Woodland Academy

6.4 Spring データベース接続

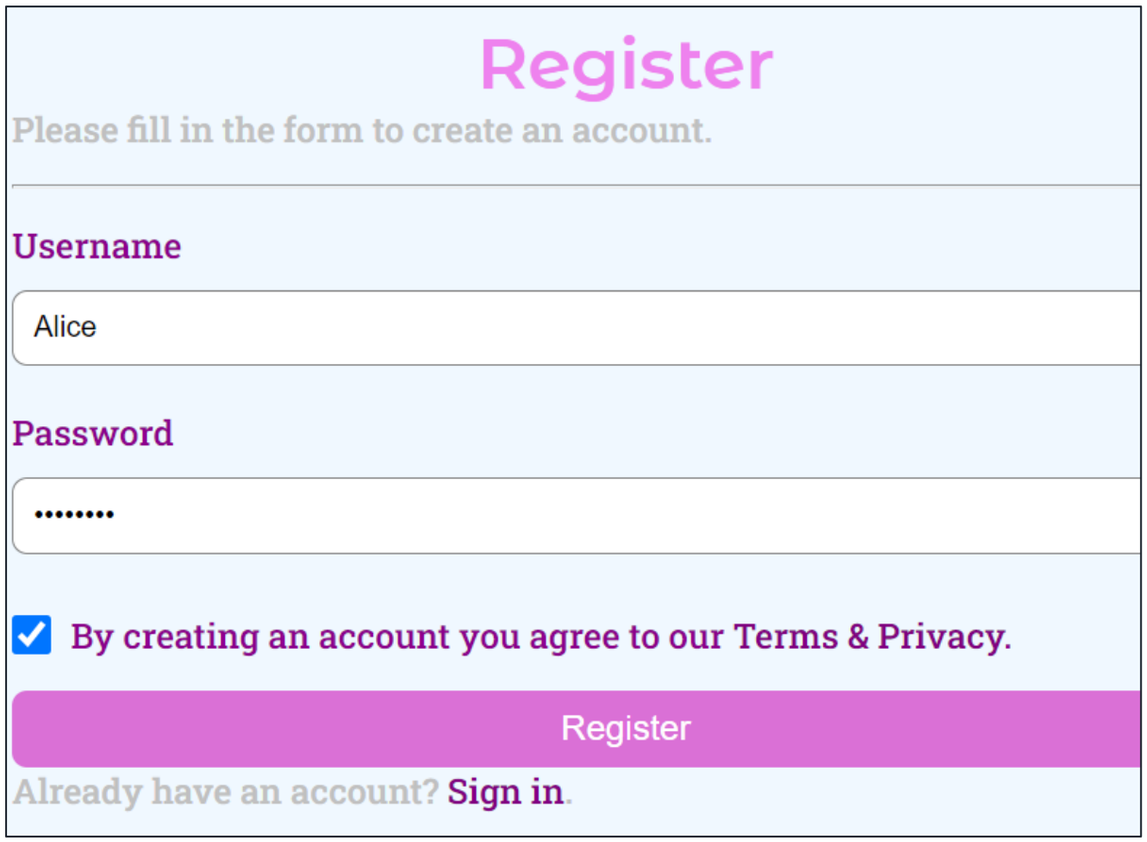
- ORM と JPA
- JPA の利用
- JPA の機能

Shape Your Future

目次

今回の目標

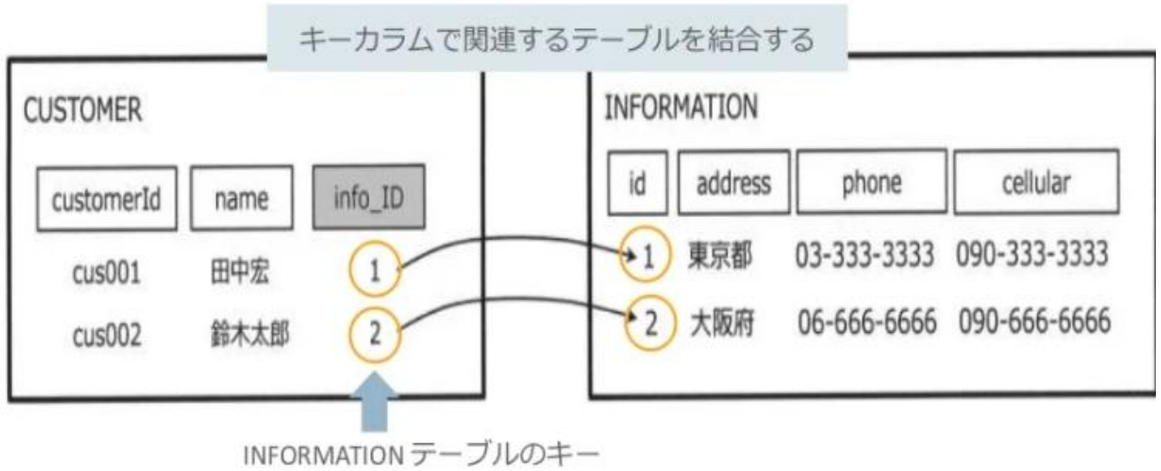
- Spring Boot でログインページと登録ページをデータベースに接続します。

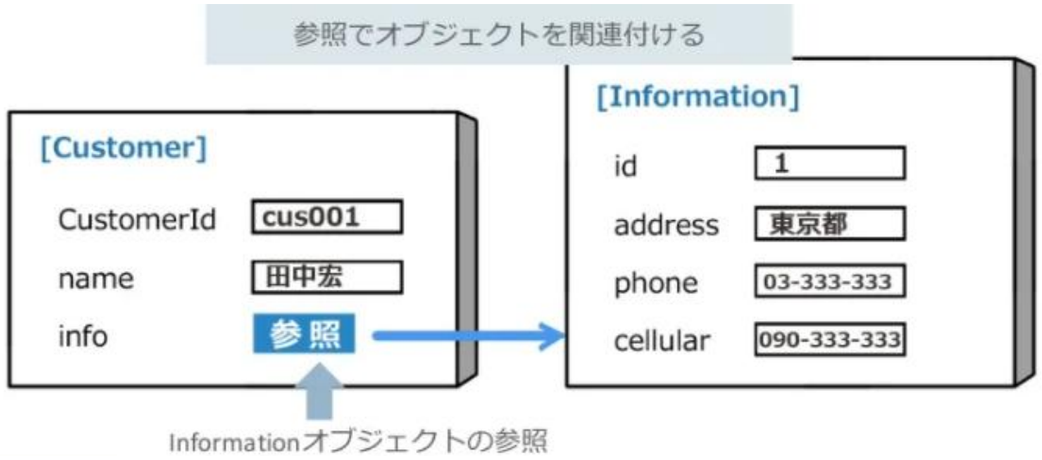


ORM とは

- **ORM**（Object Relational Mapping：オブジェクト関係マッピング）とは、**オブジェクト**と**関係**（テーブル）を対応させるソフトウェア設計技法です。

- ORM は、関係をオブジェクト指向の諸要素に対応させます：
  - 各**テーブル**は、**クラス**に対応
  - 各**レコード**（行）は、**オブジェクト**に対応
  - 各**フィールド**（カラム）は、オブジェクトの**属性**に対応
  - **外部キー**は、他のオブジェクトへの**参照**に対応





ORM の例

- 例えば、次のような SQL 文：

```sql
SELECT id, first_name, last_name, phone
FROM person
WHERE id = 10;
```

- これを、ORM の考え方で書き直したら：

```java
Person person = repository.findById(10);

System.out.println("First Name: " + person.first_name);
// ...
```

- 比較すると、ORM はオブジェクトを利用し、データベース操作をまとめることで、SQL 文を使わなくなります。開発者はオブジェクト指向プログラミングだけを書き、データベースを深く考えずに情報と直接なやり取りができます。

JPA とは

- **JPA**（Java Persistence API）とは、Java EE で開発された ORM 規格を実装するための**インタフェース**です。

- Spring は、JPA を実装する **Spring Data JPA** を開発しました。Spring Data JPA を使って、ほとんど **SQL 文を書かなくても**、以下の機能が実現できます：
  - **テーブルの作成**、
  - データの生成、削除、更新、読み出し（**CRUD**）、
  - 特定の列による検索、ソート、ページングなどの比較的**複雑なクエリ**、
  - **外部キー**による他のテーブルのオブジェクトの取得など。

ORM のメリット・デメリット

- メリット：
  - データモデルが一箇所で定義されるため、更新やメンテナンスが簡単になって、コードの**再利用も簡単**になります。
  - ORM には、データの検証、前処理、トランザクションなど、多くの**機能を自動化**するためのツールが用意されています。
  - ORM は **MVC アーキテクチャ**をさせるようなプログラムを強要し、コード構造が簡潔にできます。
  - ORM に基づくビジネスコードは、より簡単で、**コード量が少ない。**

- デメリット：
  - ORM ツールの**学習コストと配置**に多くの労力が必要。
  - ORM で表現できない複雑なクエリもあります。表現できても、直接な SQL クエリほどの**パフォーマンスを発揮できません。**
  - ORM はデータベース層を抽象化するため、開発者はインフラのデータベース操作を理解することができず、一部の特定の SQL を設定することが困難です。

Woodland Academy

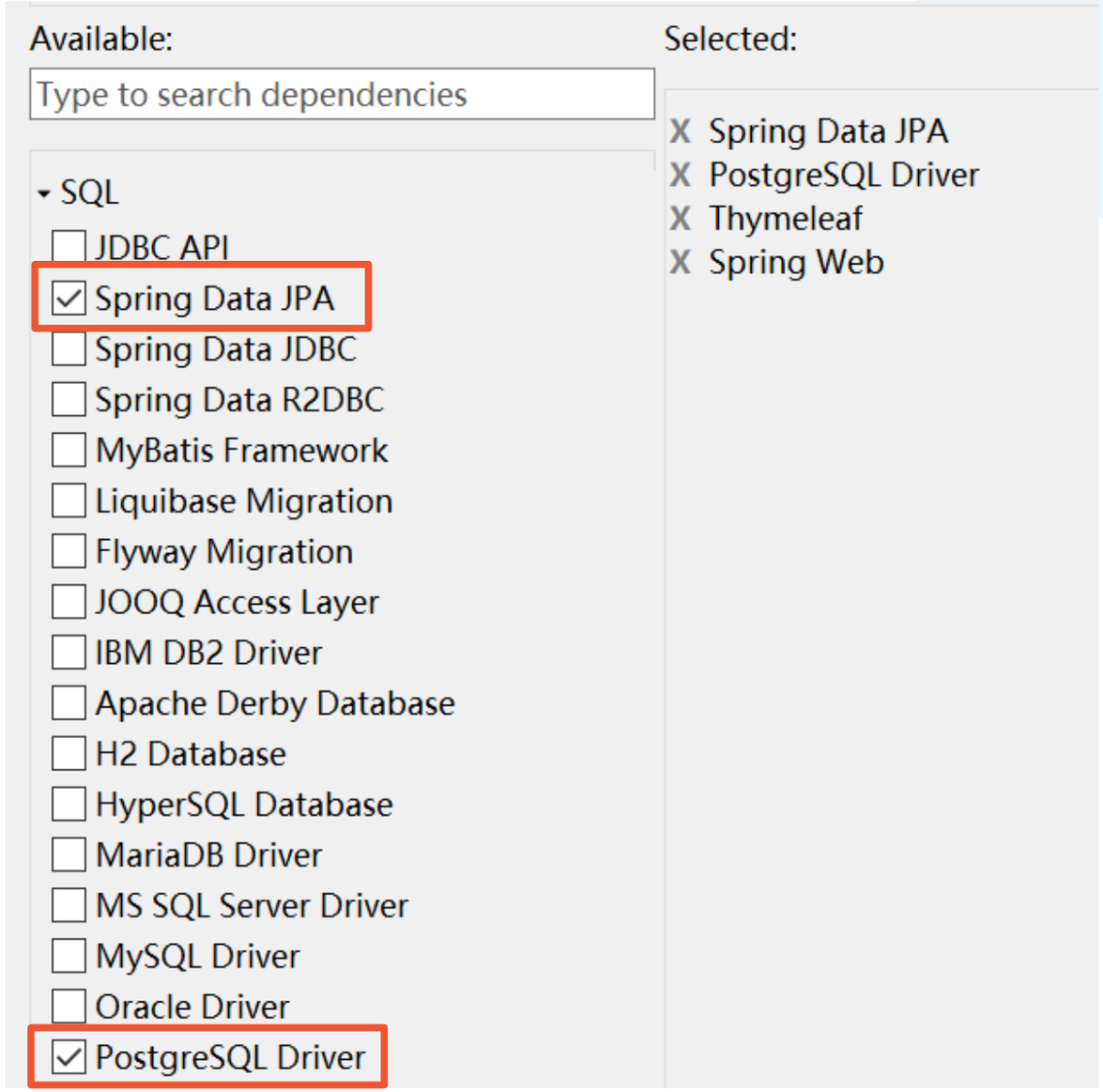
Woodland Academy

Q&A

目次

JPA の依存を追加

- プロジェクト作成時に JPA と Postgres のドライバに依存：



JPA を設定

- JPA の使用は、resrouces/application.properties というファイルにいくつかの設定を追加する必要があります。

- まず、JDBC と同様に、**データベースに接続するための情報**を追加します：

```
spring.datasource.url=jdbc:postgresql://localhost:5432/jpa_test
spring.datasource.username=postgres
spring.datasource.password=123456
```

- 次に、**テーブルを自動生成**するために、以下の設定を追加します。今のところ、これらを変更する必要はありません：

```
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update
```

データベースの設計

- まず、ログインシステムに必要なデータベースとは、どのようなものかを考えてみましょう。

- 機能：
  - 新規ユーザーの**登録**
  - 既存ユーザーの**ログイン**

- データベース：
  - 「account」というユーザーテーブル

  - テーブルのカラム：id、username、password（テーブルの主キーとして整数型の ID を追加するのは良い習慣である。ID の値は通常、自動で増加できる（auto-increment）ように設定）

Woodland Academy

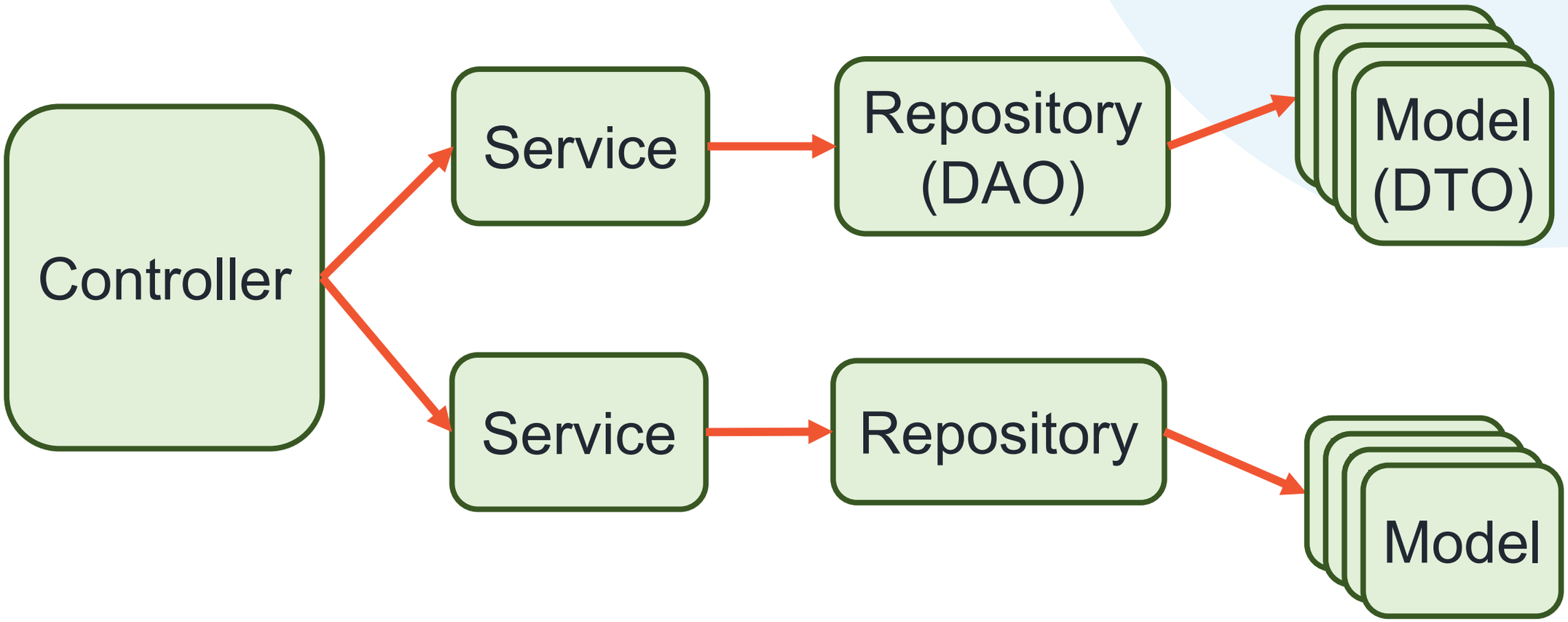
サーバのアーキテクチャ

- データベースに接続するサーバを開発する場合、通常、以下の 3 つのタイプのクラスを作成し、3 つの階層からなるアーキテクチャを構築します：
  - **永続層**（**Repository**）、または **DAO 層**：**データベースを直接操作**して、生成、削除、更新、読み取りなどの永続的な操作を行う。
  - **サービス層**（**Service**）：DAO 層のメソッドを使用して、コントローラ内のメインロジックにいくつかの利用可能な**機能（サービス）を提供**。
  - **コントローラ層**（**Controller**）：サービス層で提供されるメソッドを使用して、実際のビジネス、即ちコントローラメソッドの**実際の機能**を実装。

次へ

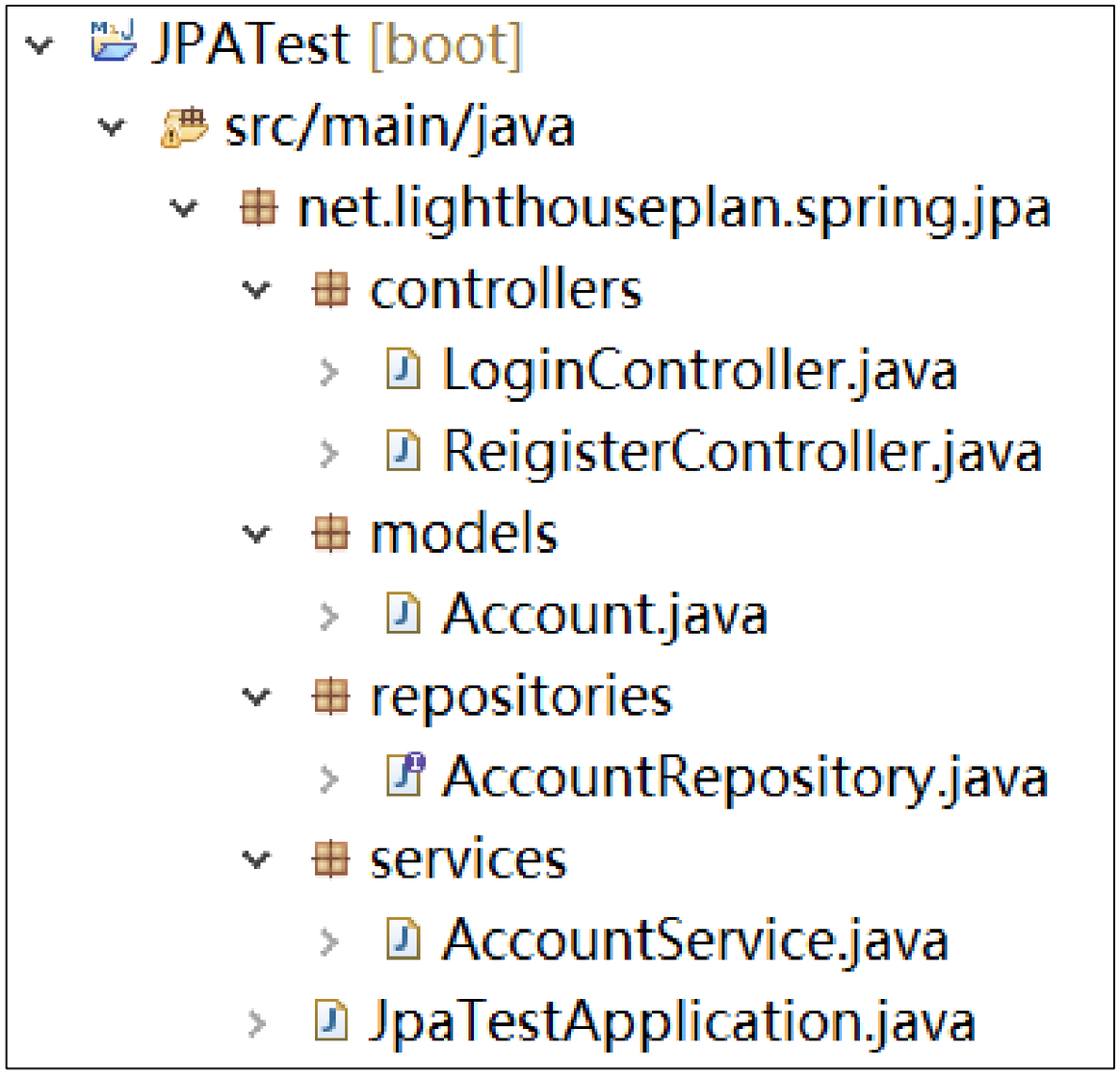
Woodland Academy

前へ



コードの構成

- 次のような構造で、異なる階層のコードを異なるパッケージに整理します：



Model の定義

- データベースの account テーブルに対応するために、Account クラスを定義し、カラムに対応する**メンバ変数**と、それらの**ゲッター**と**セッター**を追加する必要があります：

```
1 public class Account {
2     private Long id;
3     private String username;
4     private String password;
5
6     public Long getId() { return id; }
7     public void setId(Long id) { this.id = id; }
8
9     public String getUsername() { return username; }
10    public void setUsername(String username) { this.username = username; }
11
12    public String getPassword() { return password; }
13    public void setPassword(String password) { this.password = password; }
14 }
```

Account クラスにアノテーションの追加

- Account クラスの前に **@Entity** アノテーションを追加する必要があります。 これは、このクラスがデータベースの**テーブルに対応**することを Spring JPA に伝えます：

```
@Entity
public class Account {
```

次へ

- デフォルトでは、JPA は自動的に対応するテーブルの名前を与えてくれます。対応ルールは、**大文字のキャメルケース → 小文字のスネークケース**です。例えば、AccountGroup クラスは、データベースの account_group テーブルに対応されます。

- 対応するテーブル名を手動で設定したい場合は、**@Table** アノテーションを使用します：

```
@Entity
@Table(name = "account_2")
public class Account {
```

属性のアノテーション

- クラスの各**属性**は、データベースの**カラム**（フィールド）に対応しています。特に設定しない場合は、変数名から自動的にカラム名が設定されます（**大文字のキャメルケース → 小文字のスネークケース**）。

- **@Column** アノテーションを使用して手動で設定することもできます：

```
@Column(name = "user")
private String username;
```

次へ

- 特に ID 列に関しては、**@Id** アノテーションを付けることができます：

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

- ここでは、**@GeneratedValue** アノテーションによって、ID が自動増加されるように設定します。

DAO 層

- **DAO 層**は、データベースに関連する操作を処理します。
- このプロジェクトには、テーブルは account の一つしかないので、DAO レイヤーには AccountReposiroty の一つのクラスだけが必要です。
- AccountRepository インタフェースは、次のコードで定義します：

```
@Repository
public interface AccountRepository extends JpaRepository<Account, Long> { }
```

Woodland Academy

Repository の定義の解説

```
@Repository
public interface AccountRepository extends JpaRepository<Account, Long> { }
```

- **@Repository** アノテーションは、これがデータベース操作に関連するメソッドを提供する DAO 層のインタフェースであることを Spring に伝えます。

- ここで、クラスではなく、インタフェースを定義しています。このインタフェースに実装する必要のある**メソッドを宣言するだけ**で、JPA が自動的に実装してくれます。

- そのため、**JpaRepository<T, ID>** インタフェースを継承する必要があります。ここで、T はテーブルのデータのタイプ（ここでは Account）、ID はデータの ID のタイプ（ここでは Long）です。

メソッドの宣言

- メソッドは宣言だけをすればよいのです：

```
1 @Repository
2 public interface AccountRepository extends JpaRepository<Account, Long> {
3     Account findByUsername(String username);
4
5     Account save(Account account);
6 }
```

- ここで、**findByUsername()** は、ユーザー名を受け取り、そのユーザー名に対応するユーザー情報を返します。

- **save()** メソッドは、新しいユーザーをデータベースに生成し、最終的にテーブルに生成された情報を返します。（saveメソッドの宣言は JpaRepository に既に用意されているため、ここでは省略しても良いです。）

サービス層

- **サービス層**は、DAO 層の Repository クラスを使用して、コントローラが使用できる機能（サービス）を実現します。

- ここでは、 2 つのサービスが必要です
  - **validateAccount()** は、 データベースに存在するデータでユーザ名とパスワードは正しいかどうかを判断
  - **createAccount()** はユーザ名とパスワードで新規ユーザを作成

- AccountService クラスを作成し、**@Service** アノテーションを使って、Service 層のクラスであることを Spring に伝えます：

```
1 @Service
2 public class AccountService { }
```

Woodland Academy

DAO 層を利用

- AccountService は、account テーブルにアクセスの必要があります。account テーブルにアクセスするには、すでに AccountRepository インタフェースが作成されました。AccountService のメンバ変数として宣言します：

```
1 @Service
2 public class AccountService {
3     @Autowired
4     AccountRepository repository;
5 }
```

- この **@Autowired** アノテーションを付けると、Spring が**自動的に**インタフェースを**実装して、インスタンス化**させるようになり、Service 層のメソッドで直接使えるようになります。

validateAccount() メソッド

- **validateAccount()** メソッドは、username と password の 2 つのパラメータを受け取り、それらに対応するデータが既にデータベースに存在するかどうかを判断します。存在する場合は true を、存在しない場合は false を返します：

```
1 public boolean validateAccount(String username, String password) {
2     Account account = repository.findByUsername(username);
3     if (account == null || !account.getPassword().equals(password)) {
4         return false;
5     } else {
6         return true;
7     }
8 }
```

createAccount() メソッド

- **createAccount()** メソッドは、username と password の 2 つのパラメータを受け取り、新しいユーザーアカウントを作成します。ユーザ名が既にデータベースに存在する場合は、生成に失敗して false を返し、そうでない場合は true を返します：

```
1 public boolean createAccount(String username, String password) {
2     if (repository.findByUsername(username) == null) {
3         repository.save(new Account(username, password));
4         return true;
5     } else {
6         return false;
7     }
8 }
```

コントローラーの書き方

- login.html ページと LoginController のコードは、この前のコードとほぼ同じですが、今回はサービス層が使いたいのでその宣言を追加：

```
1 public class LoginController {
2     @Autowired
3     AccountService accountService;
4
5     @PostMapping("/login")
6     public ModelAndView login(@RequestParam String username,
7             @RequestParam String password, ModelAndView mav) {
8         if (accountService.validateAccount(username, password)) {
9             mav.addObject("name", username);
10            mav.setViewName("hello.html");
11        } else {
12            mav.setViewName("login.html");
13        }
14        return mav;
15    }
16    // ...
```

```
@Autowired
AccountService accountService;
```

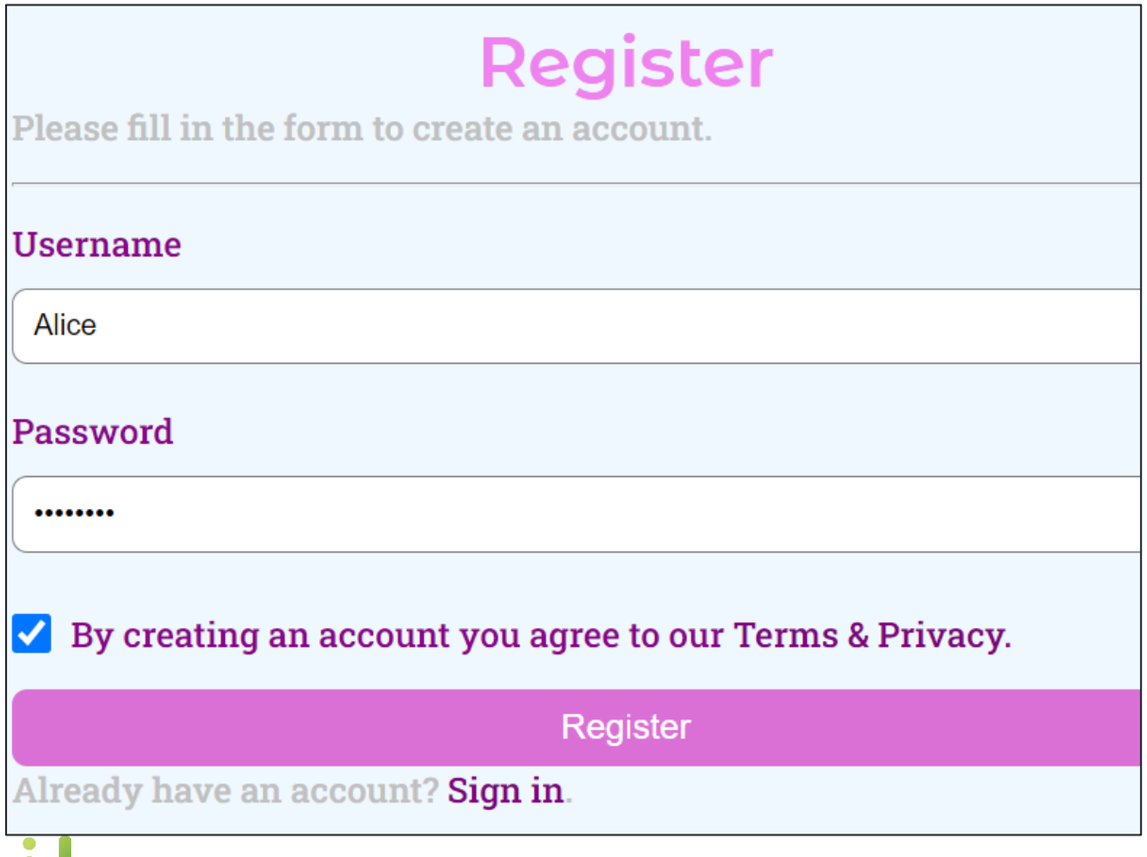
- Repository と同様に、Spring は自動的に Service クラスをインスタンス化させてくれるので、このオブジェクトのメソッドを直接利用することができます。
- 完成したら、サーバを起動して、いくつかのユーザー情報をデータベースに手動で追加してください。ブラウザでログインページの URL にアクセスし、動きを確認しましょう。

Woodland Academy

Q&A

演習：RegisterController の作成

- 登録画面 register.html とそれに対するコントローラ RegisterController がまだ完成していません。
- 既存の AccountService はどのように実現すべきかを考えて、自分で実現してみましょう！




```
jpa_test=# SELECT * FROM account;
 id | password | username
----+----------+----------
  1 | ABC87654 | Alice
(1 行)
```

Service 層の役割（1）

- 単純なアプリケーションでは、DAO 層の機能は Service の機能と非常によく似ていて、初心者ではよく、Service 層は DAO 層と同じように感じ、「Service 層がなくてもいいじゃないか？」と思っているかもしれません。

- 原則として、DAO 層は可能な限り**シンプル**であるべきです。データベースとの接続や簡単な生成・削除・ 更新・読み取りなどを提供し、Service 層はビジネスロジックが比較的複雑で、繰り返し利用される機能を提供すべきです。

Service 層の役割（2）

- Service 層を作ることにははいくつかのメリットがあります。
  - まず、**データベースを変更**する場合、DAO 層のコードだけを変更すれば良いのです。

  - 次に、ビジネスロジックが複雑な場合は、このコードを Service に実装してもらうことで、**コントローラのロジックを簡略化**することができます。

  - また、コントローラにはページの生成や HTTP リクエストに関するコードしか含まれていないため、Service 層のコードは**単独で独立テスト**をすることができます。

目次

JPA のその他の機能

- 基本的な生成、削除、更新、読み取りに加えて、JPA には他にも多くの機能を提供してくれます：
  - 各種のデータ型の対応
  - 制約
  - より多くのクエリ方法
  - 外部キーの対応
  - SQL 文を手動で設定
  - ……

- 授業で紹介されていない機能は、公式ドキュメントで確認しましょう：
  https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories

データタイプ

- Postgres と Java のデータタイプは、どのように対応しているのかを勉強していきましょう：

データタイプ	Postgres でのタイプ名	Java でのタイプ名
整数	INT	Integer
大きな整数	BIGINT	Long
自動増加な整数	SERIAL	Integer
自動増加な大整数	BIGSERIAL	Long
小数	NUMERIC	java.math.BigDecimal
文字列	VARCHAR	String
日付と時間	TIMESTAMP	java.sql.Timestamp
日付	DATE	java.sql.Date
時間	TIME	java.sql.Time
ブール値	BOOL	Boolean

Woodland Academy

外部キーの対応

- @OneToOne、@OneToMany、@ManyToOne などのアノテーションを追加すると、それぞれ属性を通じて一対一、一対多、多対一の外部キー対応を追加できます。例として、**@OneToOne** を紹介します：

```
1 @OneToOne(cascade = CascadeType.ALL)
2 @JoinColumn(name = "address_id", referencedColumnName = "id")
3 private Address address;
```

- これは、データベースにおいて、このフィールドが address テーブルのあるレコードに対応することを意味します。Java では、この属性の Address オブジェクトを直接使えます。

- **@JoinColumn** のパラメータは、このフィールドはテーブルの address_id というカラムで、address テーブルの id に対応していることを表します。

Woodland Academy

CRUD

- 次の表は、JPA がサポートする基本的な追加、削除、更新、読み取りメソッドの一覧です：

T:Entity　t:パラメター

メソッド	機能	引数	戻り値
save()	キーが存在する場合は保存 存在しない場合は、登録	T	T
findAll()	すべてのレコードを読み取り		List <T>
delete(All)ByXxx()	xxx フィールドでレコードを削除	t	int/void(All)
find(All)ByXxx()	xxx フィールドでレコードを検索	t	単一：T 複数/All:List<T>
existsByXxx()	xxx フィールドでレコードの存在を判定	t	boolean
countByXxx()	xxx フィールドでレコードの数をカウント	t	long

前へ

メソッド	例
save()	AdminEntity save(AdminEntity adminEntity);
findAll()	List<AdminEntity> findAll();
deleteByXxx()	int deleteByLastName(String lastName);
deleteAllByXxx()	void deleteAllByAgeLessThan(int age);
findByXxx():単一行取得	UserEntity findByLastName(String lastName);
findByXxx():複数行取得	List<UserEntity> findByLastName(String lastName);
findAllByXxx()	List<UserEntity> findAllByAgeGreaterThan(int age);
existsByXxx()	boolean existsByEmail(String email);
countByXxx()	long countByAgeGreaterThan(int age);

Woodland Academy

複雑なクエリ

- 複雑な条件を表現するため、色々な演算子を使えします：

```
List<Student> findAllByScoreIsNotNull();
List<Student> findAllByNameLike(String pattern);
List<Student> findAllByScoreGreaterThan(Integer min);
```

- 複数のクエリ条件を And などのキーワードで組み合わせることも可能です：

```
List<Student> findAllByNameLikeOrId(String pattern, Long id);
List<Student> findAllByScoreIsNotNullAndScoreGreaterThan(Integer minScore);
List<Student> findAllBySubjectOrderByAgeDesc(String subject);
```

Woodland Academy

Pageable 利用するクエリ

- **Pageable** オブジェクトを使用すると、すべてのデータが**ページ**に分けられ、クエリ結果のどのページを返すかは指定できます：

```
1 Pageable paging = PageRequest.of(2, 5);
2
3 Page<Product> productList = productRepository.findAll(paging);
```

- PageRequest.of() には、少なくとも 2 つのパラメータが必要です。1 番目の整数はクエリのページ番号（**0 から始まる**）を示し、2 番目の整数はページごとにいくつのレコードがあるのかを示します。例えば、PageRequest.of(2, 5) は、11 番目か ら 15 番目までのレコードを検索します。

SQL 文を手動での設定

- Spring の文法は我々の要求を満たさない場合もあって、データベース操作の **SQL 文を自分で書く**必要があります。

- 自分で書いた SQL クエリを実行するメソッドを設定するには、メソッドの先頭に **@Query** アノテーションを追加します：

```
@Query(value = "SELECT * FROM employee WHERE name = ?1",
        nativeQuery = true)
public List<Employee> findByName(String name);
```

- SQL クエリの中の「?1」や「?2」の記法は、メソッドの 1 番目、2 番目のパラメータを対応することを意味します。

Woodland Academy

Q&A

まとめ

Sum Up

1.Java におけるデータベース接続の標準：ORM と JPA。

2.ウェブアプリケーションを開発する際の 3 層モデル：Repository（DAO）、Service、Controller。

3.Spring で JPA を実装した Spring Data JPA：その基本文法と各アノテーションの意味。

Woodland Academy

Woodland Academy

Thank you!

From Seeds to Woodland — Shape Your Future.

Shape Your Future