Spring Boot

目次

[データベースの利用 2](#_Toc514013527)

[HSQLDB 2](#_Toc514013528)

[リポジトリのメソッド自動生成 5](#_Toc514013529)

[バリデーションの利用 5](#_Toc514013530)

[オリジナルのバリデータの作成 7](#_Toc514013531)

[Oracle接続 8](#_Toc514013532)

[SPring Data JPAフレームワーク 9](#_Toc514013533)

[基本構成の実装 9](#_Toc514013534)

[Criteria APIによる検索 14](#_Toc514013535)

[※エンティティの連携 16](#_Toc514013536)

[サービスとコンポーネント 17](#_Toc514013537)

# 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

1. 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

1. 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）

1. 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

1. 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> |

1. 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

ドメイン層

アプリケーション層

サービス

リポジトリ

コントローラー

モデル

リクエストハンドラ

ビジネスロジック

エンティティ

## 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