Spring Boot

目次

データベースの利用

HSQLDB

使用する依存関係

- JPA（必須）
- WEB（必須）
- HSQLDB
- Thymeleaf（任意）

＜参考＞HSQLDB とは？

- Java で作成されたオープンソースのデータベースライブラリ
- Java で作成されているためアプリケーションにデータベースを内蔵させることができる

●実装

① エンティティクラス

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="mydata")
public class MyData {
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	@Column
	private long id;

	@Column(length = 50, nullable = false)
	private String name;

	@Column(nullable = false)
	private Integer age;



	// getter, setter
}
```

※補足

- @Entity アノテーション
  - エンティティクラスであることを表すアノテーション
  - 必須
- @Table アノテーション
  - テーブル名を指定するアノテーション
  - デフォルトではクラス名がテーブル名となるが明示的に記述したい場合に使用
- @Id
  - プライマリーキーを指定
  - 必須
- @GeneratedValue

- 主キーに対して値を自動生成する（シーケンスのイメージ）
- @Column
  - コラム名を指定
  - 省略可能で、デフォルトはフィールド名がそのままコラム名となる

② リポジトリクラス

```
package repositories;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.tuyano.springboot.MyData;

@Repository
    public interface MyDataRepository extends JpaRepository<MyData, Long>{
}
```

※補足

- @Repository アノテーションを付与する
- JpaRepository<T, U>クラスを継承する
  - T⇒エンティティクラスを指定する
  - U⇒主キーの型を指定する

③ コントローラークラス

```
@Controller
public class QueryController {
    @Autowired
    MyDataRepository repository;

    @RequestMapping("/select")
    public ModelAndView select(ModelAndView mav) {
            // DB検索 全件取得
            Iterable<MyData> list = repository.findAll();
            mav.addObject("list", list);
            // 遷移先を指定
            mav.setViewName("select");
            mav.addObject("title", "選択");
            return mav;

    }
```

※補足

- @Autowired アノテーション

- インタフェースを実装した無名クラスを Spring が自動生成してくれるようになる

リポジトリのメソッド自動生成

実装例

```
public Optional<SampleBean> findById(long id);
public List<SampleBean> findByIdAndName(long id, String name);
public List<SampleBean> findByIdOrId(long id1, long id2);
public List<SampleBean> findByIdBetween(long id1, long id2);
public List<SampleBean> findByIdLessThan(long id);
public List<SampleBean> findByIdGreaterThan(long id);
public List<SampleBean> findByIdIsNull();
public List<SampleBean> findByIdNotNull();
public List<SampleBean> findByNameLike(String name);
public List<SampleBean> findByNameLikeOrderByIdDesc(String name);
public List<SampleBean> findByIdNot(long id);
public List<SampleBean> findByIdIn(List<Long> id);
```

バリデーションの利用

●エンティティクラスにアノテーションを付与する

```
@Column(length=10, nullable=false)
@Size(min=1, max=10)
private String name;
```

※付与できるアノテーション一覧

@Null @NotNull	Null かどうかをチェックする
@Min @Max	整数の最大値、最小値を指定
@Digits(integer=整数桁数, fraction=小数桁数)	桁数を指定
@Future	未来日付であることをチェック
@Past	過去日付であることをチェック
@Size(min=最小要素数,max=最大要素数)	要素数の最大値、最小値を指定 （文字列の場合文字数）
@Pattern(regexp="正規表現")	正規表現パターンに合致しているかチェック

●View にエラーメッセージを表示する

```
<ul>
    <li th:each="error: ${#fields.detailedErrors()}" th:text="${error.message}" class="text-danger"></li>
</ul>
```

▽各フィールドにエラーを表示したい場合

```
<div class="col-sm-10">
        <input type="text" id="mail" name="mail" th:value="*{mail}" th:errorclass="err" class="form-control">
        <div th:if="${#fields.hasErrors('mail')}" th:errors="*{mail}" th:errorclass="${err}"></div>
</div>
```

ポイント！

- th:errorclass　⇒エラーが発生したときに生成されるインスタンスを指定
- th:if="${#fields.hasErrors('フィールド名')}"　⇒指定したフィールドにエラーが発生したかチェック
- th:errors="*{フィールド名}"　⇒指定したフィールドのエラーメッセージを表示する

●コントローラーの実装

```
@RequestMapping(value="/validate", method=RequestMethod.POST)
public ModelAndView validate(@ModelAttribute("target") @Validated ValidateTarget target
                                                                ,BindingResult result
                                                                ,ModelAndView mav) {
        if(!result.hasErrors()) {
                mav.addObject("result", "検証結果：ＯＫ");
        }else {
                mav.addObject("result", "検証結果：ＮＧ");
        }
        return mav;
}
```

ポイント！

- フォームオブジェクトに@Valudated アノテーションを付与する
- 引数に BindingResult を追加する
- result.hasErrors()でバリデーション結果を確認できる

●View に表示するテキストを変更する

resource フォルダ配下に「ValidationMessages.properties」を作成する

⇒このファイルがあれば自動で読み取ってくれる

▽実装

```
javax.validation.constraints.Max.message = Please input less than {value} value.

javax.validation.constraints.Min.message = Please input greater than {value} value.
```

ポイント！

- 左辺にはアノテーションクラスを指定（パッケージ込み）
- 右辺には表示したいテキストを指定

オリジナルのバリデータの作成

※実装例として携帯電話番号のバリデートを考える

▽実装例（アノテーションクラス）

```
package com.tuyano.springboot;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;
import javax.validation.ReportAsSingleViolation;

@Documented
@Constraint(validatedBy=PhoneValidator.class)
@Target({ ElementType.METHOD, ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
@ReportAsSingleViolation
public @interface Phone {
	String message() default "please input a phone number !";

	Class<?>[] groups() default {};

	Class<? extends Payload>[] payload() default {};
}
```

ポイント！

- 基本この通りの実装になる
- @Constraint の引数はバリデータクラスを指定
- message()はエラー時に表示したいメッセージを指定

▽実装例（バリデータクラス）

```
package com.tuyano.springboot;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class PhoneValidator implements ConstraintValidator<Phone, String> {
	@Override
	public void initialize(Phone phone) {
		// 必要に応じてアノテーションの情報を取得＆初期化
	}
	@Override
	public boolean isValid(String value, ConstraintValidatorContext context) {
		if(value == null) {
			return false;
		}
		return value.matches("[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]");
	}

}
```

ポイント！

- isValid メソッドにバリデート内容を記述する

ORACLE 接続

●手順

プロジェクト作成～ソースの修正まで

① SpringBoot スタータープロジェクトを作成

- 以下の依存関係にチェック
  - JPA（必須？）
  - WEB（必須）
  - Themeleaf（任意）

② ojdbc.jar を配置

- プロジェクト配下に「lib」フォルダを作成
- 「lib」フォルダ配下に ojdbc.jar を配置
- プロジェクトルート＞lib＞ojdbc,jar

③ pom.xml を修正

- 以下の情報を追記

```
<dependency>
        <groupId>com.oracle</groupId>
        <artifactId>ojdbc6</artifactId>
        <version>11.0.0</version>
        <scope>system</scope>
        <systemPath>${basedir}/lib/ojdbc6.jar</systemPath>
</dependency>
```

④ application.properties を修正

```
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:XE
spring.datasource.username=SAMPLE
spring.datasource.password=SAMPLE
spring.datasource.driverClassName=oracle.jdbc.driver.OracleDriver
```

SPRING DATA JPA フレームワーク

基本構成の実装

▽DAO インタフェースの実装

```
package com.tuyano.springboot;

import java.io.Serializable;
import java.util.List;

public interface DaoBase<T> extends Serializable {
	public List<T> getAll();
}
```

ポイント！

- Serializable を継承する
- メソッドは DAO クラスで統一させたいものを定義

▽DAO クラスの実装

```
package com.tuyano.springboot;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.Query;

public class SampleDAO implements DaoBase<SampleBean>{
	private EntityManager manager;

	public SampleDAO() {
		super();
	}
	public SampleDAO(EntityManager manager) {
		this();
		this.manager = manager;
	}

	@Override
	public List<SampleBean> getAll() {
		// select * from tbl_sample
		Query query = manager.createQuery("from SampleBean");
		List<SampleBean> list = query.getResultList();
		manager.close();
		return list;
	}

}
```

ポイント！

- EntityManager クラス
  - エンティティを操作するための機能を持っている
  - DAO にフィールドとして用意する
- manager.createQuery
  - JPQL を利用し SQL を自動生成してくれる
  - 「from エンティティクラス」でエンティティクラスで Table で指定したテーブルを見に行ってくれる

▽コントローラーの実装

```
package com.tuyano.springboot;

import java.util.List;

import javax.annotation.PostConstruct;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class DAOController {
	@PersistenceContext
	EntityManager manager;

	SampleDAO dao;

	@PostConstruct
	public void init() {
		dao = new SampleDAO(manager);
	}

	@RequestMapping("/dao")
	public ModelAndView index(ModelAndView mav) {
		List<SampleBean> samples = dao.getAll();
		mav.addObject("samples", samples);
		mav.setViewName("dao");
		return mav;
	}

}
```

ポイント！

- @PersistenceContext アノテーション
  - EntityManager オブジェクトに付与する
  - Spring が自動で生成した EntityManager オブジェクトを設定してくれる
  - このアノテーションは複数置くことができない

    ⇒複数利用される可能性がある DAO クラスではなく、コントローラーで使用
- DAO インスタンスの生成
  - インスタンスの生成に EntityManager クラスを使用するため、初期化メソッド内で実装

●バインド変数の利用

▽DAO クラスの実装

```
	public List<SampleBean> selectByPkUseBind(long id) {
		String qstr = "from SAMPLE where id = :id";
		Query query = manager.createQuery(qstr)
				.setParameter("id", id);
		return query.getResultList();

	}
```

ポイント！

- SQL に「:変数名」の形で組み込む
- query.setParameter(“変数名”, 値)で値をセットする

●@NamedQuery の利用

▽Bean クラスの実装

```
@Entity
@Table(name="TBL_SAMPLE")
@NamedQueries(
	@NamedQuery(
		name="findById",
		query="from SampleBean where id = :id"
	)
)

public class SampleBean implements BeanBase{

…
```

ポイント！

- @NamedQuery アノテーションの付与
  - クラス定義の直前に実装
  - name　⇒クエリーの名前を定義
  - query　⇒クエリー文を記述
- @NamedQueries アノテーションの付与
  - @NamedQuery を複数用意したい場合に使用

▽DAO クラスの実装

```
	public List<SampleBean> namedQuery(long id){
		Query query = manager.createNamedQuery("findById ")
				.setParameter("id", id);
		return query.getResultList();

	}
```

ポイント！

- Query インスタンスの取得は「manager.createNamedQuery(“クエリー名”)」

●@Query の利用

@NamedQuery がエンティティクラスに付与したのに対し、

@Query はリポジトリインタフェースのメソッドに付与する

▽リポジトリクラスの実装

```
// import org.springframework.data.jpa.repository.Query;
        @Query("from SampleBean where id > 5 order by id desc")
        public List<SampleBean> query();
```

CRITERIA API による検索

●CriteriaAPI で使用する3クラス

CriteriaQuery クラス	クエリー生成を管理する
CriteriaQuery クラス	クエリー実行のためのクラス
Root クラス	検索されるエンティティのルートとなるクラス ここからエンティティを絞り込む

●DAO クラスの実装

```
	public List<SampleBean> getAll() {
		CriteriaBuilder builder = manager.getCriteriaBuilder();
		CriteriaQuery<SampleBean> query = builder.createQuery(SampleBean.class);
		Root<SampleBean> root = query.from(SampleBean.class);
		query.select(root);

		return manager.createQuery(query).getResultList();

	}
```

ポイント！

- CriteriaAPI の利用順序は基本的に以下の通り
  - CriteriaBuilder の取得
  - CriteriaQuery の生成
  - Root の取得
  - CriteriaQuery メソッドを実行
    - エンティティを絞り込むためのメソッドを実行する
    - 必要に応じてメソッドチェーンで呼び出す
  - createQuery⇒getResultList で検索の実行
    - 通常の検索の実装と同じ
    - ただし、引数に CriteriaQuery を渡す

●CriteriaBuilder のメソッド一覧

メソッド	対応する SQL
<CriteriaQuery>.where 内で使用	
equal	= :1
notEqual	<> :1
gt, greaterThan	> :1
ge, greaterthanOrEqualTo	>= :1
lt, lessThan	< :1
le, lessThanOrEqualTo	<= :1
between	between :1 and :2
isNull	is null
isNotNull	is not null
isEmpty	is null or length() = 0
isNotEmpty	is not null and length() <> 1
like	like ':1'
and(Predicate ...)	and
or(Predicate ...)	or
<CriteriaQuery>.orderBy 内で使用	
asc	asc
desc	desc

▽DAO クラス　-where メソッドの実装例

```
query.select(root)
        .where(builder.like(root.get("name"), "%" + name + "%"));
```

▽DAO クラス -orderBy メソッドの実装例

```
query.select(root)
        .orderBy(builder.desc(root.get("id")));
```

●取得位置と取得個数の設定

▽DAO クラスの実装

```
List<UserBean> list = manager.createQuery(query)
                .setFirstResult(first)
                .setMaxResults(count)
                .getResultList();
```

ポイント！

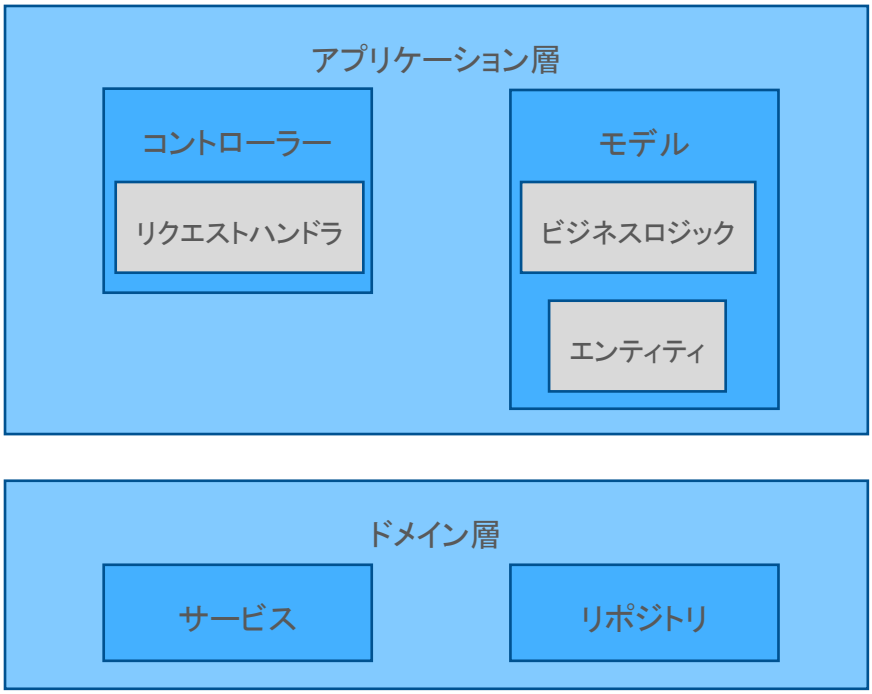
- setFirstResult(取得位置)で取得位置＋１件目からレコードを取得
- setMaxResults(件数)で指定した件数分だけ結果を取得

※エンティティの連携

サービスとコンポーネント

Spring らしく、DI を用いて実装する

▽構成概要図



最小限の実装

テーブルからレコードを全件取得するサービスを作成＆利用する

▽サービスクラスの実装

```
package com.tuyano.springboot;

import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import org.springframework.stereotype.Service;

@Service
public class SampleDataService {
	@PersistenceContext
	private EntityManager manager;

	public List<UserBean> getAll(){
		return (List<UserBean>) manager.createQuery("from UserBean").getResultList();
	}
}
```

ポイント！

- @Service アノテーション
  - サービスとして登録するクラスに付与する

▽コントローラークラスの実装

```
	@Autowired
	private SampleDataService service;
	@RequestMapping("/service/getAll")
	public ModelAndView getAll(ModelAndView mav) {
		List<UserBean> users =  service.getAll();
		mav.addObject("users", users);
		mav.setViewName("service1");
		return mav;
	}
```

ポイント！

- @Servie アノテーションを付与したクラスは自動で Bean 化されている
- @Autowired アノテーションをフィールド変数に付与することで自動で割り振れる

コンポーネントの利用

▽コンポーネントクラスの作成

```
package com.tuyano.springboot.component;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.ApplicationArguments;
import org.springframework.stereotype.Component;

@Component
public class SampleComponent {
	private int counter;

	@Autowired
	public SampleComponent(ApplicationArguments arg) {
		this.counter = 0;
	}

	public int count() {
		return this.counter++;
	}
}
```

ポイント！

- コンポーネントとしてアプリケーションに認識させるには@Component を付与する
- @Autowired を付与したコンストラクタが自動で呼び出される

▽コントローラークラスで利用する

```
package com.tuyano.springboot.component;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class ConponentController {
	@Autowired
	SampleComponent component;

	@RequestMapping("/component/count")
	public ModelAndView count(ModelAndView mav) {
		mav.addObject("count", component.count());
		mav.setViewName("component/count");
		return mav;
	}

}
```

ポイント！

- コンポーネントに@Autowired を付与することで自動でインスタンス化してくれる

構成クラスの利用

構成クラスを利用することで POJO クラスを Bean 登録して利用することができる

▽Bean クラスの作成

```
package com.tuyano.springboot.component;

public class SampleBean {
	private int counter = 0;

	public int count() {
		return this.counter++;
	}

}
```

▽構成クラスの作成

```
package com.tuyano.springboot.component;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
	@Bean
	public SampleBean get() {
		return new SampleBean();
	}
}
```

ポイント！

- @Configuration アノテーションを付与することでアプリが構成クラスだと認識する
- 取得したい Bean クラスを返すメソッドを作成する
  - メソッド名は任意
  - @Bean アノテーションによってメソッドが実行されてアプリに Bean 登録される（？）

覚えておきたいその他の機能

ページネーション

DB の検索結果に対して「〇件目から〇件取得したい」というときに便利

※準備中

実行 SQL のコンソール出力

リポジトリの SQL 自動生成をした際に SQL を確認するために使える

▽appliction.properties

```
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.use_sql_comments=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.type=trace
```