小さな単位（メソッド）が正しく機能しているかどうかを検証するテストのことです。昨今のシステム開発では必ずと言ってよいほど、導入されているテストです。ユニットテストの利点は次が挙げられます。

- モジュールが結合される前にテストを行うことができるため、問題の原因や修正が容易になる
- ユニットテストコードがテストケースとなり、妥当性の高い資産を残すことができる
- 仕様変更やリファクタリングを安心して行うことができる（心理的安全性）

テスト駆動開発（Test Driven Development：TDD）と呼ばれる開発手法もあり、興味がある方は調べてみるとよいでしょう。

4.1　テストフレームワークの導入

Ktor プロジェクトのテストフレームワークは JUnit5 を利用しますので、build.gradle の依存関係（dependencies）に JUnit5 のモジュールを追加します。
JUnit5 を利用する場合、Gradle に JUnit5 を使うことを明示的に宣言する必要あるため Test タスクに useJUnitPlatform を呼ぶ必要があります。
ToDo アプリケーションではデータベースを PostgreSQL を利用していますが、ユニットテストでも PostgreSQL を利用してしまうとシステムのデータと混在してしまったり、起動に時間がかかってしまったりとデメリットが多いため、H2 データベースを利用します。
H2 データベースは Java 製でメモリ上で動作ができるデータベースです。

▼ build.gradle

```
dependencies {
+    testImplementation "io.ktor:ktor-server-tests:$ktor_version"
+    testImplementation 'org.jetbrains.kotlin:kotlin-test-junit5:1.4.20'
+    testImplementation("org.koin:koin-test:0.9.3")
+    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.7.0'
+    testRuntimeOnly 'org.junit.platform:junit-platform-launcher:1.7.0'
+    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.7.0'
+    testCompile 'com.h2database:h2":"1.4.200'
}

test {
    useJUnitPlatform()
}
```

4.2 プロジェクトディレクトリ構成の変更

ユニットテストを導入する前のディレクトリ構成はプロジェクト本体を格納している src と resources のみです。ユニットテストソースを格納するディレクトリを追加します。次のようなディレクトリ構成となります。

```
.
├── ./resources
├── ./src
├── ./test
└── ./testResources
```

testResources に application.conf を配置しておくことでユニットテスト実行の DB 接続先を変えることができますので、H2 データベースの接続情報を記載します。h2:mem と記載することでメモリ起動になり、MODE=PostgreSQL オプションを付けることで H2 の挙動を PostgreSQL モードにできます。

参考：http://www.h2database.com/html/features.html#compatibility

▼ resources/application.conf

```
db {
    jdbcUrl = "jdbc:h2:mem:todo;MODE=PostgreSQL;DATABASE_TO_LOWER=TRUE"
    dbUser = sa
    dbPassword = password
    driverClass = org.h2.Driver
}

flyway {
    url = "jdbc:h2:mem:todo;MODE=PostgreSQL;DATABASE_TO_LOWER=TRUE"
    user = sa
    password = password
    baselineOnMigrate=true
    locations = ["filesystem:resources/db/migration"]
```

```
}
```

これでユニットテストを行う準備が整いました。

4.3 テストクラスの作成

4.3.1 Controller のテスト

Controller のユニットテストは ktor-server-tests の TestApplicationEngine を利用します。
TestApplicationEngine のインスタンスを作成して handleRequest で API をコールしレスポンスを検証します。
前章で DI コンテナを導入したことで、Service クラスをモック化できます。モックとは実際の処理ではなく、戻り値をあらかじめ決めておくことができる機能になります。これにより、ユニットテストの実行スピードやテスト対象のテストがしやすくなります。

TodoServiceImpl をモック化するために、TodoService を継承した MockTodoService クラスを作成します。MockTodoService クラスの特徴はコンストラクタで受け取った値をメソッドの戻り値に設定しているところで、これにより getAllTodos メソッドは必ず想定した結果を返却します。

▼ test/service/MockTodoService.kt

```
class MockTodoService(
    private val todos: List<Todo>
): TodoService {
    override suspend fun getAllTodos(): List<Todo> {
        return todos
    }
    override suspend fun getTodo(id: Int): Todo? {
        return null
    }
    override suspend fun addTodo(todo: NewTodo): Todo {
        return Todo(3, "test3")
    }
    override suspend fun updateTodo(todo: NewTodo): Todo? {
        return null
    }
    override suspend fun deleteTodo(id: Int): Boolean {
        return true
    }
}
```

Ktor の API テストは TestApplicationEngine のインスタンスを生成し、handleRequest でエンドポイントを評価する形となります。

@Before アノテーションは付与されたメソッドが各テストメソッドの実行する前に都度実行してくれるようにする機能で、@ Test はユニットテストメソッドを示すアノテーションになります。
TestApplicationEngine.apply 内でモックモジュールを呼び出しています。これにより TestApplicationEngine でモック定義したモジュールが有効になります。

with(engine) を用いることで TestApplicationEngine をレシーバとしてメソッドを呼び出すことができ、handleRequest 経由で API を実行しています。

（1）は「/todos」エンドポイントのレスポンスコードが 200 であることを評価しています。
（2）は返却結果 JSON が想定した結果になっているか評価しています。モックにしていることで返却される JSON が想定できるようになっています。

なお、テストメソッドは「タスク一覧を呼び出すと 200 が変えること」といった形で日本語にしておくと、どんなテストを行っているのかわかりやすくなります。

▼ test/web/TodoControllerTest.kt

```
class TodoControllerTest {
    private lateinit var mapper: ObjectMapper

    @Before
    fun before() {
        mapper = jacksonObjectMapper()
        mapper.enable(SerializationFeature.INDENT_OUTPUT)
    }

    @Test
    fun タスク一覧を呼び出すと200が変えること() {
        val todos = listOf(Todo(1, "test1"), Todo(2, "test2"))
        val mockModule: Module = module {
            single{ MockTodoService(todos) as TodoService }
        }

        val engine: TestApplicationEngine = TestApplicationEngine().apply {
            start(wait = false)
            application.module(true, mockModule)
        }

        with(engine) {
            handleRequest(HttpMethod.Get, "/todos").response.apply {
                assertEquals(HttpStatusCode.OK, status())   // (1)
                assertEquals(mapper.writeValueAsString(todos), content) // (2)
            }
        }
    }
```

```
}
```

4.3.2 Service のテスト

TodoServiceImpl クラスのユニットテストです。H2 データベースにテスト用データを投入し想定した結果が得られるか評価しています。Controller では TodoServiceImpl クラスをモック化したことでテストできていないので、H2 データベースに接続してテストを行います。

▼ test/service/TodoServiceImplTest.kt

```
class TodoServiceImplTest {
    private val serviceImpl: TodoServiceImpl = TodoServiceImpl()

    @Before
    fun setup() {
        DatabaseFactory.init()
    }

    @Test
    fun タスクが5件取得できること() {
        runBlocking {
            repeat(5) {
                DatabaseFactory.dbQuery {
                    Todos.insert { it[task] = "test$it" }
                }
            }

            val todos = serviceImpl.getAllTodos()
            Assert.assertEquals(todos.size, 5)
        }
    }
}
```

Controller テストはモジュールをロードした段階でマイグレーションが起動していましたが、Service テストでは setup メソッドでデータベースマイグレーションを実行しています。

‘repeat(5)‘でテストデータを 5 件登録しています。タスクを全件取得する getAllTodos を実行し、得られた結果が 5 件であることを評価しています。

第 5 章

ドメイン駆動設計

ドメイン駆動設計（以下 DDD）とはエリック・エヴァンズが提唱した、ユーザーーが従事する業務に合わせてソフトウェアを開発する設計手法です。次のような特徴があります。

- ソフトウェアで解決しようとする領域（ドメイン）を中心にとらえて開発を行う
- ドメイン知識がそのままソースコードに反映される
- ドメイン知識とソースコードが地続きになることで、変化に強いソフトウェアになる

DDD はソフトウェア設計手法の 1 つで開発対象の問題を解決することを目的としており、この開発対象のことをドメインと呼びます。ドメインモデリングを行うことでソフトウェアを必要としている人との共通認識が取りやすく、フィードバックが得やすくなります。
ソフトウェアを必要としている（開発対象に詳しい人）のことを「ドメインエキスパート」と呼び、ドメインエキスパートと開発者が同じ呼称をするために「ユビキタス言語」を作成して会話をしていくことで認識の齟齬が起きないようにします。
詳しくは次の書籍やサイトを参考にしてみてください。

- エリック・エヴァンスのドメイン駆動設計（https://www.amazon.co.jp/exec/obidos/ASIN/4798121967/hatena-blog-22/）
- ドメイン駆動設計　モデリング/実装ガイド（https://little-hands.booth.pm/items/1835632）

今まで実装した ToDo アプリケーションをドメインモデリングを行い、クリーンアーキテクチャへリファクタリングを行っていきます。私自身まだまだ勉強中で私的な解釈もありますがご了承ください。

5.1 ドメインモデリング

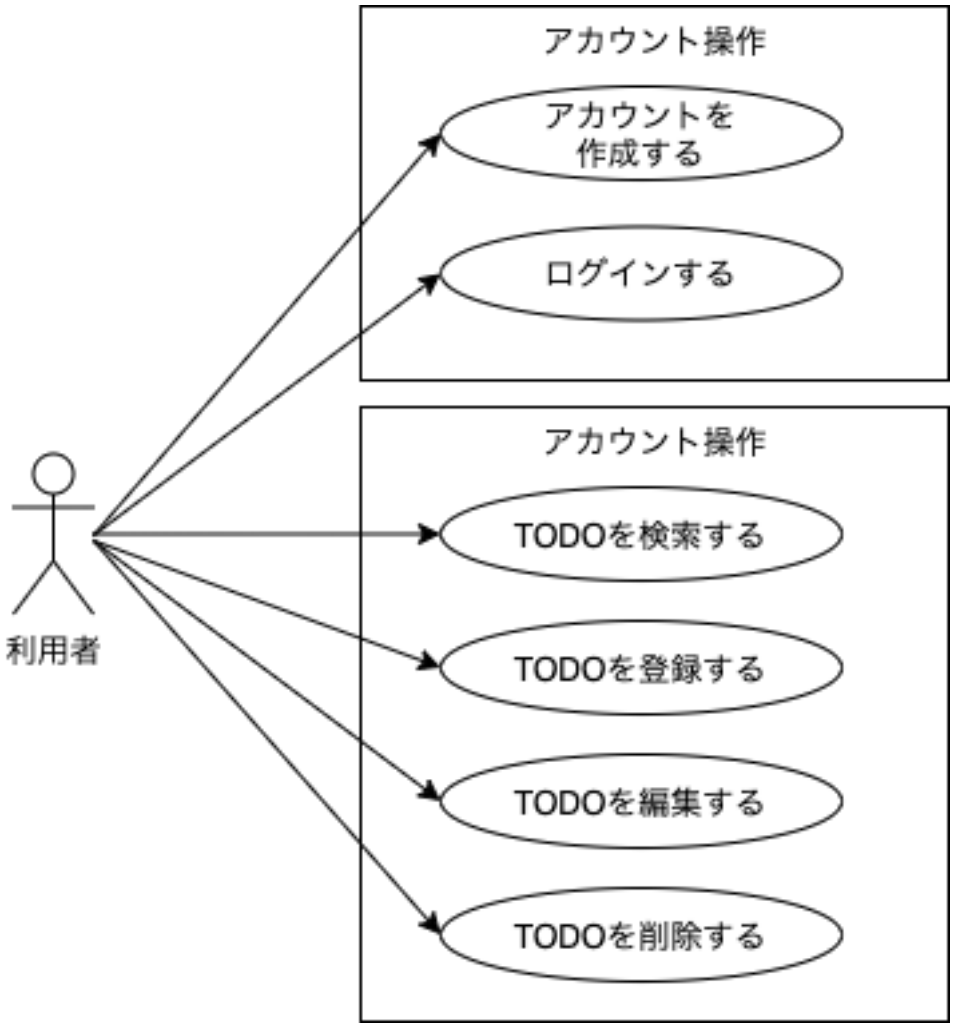
ドメインモデリングとは、システムに関する業務知識を整理して、関係性を明確にすることを目的としています。
モデリングをすることでドメインに対する知識を深めることができ、開発者にドメインに対する認識を合わせることができます。
具体的に何をしていくかというと、まずは業務を知ることから始めます。

1. マニュアルを読んでみる
2. 書籍などで一般的な業務知識を得る
3. 業務経験者（ドメインエキスパート）と会話する

業務をある程度理解できたら、ユースケース図やドメインモデル図を書いていきます。
ドメインモデリングは完成という定義がなく常に変化し続ける業務を追って、常に進化させていくため正確性を求めず・細かく定義しないことが重要です。

5.1.1 ユースケース図

ToDo アプリケーションではどのようなユースケースが考えられるでしょうか。
「ユーザーがアカウントを作成」「ユーザーがログインする」といった認証ユースケースと「ToDo を検索する」「ToDo を登録する」といった ToDo 操作のユースケースが考えられます。それらをユースケース図を使ってシステムの振る舞いを定義します。
本システムでは利用者属性が１つのため「ユーザー」と表記していますが、「一般ユーザー」や「管理者ユーザー」といった属性をアクターに表現するとよいでしょう。



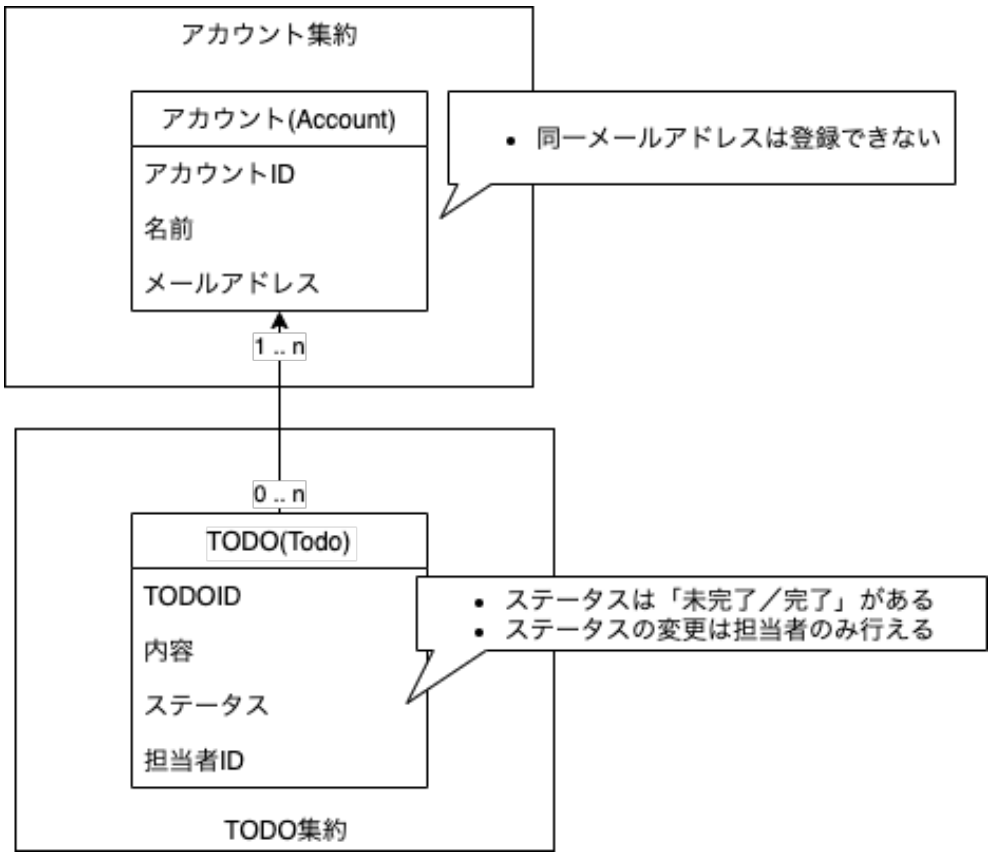
▲図 5.1　ユースケース図

すでに実装済みのユースケースのみを記載していますが、機能追加する際はユースケース図も合わせて修正しておくのがよいでしょう。

5.1.2　ドメインモデル図

ドメインモデル図はクラス図にようなもので、ルールや成約（ドメイン知識）、オブジェクトどうしの関係といった内容を記述していきます。ドメインモデル図を作成する際に決

めておきたいのが集約です。集約とはオブジェクトのまとまりを表し、整合性を保ちながらデータを更新する単位となります。次はユースケース図から作成したドメインモデル図になります。



▲図 5.2　ドメインモデル図

アカウント集約と ToDo 集約の 2 つあり、ToDo は担当者 ID でアカウントとつながっています。また、吹き出しでルール・制約を記述してあることでオブジェクトの状態が明確になっています。

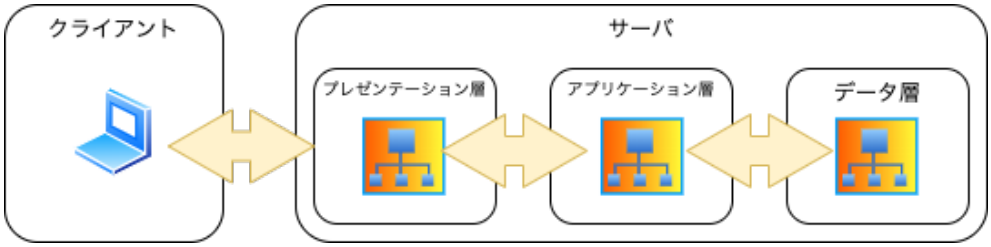
5.2　アーキテクチャ

前節で作成したドメインモデルをプログラムに反映していきます。実装を始める前にアーキテクチャを決める必要があります。リファクタリング前に採用していたアーキテクチャは「3 層アーキテクチャ」で、リファクタリング後は昨今人気がある「クリーンアーキテクチャ」を利用します。

5.2.1　3 層アーキテクチャ

Web アプリケーションでよく利用されるアーキテクチャです。3 層アーキテクチャはサーバシステムを次の 3 階層に分割したシステムのことを言います。

- プレゼンテーション層
- アプリケーション層
- データ層



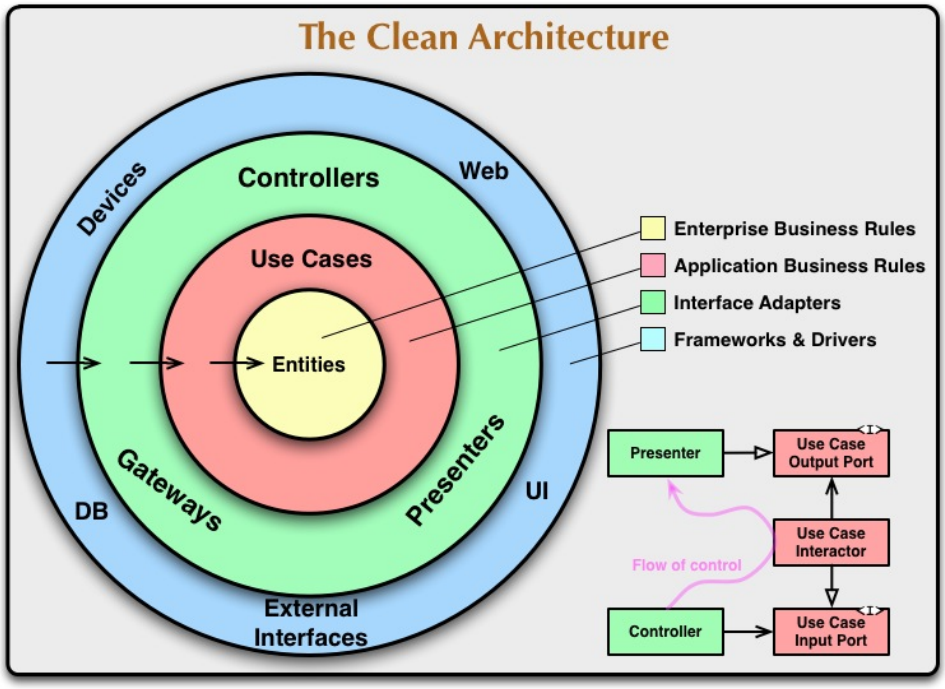
▲図 5.3　3 層アーキテクチャ

プレゼンテーション層は、ブラウザから受け取った供給をもとにページ情報を返す役割を担っており、画面表示処理に特化した層となります。アプリケーション層は開いたページの検索ボックスに入力したキーワードから情報を検索するといった処理を行う役割を担います。ロジック層と呼ばれたりしています。データ層はページを表示するための情報が詰め込まれたデータベースとの入出力する層となります。

5.2.2　クリーンアーキテクチャ

クリーンアーキテクチャは Robert C. Martin(Uncle Bob) が 2012 年に提唱した独立性を確保するためのアーキテクチャで、次の図がよく用いられます。次の図を見かけたこ

とがある人は多いかと思います。



▲図 5.4　クリーンアーキテクチャ

エンティティ：Enterprise Business Rules

円の中心に位置している層がエンティティとされており、ビジネスルールを表現するための層となっています。ドメイン層に相当する部分で、ドメインモデルに最も影響する箇所です。

ユースケース：Application Business Rules

円の中心から 2 番目に位置する層がユースケースとされており、エンティティに存在するビジネスルールを組み合わせてアプリケーションが必要する処理を行います。

コントローラ、プレゼンター、ゲートウェイ：Interface Adapters

円の中心から 3 番目に位置する層が Interface Adapters とされており、前後に挟まれた Frameworks&Drivers と Application Business Rules とのレイヤの橋渡しをする層です。Gateway はデータを抽象化する Repository などで、Presenter は View に適した形に変換します。Controller はブラウザから受け取った情報を UseCase に渡すといった役割を持っています。

デバイス、ウェブ、プレゼンター、外部インタフェース、DB:Frameworks & Drivers

一番外側の層は Frameworks & Drivers とされ、フレームワークや DB、フロントエンドの UI などを配置するための層です。Ktor に依存している実装はこのレイヤに閉じ込めます。ほかのレイヤでは Ktor に依存した作りをしないように注意します。Ktor サーバを立ち上げる Application.kt も infrastructure ディレクトリに格納します。

矢印

円の左側に矢印が明記されています。この矢印は依存の方向性を表しています。外側の層から内側の層への依存関係が許可されていることを示しています。

5.3 ドメインモデルの実装

ドメインオブジェクトの実装の前にクリーンアーキテクチャの各層に合わせてパッケージ構成を分けます。

```
.
└── com
    └── todo
        └── example
            ├── domain  // Enterprise Business Rules(エンティティ)
            ├── infrastructure  // Frameworks & Drivers(デバイス、ウェブ、プレゼ
ンター、外部インタフェース、DB)
            │   ├── dao
            │   ├── framework
            │   └── repositoryimpl
            ├── interfaces  // Interface Adapters(コントローラ、プレゼンター、ゲー
トウェイ)
            │   ├── controller
            │   ├── model
            │   ├── presenter
            │   ├── repository
            │   └── Application.kt
            └── usecase  // Application Business Rules(ユースケース)
              ├── dto
```

```
                └──── impl
```

パッケージ 役割
domain ：ドメインモデルを管理する
interfaces.controller ：フロントエンドから呼ばれる API のエンドポイントクラス
interfaces.model ：リクエスト／レスポンス JSON データを格納する data class
interfaces.presenter ：REST API の Response 形式に変換
interfaces.repository ：Repository インタフェース
infrastructure.dao ：DB アクセス Dao クラス
infrastructure.framework ：フレームワーク（Ktor）に依存するコード
infrastructure.repositoryimpl ：Repository インタフェースの実装クラス
usecase ：UseCase インタフェース
usecase.impl ：UseCase インタフェースの実装クラス
usecase.dto ：ユースケース層から戻り値クラス（Data Transfer Object）

続いてパッケージごとに主要なファイルについて解説します。完成したソースは GitHub にありますので詳しくはそちらを参照してください。

5.3.1 infrastructure

framework

Ktor や Koin、Exposed といったフレームワークに依存している実装がまとめられています。

- Ktor が下位互換のないバージョンアップが起きてしまった
- フレームワークを別のものに切り替える

などといった変更が起きた場合にこのレイヤだけを変更すれば実現でき、他レイヤに影響を及ぼすことは最小限にできます。

JWT 認証、DI コンテナ、ユニットテスト導入と拡張された Application.kt になります。

▼ Application.kt

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

```
    install(Koin) {
        if(testing && mockModule != null) modules(mockModule)
        else modules(koinModule)
    }

    val jwtConfig = JWTConfig()
    install(Authentication) {
        jwt {
            realm = jwtConfig.realm
            verifier(jwtConfig.makeJwtVerifier())
            validate {credential ->
                if (credential.payload.audience.contains(jwtConfig.audience)) {
                    val principal = JWTPrincipal(credential.payload)
                    AuthUser.toAuthUser(principal)
                    principal
                } else null
            }
        }
    }

    DatabaseFactory.init()

    install(Routing) {
        todos()
        accounts()
    }
}
```

KoinModule.kt では DI コンテナで管理設定が実装されています。依存性逆転の法則を実現させるための大事な実装になります。

▼ KoinModule.kt

```
val koinModule = module {
    // Controller
    single { AccountController(get(), get()) }
    single { TodoController(get(), get()) }

    // Presenter
    single { AccountPresenter() }
    single { TodoPresenter() }

    // UseCase
    single<AccountUseCase> { AccountUseCaseImpl(get()) }
    single<TodoUseCase> { TodoUseCaseImpl(get(), get()) }

    // Repository
    singleBy<TodoRepository, TodoRepositoryImpl>()
    singleBy<AccountRepository, AccountRepositoryImpl>()
}
```

Controller と密実装となったいたルーティングを切り出しています。 ‘by inject()‘ を呼び出して Controller を DI コンテナから取得します。リクエスト JSON を Controller へ渡す役割を担っています。

▼ Route.kt

```
fun Route.accounts() {
    route("accounts") {
        val controller: AccountController by inject()

        post("/") {
            val newAccount = call.receive<NewAccountRequest>()
            call.respond(HttpStatusCode.Created, controller.createAccount(newAccount))
        }

        post("/auth") {
            val login = call.receive<LoginRequest>()
            val result = controller.authentication(login)
            if(result == null) call.respond(HttpStatusCode.Forbidden)
            else call.respond(HttpStatusCode.OK, result)
        }
    }
}

fun Route.todos() {
    val controller by inject<TodoController>()

    route("todos") {
        authenticate {
            ・・・
        }
    }
}
```

dao

テーブル定義を表現した object クラスのみが定義されています。「認証」章で作成したクラスと同じ実装です。

▼ Accounts.kt

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

repositoryimpl

実際に SQL を発行したりする Repository の実装クラスです。こちらはリファクタリング前の Service クラスと同じ実装になります。

```
class AccountRepositoryImpl: AccountRepository {
    override suspend fun findByAccountId(id: Int): Account? = dbQuery {
        Accounts.select {
            (Accounts.id eq id)
        }.mapNotNull { convertAccount(it) }
            .singleOrNull()
    }

    override suspend fun findByEmail(email: String): Account? = dbQuery  {
        Accounts.select {
            (Accounts.email eq email)
        }.mapNotNull { convertAccount(it) }
            .singleOrNull()
    }

    override suspend fun createAccount(account: Account, passwd: String): Account {
        var key = 0
        dbQuery {
            key = (Accounts.insert {
                it[password] = createHash(passwd)
                it[name] = account.name
                it[email] = account.email
            } get Accounts.id)
        }
        return findByAccountId(key)!!
    }

    override suspend fun authentication(email: String, password: String): Account? = dbQuery {
        Accounts.select {
            (Accounts.email eq email) and (Accounts.password eq createHash(password))
        }.mapNotNull { convertAccount(it) }
            .singleOrNull()
    }

    internal fun createHash(value: String): String {
        return MessageDigest.getInstance("SHA-256")
            .digest(value.toByteArray())
            .joinToString(separator = "") {
                "%02x".format(it)
            }
    }

    private fun convertAccount(row: ResultRow): Account =
        Account.reconstruct(row[Accounts.id], row[Accounts.name], row[Accounts.email],)
}
```

5.3.2 interfaces

model

リクエストやレスポンスの JSON データを管理する data object クラスが実装されています。Controller のメソッド引数や戻り値として利用されます。

▼ AccountRequest.kt

```
data class LoginRequest(
    val email: String,
    val password: String,
)

data class NewAccountRequest(
    val id: Int?,
    val password: String,
    val name: String,
    val email: String,
)
```

▼ AuthResponse.kt

```
data class AuthResponse(
    @JsonProperty("bearer_token")
    val token: String
)
```

controller

HTTP 通信で JSON などの情報を受け取りや SESSION の操作を行ったり、リクエスト JSON のバリデーションもここで行います。

```
class AccountController(
    private val useCase: AccountUseCase,
    private val presenter: AccountPresenter,
) {

    suspend fun createAccount(newAccount: NewAccountRequest): AuthResponse {
        val account = useCase.createAccount(newAccount)
        return presenter.toAuthResponse(account) ?: throw Exception("アカウントの
作成に失敗しました")
    }

    suspend fun authentication(loginRequest: LoginRequest): AuthResponse? {
        val account = useCase.authentication(loginRequest)
        return presenter.toAuthResponse(account)
    }
}
```

AccountUseCase インタフェースと AccountPresenter クラスを Koin の依存性注入でインスタンスを取得しています。リファクタリング前と比較しますとフレームワーク（Ktor）の機能を一切使用していないので責務が明確になっていて可読性も上がっています。

presenter

ユースケース層から返却された DTO をレスポンスへ変換することを目的としています。

```
class AccountPresenter {
    private val jwtConfig = JWTConfig()

    fun toAuthResponse(account: Account?): AuthResponse? {
        return if(account == null) null
        else {
            val token = jwtConfig.createToken(account.accountId!!.raw)
            AuthResponse(token)
        }
    }
}
```

repository

データ操作を定義したインタフェースが定義されています。ユースケース層から利用されます。

▼ AccountRepository.kt

```
interface AccountRepository {
    suspend fun findByAccountId(id: Int): Account?
    suspend fun findByEmail(email: String): Account?
    suspend fun createAccount(account: Account, passwd: String): Account
    suspend fun authentication(email: String, password: String): Account?
}
```

5.3.3 usecase

ユースケース図で定義した振る舞いが実装された箇所で、ビジネスロジックが実装されます。DI コンテナを使うことでデータベース層に依存していないところがポイントです。データベース層に依存していないことで外部サービスの変更に感知しないようにします。

ユースケースの振る舞いを定義したインタフェースです。

▼ AccountUseCase.kt

```
interface AccountUseCase {
    suspend fun createAccount(newAccount: NewAccountRequest): Account
    suspend fun authentication(loginRequest: LoginRequest): Account?
}
```

impl

ユースケースインタフェースの実装クラスです。依存性逆転の法則を守るために Repository インタフェースに依存しており、実装クラスには依存していません。トランザクションの制御もユースケース層で行います。transaction で囲むことでトランザクション境界を作ることができます。

```
class AccountUseCaseImpl(
    private val accountRepository: AccountRepository
): AccountUseCase {

    override suspend fun createAccount(newAccount: NewAccountRequest): Account {
        val duplicate = accountRepository.findByEmail(newAccount.email)
        if(duplicate != null) throw IllegalArgumentException("既に登録済みのメール
アドレスです")

        val account = Account.createAccount(newAccount.name, newAccount.email)
        return accountRepository.createAccount(account, newAccount.password)
    }

    override suspend fun authentication(loginRequest: LoginRequest): Account? =
        accountRepository.authentication(loginRequest.email, loginRequest.password)
}
```

メールアドレスの重複チェックはドメインモデリングで定義したドメインルールになります。本来であればドメインサービスを作成しチェックメソッドを作成するところですが、ユースケース層はビジネスロジックが存在する箇所としてこの層に実装しています。

dto

ユースケース層から返却されるデータ形式を定義した data object クラスです。ドメインオブジェクトをそのまま返却しまうとドメインオブジェクトが想定外の利用をされてしまう可能性があるため、DTO を用意します。

▼ TodoDto

```
data class TodoDto(
    val id: TodoId,
    val task: String,
    val status: Status,
    val personName: String,
)
```

5.3.4 domain

クリーンアーキテクチャでは Entities に相当します。ドメインモデリングで行った内容が反映され UseCase と Repository 間でやりとりが行われます。データの設定が static メソッドのみなので不整合なデータが作成できないようにしています。専用 ID クラスを用意することで、誤った値の混入を防いだり採番処理を統一したりするメリットがあります。

▼ AccountId.kt

```
inline class AccountId(val raw: Int)
```

▼ Account.kt

```
package com.todo.example.domain.account

class Account private constructor(
    val accountId: AccountId?,
    var name: String,
    var email: String
) {
    companion object {
        fun reconstruct(id: Int, name: String, email: String): Account = Account(AccountId(id), name,

        fun createAccount(name: String, email: String): Account = Account(null, name, email)
    }
}
```

5.4 まとめ

本章では DDD の基本的な概要と設計方法、クリーンアーキテクチャへのリファクタリングを行いました。
アーキテクチャを一新したことで、ディレクトリ構成がガラッと変わり違和感を感じる方もいるかもしれませんが、依存関係がすっきりし各層が責務に専念できることでユニットテストも書きやすく変更もしやすくなりました。
これらのメリットは大規模になればなるほど威力を発揮しますし、Ktor + Koin はクリーンアーキテクチャを表現しやすく実運用に十分耐えられるように感じました。

終わりに

技術書典に数ある書籍の中から本書を選んでいただきありがとうございます。
長野県の田舎でサーバサイドエンジニアをしているてっしーと申します。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。

ちなみにこれを書いているのは技術書典開催の 2 日前です。皆さんが本書を手に取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ています。紙書籍も頒布したかったのですが、諸事情で断念しました。紙がほしかった方がいらっしゃいましたらごめんなさい。

本書は技術書典 9 で頒布した「Ktor と Nuxt.js で作る Web アプリケーション入門」の第 2 弾でより実績的な内容となっています。これからエンジニアになろうと考えている方がポートフォリオを作成する際にお役に立てればと思い執筆しました。
ドメイン駆動設計やクリーンアーキテクチャは筆者も勉強しながら執筆させていただいたので、言葉足らずや誤りがあったかもしれません。優しくフィードバックしていただけると幸いです。

このシリーズは本書を持って完結となります。次回以降の内容はまだ全然考えられていませんが、頒布できるよう頑張っていきたいと思います。ではでは〜

Ktor と Nuxt.js で作る Web アプリケーション入門 Vol.2

2020 年 12 月 26 日　技術書典 10 v1.0.0

著　者　てっしー
編　集　てっしー
発行所　crescent