〇Springを元にしたWebで必要な技術要素

- DI

- プロファイル サブシステムの識別子

- MVC構成

- データベースアクセス

- トランザクション管理

- Webサーバー

- HTMLなどの画面製作

- Restful Webサービス

- Restful Webサービスの呼び出し

- セキュリティ・認証・認可

- シングルトンとスレッドセーフ(マルチスレッド)

- 例外処理

- セッションスコープ

- フラッシュスコープ

- テストサポート 単体テスト・結合テスト・総合テスト

〇jarの作成と配置

エクスポートしたいプロジェクトのメインクラスを持つJavaファイルを右クリックします。

- 「実行」>「実行構成...」を選択します。

- 左側のリストで「Java アプリケーション」をダブルクリックするか、「新規」ボタンを押して新しい構成を作成します。

- 「名前」に適切な名前を入力します。

- 「プロジェクト」にエクスポートしたいプロジェクトを選択します。

・set SPRING\_PROFILES\_ACTIVE=staging　or production を設定

・java -jar 0451-training-profile.jar で実行

〇DI(Dependency Injection)

・DIしているサービスには@Service、リポジトリには@Repository、mainには@Configurationと@ComponentScanが必要

・下記のように指定してコンポーネントスキャンを行う。

@Configuration

@ComponentScan

public class ShoppingApplication {

public static void main(String[] args) {

ApplicationContext context = new AnnotationConfigApplicationContext(ShoppingApplication.class);

OrderService orderService = context.getBean(OrderService.class);

・Bean定義では、メソッドに@Beanを指定して一部のメソッドをBeanとしてDIコンテナで管理することが可能。

開発者が作成したクラスには、ステレオタイプアノテーションを付加する。ライブラリのクラスを作成する場合は(DataSource＠など)@Beanメソッドを定義する

・JavaConfigのクラスにも@Profileの指定が可能。@Beanメソッドと共に@Profileの指定も可能。

〇JPA (Java Persistence API) と Hibernate の関係

JPA (Java Persistence API) は、Java におけるオブジェクトとリレーショナルデータベースとの永続化を扱うための 仕様 (Specification) です。これは、データの保存、読み出し、更新、削除といった操作を、Java のオブジェクトを通して行うための共通のインターフェースや規約を定めています。

Hibernate は、この JPA 仕様を実装する ORM (Object-Relational Mapping) フレームワーク の一つです。つまり、JPA が定めたルールやインターフェースに従って、実際にデータベースとのやり取りを行う機能を提供しています。

〇[message.properties](https://message.properties/)例.

spring.datasource.url=jdbc:postgresql://localhost:5432/postgres

spring.datasource.username=tetsuzo

spring.datasource.password=@Engineers01

spring.datasource.driver-class-name=org.postgresql.Driver

〇Postgresセッティング

Installation Directory: C:\Program Files\PostgreSQL\17

Server Installation Directory: C:\Program Files\PostgreSQL\17

Data Directory: C:\Program Files\PostgreSQL\17\data

Database Port: 5432

Database Superuser: postgres

Operating System Account: NT AUTHORITY\NetworkService

Database Service: postgresql-x64-17

Command Line Tools Installation Directory: C:\Program Files\PostgreSQL\17

pgAdmin4 Installation Directory: C:\Program Files\PostgreSQL\17\pgAdmin 4

Stack Builder Installation Directory: C:\Program Files\PostgreSQL\17

Installation Log: C:\Users\Tetsuzo1\AppData\Local\Temp\install-postgresql.log

サーバー名PostgreSQL 17

ホスト名PostgreSQL 17

ポート5432

データベース名postgres

ユーザー名/PWD postgres/postgres >> scram-sha-256で認証エラー。tetsuzo/@Engineers01で認証OK。

PostgreSQLにおける\*\*スキーマ (Schema)\*\* は、データベースオブジェクト（テーブル、ビュー、インデックス、関数など）を論理的にグループ化するための仕組みです。名前空間のような役割を果たし、同じデータベース内で異なる名前のオブジェクトが衝突するのを防ぎ、整理された構造を提供します。

スキーマの作成: CREATE SCHEMA コマンドを使用して新しいスキーマを作成できます。

SQL

CREATE USER tetsuzo WITH SUPERUSER PASSWORD '@Engineers01';

ALTER USER tetsuzo WITH CREATEDB;

SHOW search\_path;

CREATE SCHEMA tetsuzo;

CREATE SCHEMA sales;スキーマの指定: オブジェクトにアクセスする際には、スキーマ名を明示的に指定できます。

SQL

SELECT \* FROM sales.customers;

また、SET search\_path コマンドを使用して、検索パスを設定することで、スキーマ名を省略してオブジェクトにアクセスできるようになります。検索パスに複数のスキーマを設定した場合、指定された順にオブジェクトが検索されます。

SQL

SET search\_path TO sales, public;

SELECT \* FROM customers; -- まず 'sales' スキーマ、次に 'public' スキーマの'customers' テーブルを探す

テーブルスペースの設定も可能

・CREATE　TABLESPACE my\_tablespace LOCATION '/path/to/my/tablespace';

・postgresへの接続

psql -U postgres -h localhost -d postgres

・create table xxxを実行すると、schemaのデフォルトのpublic内にテーブルが作成される

〇組み込みデータベース。EmbeddedDatabase。

・メモリ上の仮のデータベース。インストール不要。

@Bean

public DataSource dataSource() {

EmbeddedDatabase dataSource = new EmbeddedDatabaseBuilder()

.addScripts("schema.sql", "data.sql")

.setType(EmbeddedDatabaseType.H2).build();

return dataSource;

}

〇MaveのPostgresの設定

Mavenのdependency設定では、\*\*PostgreSQLにアクセスするためのJDBCドライバ\*\*を指定します。

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

</dependency>

〇接続プーリング

1. Apache Commons DBCPを使用する方法 (接続プーリング)

2. HikariCPを使用する方法 (高性能な接続プーリング)

<dependency>

<groupId>com.zaxxer</groupId>

<artifactId>HikariCP</artifactId>

</dependency>

〇Spring JDBCの設定

- mainメソッドのあるクラス

@Bean

public JdbcTemplate jdbcTemplate(DataSource dataSource) {

return new JdbcTemplate(dataSource);

}

@Bean

public DataSource dataSource() {

HikariDataSource dataSource = new HikariDataSource();

dataSource.setDriverClassName("org.postgresql.Driver");

dataSource.setJdbcUrl("jdbc:postgresql://localhost:5432/postgres");

dataSource.setUsername("tetsuzo");

dataSource.setPassword("@Engineers01");

return dataSource;

}

〇Repositoryでの設定

private final JdbcTemplate jdbcTemplate;

public JdbcProductRepository(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

・Select

@Override

public Product selectById(String id) {

Product product = jdbcTemplate.queryForObject("SELECT \* FROM t\_product WHERE id=?",

new DataClassRowMapper<>(Product.class),id);

return product;

}

・Insert

@Override

public void insert(Order order) {

jdbcTemplate.update("INSERT INTO t\_order values (?, ?, ?, ?, ?, ?, ?, ?)",

order.getId(),

order.getOrderDateTime(),

order.getBillingAmount(),

order.getCustomerName(),

order.getCustomerAddress(),

order.getCustomerPhone(),

order.getCustomerEmailAddress(),

order.getPaymentMethod().toString());

}

・Update

@Override

public boolean update(Product product) {

int count = jdbcTemplate.update("UPDATE t\_product SET name=?, price=?, stock=? WHERE id=?",

product.getName(),

product.getPrice(),

product.getStock(),

product.getId()

);

if (count == 0) {

return false;

}else {

return true;

}

}

〇logback

"C:\Eclipse202212\pleiades-2025-03-ultimate-win-64bit-jre\_20250319\workspace\.metadata\.plugins\org.eclipse.m2e.logback\logback.2.7.0.20241001-1350.xml"

"C:\Eclipse202212\pleiades-2025-03-ultimate-win-64bit-jre\_20250319\workspace.metadata.bak\_0.log"

にログは一応は出力されているようだが、内容はもう一つのよう。

log4jに切り替えることもあるが、log出力はコンソールで見て、ファイルには残さない方が良さそう。

〇トランザクション管理 Transactional

- ServiceでのTransactional定義

@Service

@Transactional

public class ReservationServiceImpl implements  ReservationService {

〇mainメソッドを含んだアプリケーションでのPlatformTransactionManagerなどの@Bean定義

@Configuration

@ComponentScan

@EnableTransactionManagement

public class TrainingApplication {

@Bean

public DataSource dataSource() {

EmbeddedDatabase dataSource = new EmbeddedDatabaseBuilder()

.addScripts("schema.sql", "data.sql")

.setType(EmbeddedDatabaseType.H2).build();

return dataSource;

}

@Bean

public JdbcTemplate jdbcTemplate(DataSource dataSource) {

return new JdbcTemplate(dataSource);

}

@Bean

public PlatformTransactionManager transactionManager(DataSource dataSource) {

return new JdbcTransactionManager(dataSource);

}

public static void main(String[] args) {

・・・

〇ソースのリポジトリ

- パス

プロジェクト名/src/main/javaの下に、/com/example/training or /shoppingがあり、その下に/entity or /service or mainのアプリケーション、がある。他に、プロジェクト名/src/main/resoucesやプロジェクト名/targetがある。

- クラスパス

/src/main/java/[com.example.shoppingの下に](https://com.example.xn--shopping-ve4g3a5054f/)、classファイルが置かれる。

/src/main/resourcesの下に、application.propertiesファイルなどが置かれる。

〇Spring-boot

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jdbc</artifactId>

</dependency>

など。

spring-boot-starter-jdbc

spring-boot-starter-web

spring-boot-starter-test

●YAML形式

●JSON形式

●SpringBootApplicationの定義

@SpringBootApplication

public class TrainingApplication {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(TrainingApplication.class,args);

〇application.propertiesの例

- データソース (例: PostgreSQL)

spring.datasource.url=jdbc:postgresql://localhost:5432/postgres?currentSchema=public

spring.datasource.username=tetsuzo

spring.datasource.password=xxx

spring.datasource.driver-class-name=org.postgresql.Driver

●Spring MVC + Thymeleaf

○pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jdbc</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

○列挙体

public enum PaymentMethod {

BANK, CONVENIENCE\_STORE

}

orderInput.setPaymentMethod([PaymentMethod.BANK](https://paymentmethod.bank/));

〇boot main

@SpringBootApplication

public class ShoppingApplication {

public static void main(String[] args) {

SpringApplication.run(ShoppingApplication.class, args);

}

}

〇Conrollerクラスの作成

@Controller

@RequestMapping("/order")

public class OrderController {

private final OrderService orderService;

public OrderController(OrderService orderService) {

this.orderService = orderService;

}

@GetMapping("/display-form")

public String displayForm(Model model) {

OrderInput orderInput = new OrderInput();

orderInput.setPaymentMethod(PaymentMethod.BANK);

model.addAttribute("orderInput", orderInput);

return "order/orderForm";

}

@PostMapping("/validate-input")

public String validateInput(

OrderInput orderInput, Model model) {

CartInput cartInput = dummyCartInput();

orderService

.placeOrder(orderInput, cartInput);

model.addAttribute("cartInput", cartInput);

return "order/orderConfirmation";

}

〇Serviceクラスの作成

@Service

public class OrderServiceImpl implements OrderService {

private final OrderRepository orderRepository;

private final OrderItemRepository orderItemRepository;

private final ProductRepository productRepository;

public OrderServiceImpl(OrderRepository orderRepository, OrderItemRepository orderItemRepository, ProductRepository productRepository) {

this.orderRepository = orderRepository;

this.orderItemRepository = orderItemRepository;

this.productRepository = productRepository;

}

@Override

public Order placeOrder(OrderInput orderInput, CartInput cartInput) {

// Repositoryに渡す注文オブジェクトを生成

Order order = new Order();

・・・

〇Repositoryクラスの作成

@Repository

public class JdbcOrderRepository implements OrderRepository {

private final JdbcTemplate jdbcTemplate;

public JdbcOrderRepository(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

@Override

public void insert(Order order) {

jdbcTemplate.update("INSERT INTO t\_order values (?, ?, ?, ?, ?, ?, ?, ?)",

order.getId(),

order.getOrderDateTime(),

〇HTMLの作成。一覧出力

<h1>商品一覧</h1>

<table>

<tr>

<th>商品名</th>

<th>価格</th>

<th>在庫数</th>

</tr>

<tr th:each="prod:${productList}">

<td><a th:href="@{/catalog/display-details(productId=${prod.id})}"><span th:text="${prod.name}"></span></a></td>

<td><span th:text="${prod.price}"></span>円</td>

<td><span th:text="${prod.stock}"></span></td>

</tr>

</table>

〇HTMLの作成。画面からの入力

<h1>注文確認</h1>

<form th:action="@{/order/place-order}" , method="post" th:object="${orderInput}">

<table>

<tr>

<th>お名前</th>

<td><span th:text="\*{name}"></span></td>

</tr>

〇Validationチェック例

@SuppressWarnings("serial")

public class OrderInput implements Serializable {

@NotBlank

private String name;

@NotBlank

private String address;

@NotBlank

@Pattern(regexp = "0\\d{1,4}-\\d{1,4}-\\d{4}")

private String phone;

@NotBlank

@Email

private String emailAddress;

@NotNull

private PaymentMethod paymentMethod;

〇Restful Webサービス

- RESTのルール

・HTTPメソッド

GET リソースを取得する @GetMapping

POST リソースを新規に登録する @PostMapping

PUT リソースを更新する @PutMapping

DELETE リソースを削除する @DeleteMapping

・HTTPリクエストのデータ構造

リクエストライン PUT /product/p01 HTTP/1.1

リクエストヘッダ Host: [shopping.example.com](https://shopping.example.com/)

Content-Type: application/json

リクエストボディ {"id": "p01", "name":"name01",・・・}

・HTTPレスポンスのデータ構造

ステータスライン HTTP/1.1 200 OK

レスポンスヘッダ Content-Type: application/json

Date: Tue, 03 Jan 2023 04:52:29 GMT

レスポンスボディ {"id": "p01", "name":"name01",・・・}

・ステータスコードの1桁目

1xx 処理がまだ継続している

2xx 正常に処理が終了

3xx 別の処理の呼び出しが必要

4xx クライアント側に起因するエラー(リクエストに不備があるなど)

5xx サーバー側に起因するエラー(データベースアクセスでエラーが発生したど)

・主なステータスコード

200 (OK) 正常に処理が行われたことを汎用的に表す

201  (Created) リソースが正常に追加されたことを表す

204 (No Content) 正常に処理が行われ、レスポンスボディのデータは空であることを表す。更新や削除のときに使用する。

400 (Bad Request) リクエストに何かしら不備があって処理ができなかったことを汎用的に表す

401 (Unauthorized) 認証が必要なことを表す

403 (Forbidden) 権限不足などでアクセスが禁止されていることを表す

404 (Not Found) リソースが見つからなかったことを表す

405 (Method Not Allowed) リクエストされたHTTPメソッドにWebサービスが対応していないことを表す

409 (Conflict) 他の処理と競合(他のクライアントが同時に更新したなど)して処理ができなかったことを表す

500 (Internal Server Error) サーバー側の処理で何かしらのエラーが発生したことを汎用的に表す

503 (Server Unavailable) メンテナンスなどでサーバーが利用できないことを表す

・主なリクエストヘッダ

Accept レスポンスで受け取りたいデータ形式を指定

Authorization 認証するための情報(IDやパスワードなど)を指定

Content-Type リクエストボディのデータ形式を指定

・主なレスポンスヘッダ

Content-Type レスポンスボディのデータ形式を指定

Location 新規に登録したリソースのURLを指定

・HttpMessageConverterによるリクエスト・レスポンスデータの変換

・ステータスコードの指定

@PutMapping("products/...")

@ResponseStatus(HttpStatus.NO\_CONTENT)

public void update(...){}

・リクエストヘッダの参照

@GetMapping("/foo")

pubic foo getFoo(@RequestHeader(HttpHeaders.USER\_AGENT) String userAgent){}

・レスポンスヘッダの指定

@GetMapping("/foo")

public ResponseEntity<Foo> getFoo(){

Foo foo = ...

return ResponseEntity.ok()

.eTag(foo.getVersion())

.header([HttpHeaders.DATE](https://httpheaders.date/),...)

.header("abc", "def")

.body(foo)

}

・参照系のREST APIの作成

@RestController

public class OrderMaintnanceController {

private final OrderMaintenanceServiceImpl orderMaintenanceServiceImpl;

public OrderMaintnanceController(OrderMaintenanceServiceImpl orderMaintenanceServiceImpl) {

this.orderMaintenanceServiceImpl = orderMaintenanceServiceImpl;

}

@GetMapping("/api/orders")

public List<Order> getOrders(){

return orderMaintenanceServiceImpl.findAll();

}

@GetMapping("/api/orders/{id}")

public Order getOrder(@PathVariable String id){

return orderMaintenanceServiceImpl.findById(id);

}

}

●更新系のREST APIの作成

@PutMapping("{id}")

@ResponseStatus(HttpStatus.NO\_CONTENT)

public void updateProduct(@PathVariable String id,

@Validated @RequestBody ProductMaintenanceInput productMaintenanceInput) {

productMaintenanceInput.setId(id);

productMaintenanceService.update(productMaintenanceInput);

}

@PostMapping

public ResponseEntity<Void> registerProduct(

@Validated @RequestBody ProductMaintenanceInput productMainenanceInput){

Product product = productMaintenanceService.register(productMainenanceInput);

URI location = ServletUriComponentsBuilder

.fromCurrentRequestUri()

.path("/{id}")

.buildAndExpand(product.getId())

.toUri();

return ResponseEntity.created(location).build();

}

@DeleteMapping("{id}")

@ResponseStatus(HttpStatus.NO\_CONTENT)

public void deleteProduct(@PathVariable String id) {

productMaintenanceService.delete(id);

}

@ExceptionHandler(DataNotFoundException.class)

public ResponseEntity<String> handleNotFound(DataNotFoundException ex) {

return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT\_FOUND);

}

- 訂正用Postメソッド

@PostMapping(value = "/update", params = "correct")

public String correct(@Validated ProductMaintenanceInput productMaintenanceInput) {

    return BASE\_VIEW\_NAME + "updateForm";

}

- 更新用HTML

<form th:action="@{/maintenance/product/update}" method="post"

th:object="${productMaintenanceInput}">

<th>価格</th>

<td>

<input type="text" th:field="\*{price}">円

<div th:errors="\*{price}"></div>

</td>

- 確認用HTML

<form th:action="@{/maintenance/product/validate-update-input}" method="post" th:object="${productMaintenanceInput}"><tr>

<th>商品名</th>

<td>

<span th:text="\*{name}"></span>

</td>

</tr>

<input type="hidden" th:field="\*{name}"/>

- 1ページ前に戻る

<a href="javascript:history.back()">戻る</a>

javascript:history.back() は、WebブラウザのJavaScriptの機能を利用して、ユーザーが現在見ているページの一つ前のページに戻るためのコードです。

〇ThymeLeafの使用

記号 用途 評価対象 主な使用場面

${} 変数式 Thymeleafコンテキスト (モデル属性など) データの表示、Thymeleafユーティリティオブジェクトの利用

例.th:object="${productMaintennceInput}"

- {} 選択変数式 th:object で選択されたオブジェクト フォームバインディングされたオブジェクトのプロパティへのアクセス (th:field 含む)

例.<input type="text" th:field="\*{name}" />

@{{}} URL式 アプリケーションのURLコンテキスト リンクの生成、CSSやJavaScriptファイルなどのリソースURLの指定

例.<form th:action="@{/maintenance/product/validate-update-input}" method="post" th:object="${productMaintenanceInput}">

〇Spring Securityを用いた認証・認可

認証(Authentication)

認可(Authorization)・・・リクエストの認可・メソッドの認可・画面表示の認可

Security Filter Chain

リクエストの認可

@Configuration

@EnableWebSecurity

public class SecurityConfig {

`@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception{

 http

 .authorizeHttpRequests()

 .requestMatchers(HttpMethod.POST, "/maintenance/product/\*\*").hasRole("ADMIN")

 .requestMatchers("/maintenance/product/\*\*").hasAnyRole("ADMIN", "STAFF")

 .anyRequest().permitAll()

 .and()

 .formLogin()

 .loginPage("/login")

 .failureUrl("/login?failure")

 .defaultSuccessUrl("/maintenance/product/display-list")

 .and()

 .exceptionHandling()

 .accessDeniedPage("/display-access-denied");

 return http.build();

}

@Bean

public UserDetailsService userDetailService() {

 UserDetails taro = User.builder()

 .username("taro").password("{noop}taro123").roles("ADMIN").build();

 UserDetails jiro = User.builder()

 .username("jiro").password("{noop}jiro123").roles("STAFF").build();

 UserDetails saburo = User.builder()

 .username("saburo").password("{noop}saburo123").roles("GUEST").build();

 return new InMemoryUserDetailsManager(taro,jiro,saburo);

}`

ログイン画面

@Controller

public class AuthPageController {

@RequestMapping("/display-access-denied")

public String accessDenied() {

return "auth/accessDenial";

}

@GetMapping("/login")

public String loginForm() {

return "auth/loginForm";

}

@GetMapping(value = "/login", params = "failure")

public String loginFail(Model model) {

model.addAttribute("failureMessage", "IDもしくはパスワードが違います");

return "auth/loginForm";

}

}

認可に失敗したときのエラー画面

〇Postgresでロールと作成者で認可する

- - DELETEポリシーの作成 (管理者ロールまたは作成者のみ削除可能)

CREATE POLICY delete\_resource\_policy ON resources

FOR DELETE TO your\_app\_user

USING (current\_user\_id() IN (SELECT user\_id FROM app\_users WHERE role ='admin') OR created\_by = current\_user\_id());

認証用のデータの取得

画面表示の認可

認証したユーザの情報を画面に表示

メソッドの認可

CSRFの対応

Spring SecurityによるCSRFの対応

〇putMappingによる更新とCSRFのヘッダー追加

<meta name="\_csrf" th:content="${\_csrf.token}"/>

<meta name="\_csrf\_header" th:content="${\_csrf.headerName}"/>

<script>

function sendPutRequest() {

const form = document.getElementById("putForm");

const formData = new FormData(form);

const csrfToken = document.querySelector('meta[name="\_csrf"]').content;

const csrfHeaderName = document.querySelector('meta[name="\_csrf\_header"]').content;

// formData.forEach((value, key) => data[key] = value);

console.log("Sending data:", productData);

HadoopとLDAPの違い

| 項目 | Hadoop | LDAP |

| --- | --- | --- |

| 主な機能 | 大量データの分散ストレージと分散処理 | ユーザー認証、承認、ディレクトリ情報の管理 |

| 扱うデータ | ギガバイトからペタバイト規模のあらゆるデータ形式 | ユーザー、グループ、リソースのメタ情報（少量） |

| データの性質 | 生データ、分析用データ | アクセス制御、管理用の構造化データ |

| 目的 | ビッグデータ分析、バッチ処理 | 集中認証、アクセス制御、リソース管理 |

| 典型的な利用シーン | ログ分析、データウェアハウス、機械学習 | 企業内ネットワークでのログイン、アプリケーション認証 |

〇シングルトンとスレッドセーフ

- 複数のスレッドが同じオブジェクトのメソッドを呼び出す。

- マルチスレッドでも安全にプログラムが動作する必要がある。

- スレッドごとに変わる値を、フィールドではなくローカル変数で保持する。

- Entityはスレッドによって値が変わるためスレッドセーフにならない。Beanとしては管理しない。

●Entityはスレッドによって値が変わるためスレッドセーフにならない。Beanとしては管理しない。

Webシステム(Spring)でデータベースへのアクセスをするときはこのEntityを使うと思いますが、Select・Insert・Updatではおのおのどのようにしてトランザクション管理・排他処理をしていますか?

>>>

「Entityはスレッドによって値が変わるためスレッドセーフにならない。Beanとしては管理しない。」という前提は、ORM (Object-Relational Mapping) におけるEntityの一般的な取り扱い、特にJPA (Java Persistence API) の文脈での理解として非常に重要です。

Entityオブジェクトは、データベースの特定のレコードをメモリ上で表現するものであり、そのインスタンスは通常、1つのトランザクションまたは1つのリクエストのスコープ内で利用され、他のスレッドと共有されるべきではありません。Springで`@Component`や`@Service`などのアノテーションをつけてBeanとして管理してしまうと、シングルトンとしてインスタンスが共有され、意図しないデータの上書きや状態の破損が発生する可能性があります。そのため、Entityはフレームワークによって適切に管理される（例えば、JPAのEntityManagerによってライフサイクルが管理される）ことが前提となります。

Webシステム（Spring）でデータベースアクセスを行う際に、Entityを利用しつつ、`SELECT`、`INSERT`、`UPDATE`におけるトランザクション管理と排他処理がどのように行われるか、それぞれ解説します。

---

### 〇トライザクション管理と排他処理

Springフレームワークでデータベースアクセスを行う場合、一般的には以下の技術スタックが用いられます。

- \*\*ORMフレームワーク:\*\* JPA（Java Persistence API）の実装であるHibernateが最も一般的です。

- \*\*Spring Data JPA:\*\* JPAをより使いやすくするためのSpringのサブプロジェクト。リポジトリパターンを簡潔に実装できます。

- \*\*Springのトランザクション管理:\*\* `@Transactional`アノテーションによる宣言的トランザクション管理が主流です。

### 1. トランザクション管理

SpringのWebシステムにおけるトランザクション管理は、主に\*\*宣言的トランザクション管理\*\*によって行われます。これは、`@Transactional`アノテーションをサービス層のメソッドに付与することで実現されます。

- \*\*`@Transactional` アノテーションの役割:\*\*

 - \*\*トランザクションの開始:\*\* メソッドが呼び出されると、JDBC接続が取得され、トランザクションが開始されます。

 - \*\*トランザクションのコミット/ロールバック:\*\*

 - メソッドが正常に完了すると、トランザクションがコミットされます。

 - メソッド内でチェック済み例外でない例外（RuntimeExceptionなど）が発生すると、トランザクションがロールバックされます。

 - 特定のチェック済み例外でもロールバックさせたい場合は、`@Transactional(rollbackFor = {MyCheckedException.class})`のように指定できます。

 - \*\*データベース接続の管理:\*\* トランザクションスコープ内で、同じデータベース接続が共有されます。

 - \*\*エンティティの永続化コンテキスト管理:\*\* JPAの場合、`EntityManager`（Hibernateでは`Session`）のライフサイクルがトランザクションと同期します。

 - トランザクション開始時に新しい永続化コンテキストが作成され、トランザクション終了時に閉じられます。

 - 永続化コンテキストは、そのトランザクション内でロードされたエンティティを管理し、変更を追跡します（ダーティチェッキング）。

### SELECT (読み取り)

`@Transactional`が付与されたメソッド内で`SELECT`を行う場合、その読み取り操作もトランザクションスコープ内で実行されます。

- \*\*分離レベル:\*\* デフォルトでは、データベースのデフォルト分離レベル（例えば、MySQLのInnoDBでは`REPEATABLE READ`）が適用されます。これにより、トランザクション内で同じデータを複数回読み込んでも、そのトランザクションが開始されてからコミットされるまでの間は常に同じ結果が保証されます（ファントムリードは発生する可能性あり）。

- \*\*読み取り一貫性:\*\* データベースによってはMVCC (Multi-Version Concurrency Control) を利用して、読み取り中に他のトランザクションによるデータの変更をブロックすることなく、一貫性のあるスナップショットを読み取ります。

### INSERT / UPDATE / DELETE (書き込み)

これらの操作は、必ずトランザクションスコープ内で実行される必要があります。

1. \*\*`INSERT`:\*\*

 - アプリケーションサーバーが`@Transactional`かメソッド内で`repository.save(entity)`（JPAの場合）などを呼び出すと、Entityオブジェクトが永続化コンテキストに追加されます。

 - トランザクションがコミットされる際に、データベースに`INSERT`文が発行され、データが永続化されます。

2. \*\*`UPDATE`:\*\*

 - `@Transactional`メソッド内でデータベースからEntityをロードし（例: `repository.findById(id)`）、そのEntityのプロパティを変更します。

 - JPA/Hibernateは、トランザクションがコミットされる際に、永続化コンテキストに保持されているエンティティの変更（ダーティチェッキング）を検出し、自動的に`UPDATE`文を生成してデータベースに反映します。`repository.save(entity)`を明示的に呼び出す必要はありませんが、明示的に呼び出しても問題ありません（多くの場合、内部でマージ処理が行われます）。

 - この「ダーティチェッキング」の仕組みが、Entityの値を直接変更するだけでデータベースに反映される理由です。

### 2. 排他処理

排他処理は、複数のトランザクションが同時に同じデータにアクセスし、競合が発生することを防ぎ、データの一貫性を保つために非常に重要です。

### 2.1. 楽観的ロック (Optimistic Locking)

Webシステムでは、ユーザーの操作が遅い、またはトランザクションの期間が長い場合が多いため、\*\*楽観的ロック\*\*が一般的に推奨されます。これは、データ更新時にのみ競合をチェックする方法です。

- \*\*仕組み:\*\*

 1. エンティティに\*\*バージョン番号\*\*（`@Version`アノテーションを付与した数値型フィールド）または\*\*タイムスタンプ\*\*（`@Version`を付与した`java.util.Date`や`java.sql.Timestamp`型フィールド）を追加します。

 2. データを読み込む際にはロックを行いません。

 3. データを更新する際、読み込んだ時点のバージョン番号（またはタイムスタンプ）と、現在のデータベース上のバージョン番号を比較します。

 4. \*\*一致する場合:\*\* 更新を許可し、バージョン番号をインクリメントします。

 5. \*\*不一致の場合:\*\* 他のトランザクションがそのデータを先に更新したと判断し、`OptimisticLockException`などの例外をスローします。

- \*\*利点:\*\* 読み込み時にロックしないため、並行性が高い。デッドロックの可能性が低い。

- \*\*欠点:\*\* 競合が発生した場合、ユーザーにエラーを通知し、再試行を促すなどのリカバリーロジックが必要になる。

\*\*JPAでの実装例:\*\*

Java

`@Entity

public class Product {

 @Id

 @GeneratedValue(strategy = GenerationType.IDENTITY)

 private Long id;

 private String name;

 private int stock;

 @Version // バージョン管理フィールド

 private Long version;

 // Getter, Setter

}

@Service

@Transactional

public class ProductService {

 @Autowired

 private ProductRepository productRepository;

 public void updateStock(Long productId, int quantity) {

 Product product = productRepository.findById(productId)

 .orElseThrow(() -> new EntityNotFoundException("Product not found"));

 // 在庫を更新

 product.setStock(product.getStock() - quantity);

 // save()を呼び出さなくてもトランザクションコミット時に自動的に更新される

 // ただし、明示的にsave()を呼び出すことも可能

 // productRepository.save(product);

 }

}`

上記の例では、`Product`エンティティに`@Version`フィールドがあります。`updateStock`メソッドが実行され、`Product`エンティティが更新される際に、JPAは自動的にバージョン番号をチェックし、競合があれば`OptimisticLockException`をスローします。

### 2.2. 悲観的ロック (Pessimistic Locking)

\*\*悲観的ロック\*\*は、データを読み込む際にロックをかけ、他のトランザクションからの更新や読み込み（共有ロックの場合）をブロックする方法です。データ競合が頻繁に発生すると予想される場合や、厳密なデータ整合性が求められる場合に利用されます。

- \*\*仕組み:\*\*

 1. データを読み込む際に、データベースのロック機構（`FOR UPDATE`など）を利用してロックを取得します。

 2. 他のトランザクションは、そのデータが解放されるまで待機するか、ロックタイムアウトが発生します。

 3. トランザクションがコミットまたはロールバックされると、ロックが解放されます。

- \*\*種類:\*\*

 - \*\*共有ロック (Shared Lock / Read Lock):\*\* 複数のトランザクションが同時に読み込むことはできるが、書き込みはブロックする。

 - \*\*排他ロック (Exclusive Lock / Write Lock):\*\* 他のすべての読み込みと書き込みをブロックする。

- \*\*利点:\*\* データの一貫性を強く保証できる。

- \*\*欠点:\*\* 並行性が低下する可能性がある。ロック期間が長いとデッドロックが発生しやすくなる。

\*\*JPAでの実装例:\*\*

Java

`@Service

@Transactional

public class ProductService {

 @Autowired

 private ProductRepository productRepository;

 public void updateStockWithPessimisticLock(Long productId, int quantity) {

 // 排他ロックをかけてProductエンティティを取得

 // LockModeType.PESSIMISTIC\_WRITE はデータベースの行ロック（FOR UPDATEなど）をかける

 Product product = productRepository.findById(productId, LockModeType.PESSIMISTIC\_WRITE)

 .orElseThrow(() -> new EntityNotFoundException("Product not found"));

 // 在庫を更新

 product.setStock(product.getStock() - quantity);

 // トランザクションコミット時に変更が永続化される

 }

}

// リポジトリのインターフェース

public interface ProductRepository extends JpaRepository<Product, Long> {

 @Lock(LockModeType.PESSIMISTIC\_WRITE)

 Optional<Product> findById(Long id, LockModeType lockModeType);

}`

この場合、`findById`メソッドが呼ばれると、データベースレベルで対象レコードに排他ロックがかかり、他のトランザクションが同じレコードを更新しようとするとブロックされます。

---

### まとめ

- \*\*トランザクション管理:\*\* Springの`@Transactional`アノテーションにより、メソッド単位でトランザクションの開始、コミット、ロールバック、そしてJPAの永続化コンテキストの管理が自動的に行われます。`SELECT`、`INSERT`、`UPDATE`のいずれの操作もこのトランザクションスコープ内で実行されます。

- \*\*排他処理:\*\*

 - Webシステムでは、一般的に\*\*楽観的ロック\*\*（`@Version`アノテーション）が推奨されます。これにより、並行性を高く保ちつつ、データ競合が発生した場合はアプリケーション側で例外処理を行います。

 - 厳密な整合性が求められる、または競合が非常に頻繁な場合は、\*\*悲観的ロック\*\*（`LockModeType`）も利用できますが、並行性の低下やデッドロックのリスクを考慮する必要があります。

Entityはトランザクションスコープ内でその状態が管理され、他のスレッドと共有されることはありません。これにより、Entityの「スレッドセーフでない」という特性が問題なく扱われます。SpringとJPAの組み合わせは、これらの複雑なトランザクション管理と排他処理を開発者が意識することなく、宣言的に記述できる強力な仕組みを提供します。

〇シングルトンでの各変数はの取り扱い

- オブジェクト変数

ログファイルのパス、データベース接続情報、設定パラメータ、キャッシュ、シーケンス番号、統計情報(カウンター)・リソース管理(ファイルハンドル、ソケット、スレッドプールなど、システム全体で共有・管理されるべきリソース)など、システム全体で共有されるべきデータ。

- ローカル変数

メソッドのスコープ内で完結する一時的なデータは、全てローカル変数として宣言すべきです。

- static変数

定数とシングルトンインスタンス自身を保持する用(getInstance())のみ。

public static synchronized AppConfig getInstance() {}

例.

// シングルトンインスタンスを保持するstaticなフィールド

private static AppConfig instance;

private AtomicLong requestCounter; // リクエスト数をカウントする例

// プライベートコンストラクタで外部からのインスタンス生成を禁止

private AppConfig() {

// 設定ファイルを読み込むなどの初期化処理

loadProperties();

this.requestCounter = new AtomicLong(0); // カウンターの初期化

}

// シングルトンのインスタンス変数（状態）

private String databaseUrl;

private String databaseUser;

// シングルトンインスタンスを取得するstaticメソッド

public static synchronized AppConfig getInstance() {

if (instance == null) {

instance = new AppConfig();

}

return instance;

}

// 状態（インスタンス変数）にアクセスするメソッド

public String getDatabaseUrl() {

return databaseUrl;

}

public String getDatabaseUser() {

return databaseUser;

}

public static void main(String[] args) {

// シングルトンのインスタンスを取得

AppConfig config1 = AppConfig.getInstance();

AppConfig config2 = AppConfig.getInstance();

// 設定値の利用

System.out.println("Database URL: " + config1.getDatabaseUrl());

System.out.println("Max Connections: " + config1.getMaxConnections());

// 状態（カウンター）の変更と取得

config1.incrementRequestCount();

config2.incrementRequestCount(); // 同じインスタンスなので、カウンターは共有される

System.out.println("Total Requests: " + config1.getRequestCount());

}

Javaでメソッドチェーンを使ったサンプルプログラムをいくつかご紹介します。

1. Builderパターンによるオブジェクト生成（設定メソッドチェーン）

これはメソッドチェーンの最も一般的な用途の一つです。オブジェクトを段階的に構築していく際に、各設定メソッドが自分自身（this）を返すことでチェーンを可能にします。

Java

public class Pizza {

private String size;

private boolean cheese;

private boolean pepperoni;

private boolean mushrooms;

```

// プライベートコンストラクタで、Builderからのみ生成されるようにする

private Pizza(Builder builder) {

 this.size = builder.size;

 this.cheese = builder.cheese;

 this.pepperoni = builder.pepperoni;

 this.mushrooms = builder.mushrooms;

}

// 各属性に対するゲッター (表示用)

public String getSize() { return size; }

public boolean hasCheese() { return cheese; }

public boolean hasPepperoni() { return pepperoni; }

public boolean hasMushrooms() { return mushrooms; }

@Override

public String toString() {

 return "Pizza [Size=" + size + ", Cheese=" + cheese +

 ", Pepperoni=" + pepperoni + ", Mushrooms=" + mushrooms + "]";

}

// --- Builderクラス ---

public static class Builder {

 private String size;

 private boolean cheese = false; // デフォルト値

 private boolean pepperoni = false;

 private boolean mushrooms = false;

 // 必須の引数を持つコンストラクタ

 public Builder(String size) {

 this.size = size;

 }

 // 設定メソッド (メソッドチェーンを可能にするためにthisを返す)

 public Builder withCheese() {

 this.cheese = true;

 return this;

 }

 public Builder withPepperoni() {

 this.pepperoni = true;

 return this;

 }

 public Builder withMushrooms() {

 this.mushrooms = true;

 return this;

 }

 // 最終的にPizzaオブジェクトを構築するメソッド

 public Pizza build() {

 return new Pizza(this);

 }

}

public static void main(String[] args) {

 System.out.println("--- Pizza Builder Example ---");

 // メソッドチェーンを使ってピザを注文

 Pizza largeVeggiePizza = new Pizza.Builder("Large")

 .withCheese()

 .withMushrooms()

 .build(); // 最後にbuild()を呼び出してPizzaオブジェクトを生成

 System.out.println(largeVeggiePizza);

 // 出力例: Pizza [Size=Large, Cheese=true, Pepperoni=false, Mushrooms=true]

 Pizza smallPepperoniPizza = new Pizza.Builder("Small")

 .withCheese()

 .withPepperoni()

 .build();

 System.out.println(smallPepperoniPizza);

 // 出力例: Pizza [Size=Small, Cheese=true, Pepperoni=true, Mushrooms=false]

 Pizza customPizza = new Pizza.Builder("Medium")

 .withMushrooms()

 .build(); // チーズやペパロニなし

 System.out.println(customPizza);

 // 出力例: Pizza [Size=Medium, Cheese=false, Pepperoni=false, Mushrooms=true]

}

```

}

2. Stream APIによるデータ処理（異なるオブジェクトを返すチェーン）

Java 8から導入されたStream APIは、メソッドチェーンの強力な例です。各中間操作メソッドは、処理済みの要素を持つ新しいStreamオブジェクトを返すことでチェーンが継続されます。

Java

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class StreamChainingExample {

public static void main(String[] args) {

System.out.println("--- Stream API Chaining Example ---");

```

 List<String> names = Arrays.asList("Alice", "bob", "Charlie", "david", "Eve");

 // メソッドチェーンを使ってデータを変換・フィルタリング

 List<String> processedNames = names.stream() // ListからStream<String>を生成

 .map(String::toUpperCase) // 各要素を大文字に変換し、新しいStream<String>を返す

 .filter(s -> s.startsWith("A")) // 'A'で始まる要素のみをフィルタリングし、新しいStream<String>を返す

 .sorted() // ソートし、新しいStream<String>を返す

 .collect(Collectors.toList()); // 結果をListに収集 (終端操作)

 System.out.println("Original Names: " + names);

 // 出力例: Original Names: [Alice, bob, Charlie, david, Eve]

 System.out.println("Processed Names: " + processedNames);

 // 出力例: Processed Names: [ALICE]

 // 別の例: 数値の集計

 List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

 long sumOfEvenSquares = numbers.stream()

 .filter(n -> n % 2 == 0) // 偶数のみをフィルタリング

 .mapToLong(n -> (long) n \* n) // 各偶数を2乗し、long型に変換 (新しいLongStreamを返す)

 .sum(); // 合計を計算 (終端操作)

 System.out.println("Sum of even squares: " + sumOfEvenSquares);

 // 出力例: Sum of even squares: 220 (2^2 + 4^2 + 6^2 + 8^2 + 10^2 = 4 + 16 + 36 + 64 + 100 = 220)

}

```

}

3. シンプルなユーティリティクラス（thisを返すチェーン）

ログメッセージの組み立てなど、シンプルなユーティリティクラスでもメソッドチェーンは有効です。

Java

public class MessageBuilder {

private StringBuilder message;

```

public MessageBuilder() {

 this.message = new StringBuilder();

}

public MessageBuilder append(String text) {

 message.append(text);

 return this; // 自身を返す

}

public MessageBuilder appendLine(String text) {

 message.append(text).append("\\n");

 return this; // 自身を返す

}

public String build() {

 return message.toString();

}

public static void main(String[] args) {

 System.out.println("--- Message Builder Example ---");

 String formattedMessage = new MessageBuilder()

 .append("Hello")

 .append(" ")

 .append("World")

 .appendLine("!")

 .append("This is a test.")

 .build();

 System.out.println(formattedMessage);

 /\*

 出力例:

 Hello World!

 This is a test.

 \*/

}

```

}

これらのサンプルプログラムは、Javaでメソッドチェーンがどのように実装され、利用されるかを示しています。特にBuilderパターンやStream APIでは、コードの可読性と表現力を大幅に向上させることができます。

〇シングルトン

スレッドセーフなソースコードを作成するには、フィールドではなくローカル変数を利用する。

@Controller

public class CalculateController {

```

//private int total;

@GetMapping("/calculate-price")

public String calculatePrice(@RequestParam int a, @RequestParam int b, @RequestParam int c, Model model) {

 int total = a + b + c;

 System.out.println(total);

 int price = (int)(total \* 1.1);

 model.addAttribute("price", price);

 return "price";

}

```

}

〇Join含んだJDBC呼び出しの例

1対1のJoinの例　RowMapper

1対多のJoinの例　ResultSetExtractor

mainからのRepositoryの呼び出し

@Repository

public class JdbcOrderItemRepository implements OrderItemRepository {

private final JdbcTemplate jdbcTemplate;

```

public JdbcOrderItemRepository(JdbcTemplate jdbcTemplate) {

 this.jdbcTemplate = jdbcTemplate;

}

@Override

public OrderItem selectById(String id) {

 return jdbcTemplate.queryForObject("""

 SELECT

 i.id AS i\_id,

 i.order\_id AS i\_order\_id,

 i.product\_id AS i\_product\_id,

 i.price\_at\_order AS i\_price\_at\_order,

 i.quantity AS i\_quantity,

 p.id AS p\_id,

 p.name AS p\_name,

 p.price AS p\_price,

 p.stock AS p\_stock

 FROM

 t\_order\_item i

 LEFT OUTER JOIN t\_product p

 ON i.product\_id = p.id

 WHERE

 i.id = ?""", new OrderItemRowMapper(), id);

}

static class OrderItemRowMapper implements RowMapper<OrderItem> {

 @Override

 public OrderItem mapRow(ResultSet rs, int rowNum) throws SQLException {

 OrderItem orderItem = new OrderItem();

 orderItem.setId("i\_id");

 orderItem.setOrderId("orderId");

 orderItem.setProductId("i\_product\_id");

 orderItem.setPriceAtOrder(rs.getInt("i\_price\_at\_order"));

 orderItem.setQuantity(rs.getInt("i\_quantity"));

 Product product = new Product();

 product.setId("p\_id");

 product.setName("p\_name");

 product.setPrice(rs.getInt("p\_price"));

 product.setStock(rs.getInt("p\_stock"));

 orderItem.setProduct(product);

 return orderItem;

 }

}

```

}

Repository

public class JdbcOrderRepository implements OrderRepository {

private final JdbcTemplate jdbcTemplate;

```

public JdbcOrderRepository(JdbcTemplate jdbcTemplate) {

 this.jdbcTemplate = jdbcTemplate;

}

@Override

public Order selectById(String id) {

 return jdbcTemplate.query("""

 SELECT

 o.id AS o\_id,

 o.order\_date\_time AS o\_order\_date\_time,

 o.billing\_amount AS o\_billing\_amount,

 o.customer\_name AS o\_customer\_name,

 o.customer\_address AS o\_customer\_address,

 o.customer\_phone AS o\_customer\_phone,

 o.customer\_email\_address AS o\_customer\_email\_address,

 o.payment\_method AS o\_payment\_method,

 i.id AS i\_id,

 i.order\_id AS i\_order\_id,

 i.product\_id AS i\_product\_id,

 i.price\_at\_order AS i\_price\_at\_order,

 i.quantity AS i\_quantity

 FROM

 t\_order o

 LEFT OUTER JOIN t\_order\_item i

 ON o.id = i.order\_id

 WHERE

 o.id = ?""", new OrderResultSetExtractor(), id);

}

public class OrderResultSetExtractor implements ResultSetExtractor<Order>{

 @Override

 public Order extractData(ResultSet rs) throws SQLException, DataAccessException{

 Order order = null;

 while (rs.next()) {

 if (order == null) {

 order = new Order();

 order.setOrderItems(new ArrayList());

 order.setId("o\_id");

 order.setOrderDateTime(rs.getTimestamp("o\_order\_date\_time").toLocalDateTime());

 order.setBillingAmount(rs.getInt("o\_billing\_amount"));

 order.setCustomerName("o\_customer\_name");

 order.setCustomerAddress("o\_customer\_address");

 order.setCustomerPhone("o\_customer\_phone");

 order.setCustomerEmailAddress("o\_customer\_email\_address");

 // ここが重要: データベースから取得した文字列を列挙型に変換

 String paymentMethodString = rs.getString("o\_payment\_method");

 if (paymentMethodString != null) {

 try {

 order.setPaymentMethod(PaymentMethod.valueOf(paymentMethodString));

 } catch (IllegalArgumentException e) {

 // データベースに存在しない不正な支払い方法が格納されていた場合の処理

 // 例: ログ出力、デフォルト値の設定など

 System.err.println("不正な支払い方法がデータベースに格納されています: " + paymentMethodString);

 // または throw new SomeSpecificException("Invalid payment method in database: " + paymentMethodString, e);

 }

 }

 }

 OrderItem orderItem = new OrderItem();

 orderItem.setId("i\_id");

 orderItem.setOrderId("i\_order\_id");

 orderItem.setProductId("i\_product\_id");

 orderItem.setPriceAtOrder(rs.getInt("i\_price\_at\_order"));

 orderItem.setQuantity(rs.getInt("i\_quantity"));

 order.getOrderItems().add(orderItem);

 }

 return order;

 }

}

```

}

@SpringBootApplication

public class ShoppingApplication {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(ShoppingApplication.class, args);

OrderItemRepository orderItemRepository = context.getBean(OrderItemRepository.class);

OrderItem orderItem = orderItemRepository.selectById("i01");

```

 System.out.println("------------- OrderItemの中身 -------------");

 System.out.println(orderItem.getPriceAtOrder());

 System.out.println(orderItem.getProduct());

 OrderRepository orderRepository = context.getBean(OrderRepository.class);

 Order order = orderRepository.selectById("o01");

 System.out.println("------------- Orderの中身 -------------");

 System.out.println(order.getCustomerName());

 for (OrderItem i: order.getOrderItems()) {

 System.out.println(i.getPriceAtOrder());

 }

}

```

>>>1オブジェクトの抽出は、queryForObject()

複数のオブジェクトの抽出は、query()

〇キーが重複した時のエラー出力と画面出力

●DuplicateKeyExceptionによる捕捉

@Repository

public class JdbcTrainingRepository implements TrainingRepository {

・・・

@Override

public void insert(Training training) {

try {

jdbcTemplate.update("INSERT INTO training VALUES (?,?,?,?,?,?)",

training.getId(),

training.getTitle(),

training.getStartDateTime(),

training.getEndDateTime(),

training.getReserved(),

training.getCapacity());

} catch (DuplicateKeyException e) {

// キーが重複した場合の処理

System.err.println("研修ID '" + training.getId() + "' は既に存在します。");

// 例外を再スローするか、特定のビジネス例外に変換してスローすることもできます。

throw new TrainingIdAlreadyExistsException("指定された研修IDは既に存在します。", e);

// または null を返してコントローラーで処理

// return null;

} catch (DataAccessException e) {

// その他のデータアクセスエラー

System.err.println("データベース操作中にエラーが発生しました: " + e.getMessage());

throw e; // または適切な例外に変換

}

}

●例外クラスをRuntimeExceptionを継承して作成

package com.example.training.exception;

/\*\*

- 研修IDが既に存在する場合にスローされるカスタム例外。

- RuntimeException を継承することで、非チェック例外として扱われます。

\*/

public class TrainingIdAlreadyExistsException extends RuntimeException {

 // コンストラクタ1: エラーメッセージのみを受け取る

 public TrainingIdAlreadyExistsException(String message) {

 super(message); // 親クラス(RuntimeException)のコンストラクタを呼び出す

 }

 // コンストラクタ2: エラーメッセージと原因となった例外を受け取る

 // このコンストラクタは、DuplicateKeyExceptionなどの元の例外をラップする場合に便利です。

 public TrainingIdAlreadyExistsException(String message, Throwable cause) {

 super(message, cause); // 親クラスのコンストラクタを呼び出す

 }

 }

●@ControllerAdvice でのハンドリングによる画面出力(ControllerのGlobalControllerAdviceクラスで取得)

package com.example.training.controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.ControllerAdvice;

import org.springframework.web.bind.annotation.ExceptionHandler;

import com.example.training.entity.Training;

import com.example.training.exception.TrainingIdAlreadyExistsException;

@ControllerAdvice

public class GlobalControllerAdvice {

@ExceptionHandler(Exception.class)

public String systemError(Exception ex) {

ex.printStackTrace();

return "systemError";

}

// TrainingIdAlreadyExistsException 専用のハンドラ

@ExceptionHandler(TrainingIdAlreadyExistsException.class)

public String handleTrainingIdAlreadyExists(TrainingIdAlreadyExistsException ex, Model model) {

model.addAttribute("duplicateError", ex.getMessage()); // エラーメッセージをModelに追加

model.addAttribute("trainingAdminInput", new Training()); // フォームの再表示のために空のTrainingオブジェクトも渡す

return "admin/training/registrationForm"; // 研修登録フォームのテンプレート名を返す

}

}

●エラー画面出力

<h1>研修登録</h1>

<form th:action="@{/admin/training/validate-registration-input}" method="post"

th:object="${trainingAdminInput}">

<table>

<tr>

<th>研修ID</th>

<td>

<input type="text" th:field="\*{id}"/>

<div th:errors="\*{id}"></div>

<div th:if="${duplicateError != null}" th:text="${duplicateError}"></div>

</td>

</tr>

〇トランザクションの伝搬

ログ出力などロールバックしたくない処理では、Propagation.REQUIRES\_NEWで新しいトランザクションが開始されコミットされる。

```

@Transactional(propagation = Propagation.REQUIRES\_NEW)

//@Transactional

@Override

public void registerLog(String functionName) {

 AuditLog auditLog = new AuditLog();

 auditLog.setId(UUID.randomUUID().toString());

 auditLog.setFunctionName(functionName);

 auditLog.setEventDateTime(LocalDateTime.now());

 auditLog.setUserId("dummy");

 auditLogRepository.insert(auditLog);

}

```

〇セッションスコープ

●セッションスコープのBean

コントローラに、セッションスコープBeanを設定する。

@Component

@SessionScope(proxyMode = ScopedProxyMode.TARGET\_CLASS)

@SuppressWarnings("serial")

public class ReservationSession implements Serializable {

private ReservationInput reservationInput;

```

public void clearData() {

 this.reservationInput = null;

}

public ReservationInput getReservationInput() {

 return reservationInput;

}

public void setReservationInput(ReservationInput reservationInput) {

 this.reservationInput = reservationInput;

}

```

}

●コントローラにセッションスコープのDIを追加する。

・Validationメソッドでtrueの場合は、セッションスコープのBeanにセッションスコープのエンティティをsetする。

・Correctメソッドで、セッションスコープのエンティティをgetしてModelで渡す。

・更新メソッドで、セッションスコープのエンティティをgetしてupdateメソッドに渡して更新し、その後セッションスコープをclearする。

・例外ハンドラーメソッドを設定し、エラー画面出力を行う。

@Controller

@RequestMapping("/reservation")

public class ReservationController {

```

private final ReservationService reservationService;

private final TrainingService trainingService;

private final ReservationSession reservationSession;

public ReservationController(ReservationService reservationService, TrainingService trainingService, ReservationSession reservationSession) {

 this.reservationService = reservationService;

 this.trainingService = trainingService;

 this.reservationSession = reservationSession;

}

@PostMapping("/validate-input")

public String validateInput(@Validated ReservationInput reservationInput, BindingResult bindingResult, Model model) {

 if (bindingResult.hasErrors()) {

 List<StudentType> studentTypeList = reservationService.findAllStudentType();

 model.addAttribute("studentTypeList", studentTypeList);

 return "reservation/reservationForm";

 }

 StudentType studentType = reservationService.findStudentTypeByCode(reservationInput.getStudentTypeCode());

 model.addAttribute("studentType", studentType);

 Training training = trainingService.findById(reservationInput.getTrainingId());

 model.addAttribute("training", training);

 reservationSession.setReservationInput(reservationInput);

 return "reservation/reservationConfirmation";

}

@PostMapping(value = "/reserve", params = "correct")

public String correctInput(Model model) {

 model.addAttribute("reservationInput", reservationSession.getReservationInput());

 List<StudentType> studentTypeList = reservationService.findAllStudentType();

 model.addAttribute("studentTypeList", studentTypeList);

 return "reservation/reservationForm";

}

@PostMapping(value = "/reserve", params = "reserve")

public String reserve(Model model) {

 ReservationInput reservationInput = reservationSession.getReservationInput();

 Reservation reservation = reservationService.reserve(reservationInput);

 model.addAttribute("reservation", reservation);

 reservationSession.clearData();

 return "reservation/reservationCompletion";

}

@ExceptionHandler(CapacityOverException.class)

public String displayCapacityOverPage() {

 return "reservation/capacityOver";

}

```

〇フラッシュスコープ

・確定画面の確定ボタンを2度押した状態で2度更新しないようにしたい。ー

> リダイレクトで解決

・リダイレクト元と先でデータ共有をしたい

redirectAttributes.addFlashAttributs()で、予約オブジェクト(予約ID)を次の画面に引き継ぐ。

※Modelだと、リクエストのデータはレスポンスでは消去される。

※HTMLへのデータ渡しは、model.addAttribute()またはセッションスコープのset()で渡す。

>

@Controller

@RequestMapping("/reservation")

public class ReservationController {

```

・・・

@PostMapping(value = "/reserve", params = "reserve")

public String reserve(RedirectAttributes redirectAttributes) {

 ReservationInput reservationInput = reservationSession.getReservationInput();

 Reservation reservation = reservationService.reserve(reservationInput);

 redirectAttributes.addFlashAttribute("reservation", reservation);

 reservationSession.clearData();

 return "redirect:/reservation/display-completion";

}

@GetMapping("display-completion")

public String displayCompletion() {

 return "reservation/reservationCompletion";

}

<body>

 <h1>予約完了</h1>

 <div th:if="${reservation != null}">予約ID「<span th:text="${reservation.id}"></span>」</div>

 <a th:href="@{/training/display-list}">研修一覧に戻る</a>

</body>

```

〇Security Contextの取得

・全ての処理で、UserName、Password、Roleなどが取得可能。

・Javaの場合は、Authentication、または@AuthenticationPrincipal UserDetailsで取得。

・HTMLの場合は、sec:authentication="name" またはsec:authentication="principal.username"により取得

●コントローラでは

@PostMapping(value = "/update", params = "correct")

public String correctUpdateInput(@Validated TrainingAdminInput trainingAdminInput , @AuthenticationPrincipal UserDetails userDetails) {

System.out.println("UserName" + userDetails.getUsername() + " Role" + userDetails.getAuthorities().toString());

return "admin/training/updateForm";

}

```

@PostMapping(value = "/update", params = "update")

public String update(@Validated TrainingAdminInput trainingAdminInput, Authentication authentication) {

 System.out.println("authentication UserName " + authentication.getName() );

 trainingAdminService.update(trainingAdminInput, authentication.getName());

 return "admin/training/updateCompletion";

}

```

●HTMLでは

<div>こんにちは<span sec:authentication="principal.username"></span>さん</div>

〇Restful Webサービスの呼び出し

HTTPメソッドとRestTemplateクラスのメソッド

GET・・・getForObject(リクエストを送信)、getForEntity(レスポンスの詳細データの取得)

PUT・・・put

POST・・・postForLocation、postForObject、postForEntity

DELETE・・・delete

すべて・・・exchange(リクエストの詳細データを設定)

●RestTemplate(Restful API呼び出し)のサンプルプログラム

```

 RestTemplate restTemplate

 = new RestTemplateBuilder()

 .rootUri("<http://localhost:8080>")

 .build();

 ProductMaintenanceInput productMaintenanceInput = new ProductMaintenanceInput();

 productMaintenanceInput.setId("px05");

 productMaintenanceInput.setName("name1");

 productMaintenanceInput.setPrice(1000);

 productMaintenanceInput.setStock(10);

 URI location = restTemplate

 .postForLocation("/api/products", productMaintenanceInput);

 Product products = restTemplate.getForObject(location,Product.class);

 productMaintenanceInput.setName("name改");

 restTemplate.put(location, productMaintenanceInput);

 restTemplate.delete(location);

```

〇プロパティの外部化

●例.

@Configuration

@Profile("stg")

@PropertySource("stg.properties")

public class StgConfig{}

●application.properties

discount.rate=0.3

discount.max=1000

●プロパティの取得

/\*\* ディスカウント率 \*/

@Value("${discount.rate}")

private double discountRate;

```

/\*\* ディスカウント上限 \*/

@Value("${discount.max}")

private int discountMax;

```

●プロパティファイルのプロファイルの設定

ConfigurationProperties(prefix = "mail.smtp")