フレームワーク基礎実習

09. フォームとバリデーション

株式会社ジードライブ

今回学ぶこと

- Thymeleafのフォーム
- SpringとHibernate Validatorによるバリデーションの実装
  - アノテーション
  - エラーメッセージの指定
  - バリデーショングループ

Thymeleafのフォーム

- Thymeleafはフォーム用の属性(th:field, th:errors 等)をもっており、初期値の設定やエラー表示が容易になる
  - Springのバリデーション機能と連携できる

Thymleafを利用したフォームの例

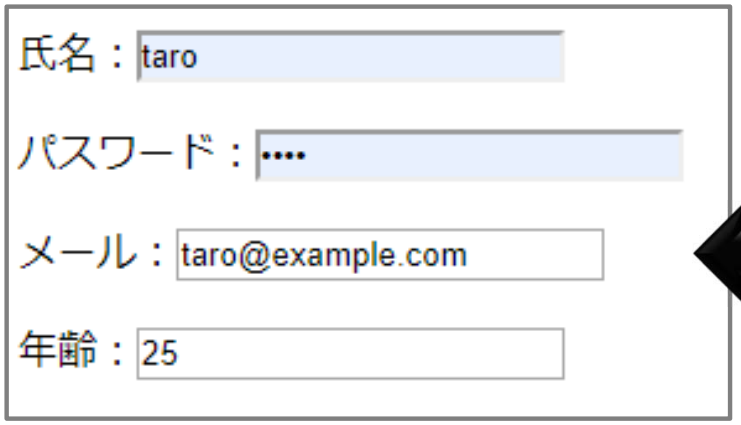
```
<form th:action="@{/process}" method="post" th:object="${user}">
    <span th:errors="*{name}"></span>
    <p>氏名：<input type="text" th:field="*{name}"></p>

    <p>種別：
       <select th:field="*{type.id}">
           <option th:each="type : ${typeList}" th:value="${type.id}">
           [[${type.name}]]</option >
       </select>
    </p>
    <input type="submit">
</form>
```

利用の準備（1）

入力値保持用のクラス

- 入力内容に対応したフィールドを持つクラスを用意する




```
public class FormItems {

  private String name;
  private String pass;
  private String mail;
  private Integer age;

  // アクセッサ
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  ...
```

Springがgetter及びsetterを呼び出すので、アクセッサは必須
また、デフォルトコンストラクタも必要になる

利用の準備 (2)

コントローラークラス

- フォームに対応するオブジェクトをThymeleafで利用できるようにする(Model#addAttribute()を記述する)
- @ModelAttributeアノテーションを使い、入力値を取得する

```
// Getリクエスト時（フォーム生成時の準備）
@GetMapping("/input")
public String inputGet(Model model) {
  model.addAttribute("items", new FormItems());
  ...
}

// Postリクエスト時（フォームから入力値を取得）
@PostMapping("/input")
public String inputPost(@ModelAttribute("items") FormItems items) {
  ...
}
```

Thymeleafで利用できるようにする

引数名と一致しているので、("items")は省略可

利用の準備 (3)

Thymeleaf

- th:objectの属性値として、先に準備したオブジェクトを指定する
- フォーム部品のth:field属性の値として、オブジェクトのフィールドを指定する

```
<form th:action="@{/input}" method="post" th:object="${items}">
    <p>氏名：<input type="text" th:field="*{name}"></p>
    <p>パスワード：<input type="password" th:field="*{pass}"></p>
    <p>メール：<input type="email" th:field="*{mail}"></p>
    <p>年齢：<input type="number" th:field="*{age}"></p>
    <input type="submit">
</form>
```

th:action / th:object

- formタグに設定できる属性
  - th:object属性はformタグ以外にも設定可能
- th:action属性で、送信先のURLを指定する
  - 記述方法はth:hrefやth:srcと同じ
  - この属性を使わず、通常のaction属性を使用することも可能
    ⇒ Spring Securityの提供するCSRF対策を利用する場合は、th:actionを使用する
- th:object属性で、フォーム内で使用するオブジェクトを指定する

記述例

```
<form th:action="@{/input}" th:object="${items}"
          method="post">
      <p>氏名：<input type="text" th:field="*{name}"></p>
</form>
```

th:field

- フォームの部品に設定する属性で、オブジェクト名とフィールド名を指定する
  - 一行入力欄以外にも、テキストエリア、ラジオボタン、チェックボックス等で使用可能

記述例

```
<form action="">
      氏名：<input type="text" th:field="${items.name}">
</form>
```

th:object使用時の記述例

```
<form action="" th:object="${items}">
      氏名：<input type="text" th:field="*{name}">
</form>
```

itemsオブジェクトを参照

th:fieldの効果

- name属性の代わりにth:fieldを設定することで、自動的に初期値の設定が行われる
  - バリデーションエラー等で、フォームを再表示しなければならない場合の記述が容易になる
  - チェックボックスやラジオボタンなどでも有効

Thymeleafの記述

```
<input type="text" th:field="name">
<input type="number" th:field="age">
<input type="password" th:field="pass">
```

HTML変換後の出力例

```
<input type="text" id="name" name="name" value="山田太郎">
<input type="number" id="age" name="age" value="25">
<input type="password" id="pass" name="pass" value="">
```

パスワードは初期値が設定されない

th:fieldのネスト

- 以下のように、FormItemsはArea型のareaフィールドをもつとする

```
@Data
public class FormItems {
    private Integer price;
    private Area area;
}
```

```
@Data
public class Area {
    private Integer id;
    …
}
```

- この時、th:fieldの属性値を「area」とすると型の問題が発生するが、「area.id」とすることで問題が解消され、送信データを取得することができる

```
<form action="" method="post" th:object="${items}">
   <p>価格：<input type="number" th:field="*{price}"></p>
   <p>エリア：
      <input type="radio" th:field="*{area.id}" value="1">国内
      <input type="radio" th:field="*{area.id}" value="2">海外
   …
```

フォームの初期値設定

- Thymeleafのフォーム(th:field) に初期値をセットしたい場合、コントローラー側で値を準備しておく

```
// Getリクエスト時（フォーム生成時の準備）
@GetMapping("/input")
public String inputGet(Model model) {
  FormItems items = new FormItems();

  items.setPrice(0);
  items.setCreated(new Date());
  items.setArea(new Area(2, null));

  model.addAttribute("items", items);
  ...
}
```

フォームの初期値となるデータを各フィールドにセット

初期値設定済みのオブジェクトをThymeleafで使えるようにする

th:value

- value値に任意のデータを設定したい場面で使用する

Thymeleafの記述

```
<p>エリア：
      <label th:each="name, status : ${areaList}">
         <input type="radio" th:field="*{area.id}"
                th:value="${status.count}">[[${name}]]
      </label>
</p>
```

この例では、th:eachの周回情報をvalue値として設定している

HTML変換後の出力例

```
<p>エリア：
      <label><input type="radio" value="1" id="area.id1"
                                  name="area.id">国内</label>
      <label><input type="radio" value="2" id="area.id2"
                                  name="area.id">海外</label>
</p>
```

checkboxの扱い

- 複数選択可能なチェックボックスの場合、入力値取得用のフィールドをListまたは配列にしておく

入力値取得用クラス

```
@Data
public class FormItems {
    private List<Integer> seasons;
    …
}
```

Thymeleaf

```
<form action= th:object=${items}>
  <input type="checkbox" th:field="*{seasons}" value="1">春
  <input type="checkbox" th:field="*{seasons}" value="2">夏
  <input type="checkbox" th:field="*{seasons}" value="3">秋
  <input type="checkbox" th:field="*{seasons}" value="4">冬
  …
```

checkboxの扱い

- 単一選択のチェックボックスでは、値を真偽値で取得することができる

入力値取得用クラス

```
@Data
public class FormItems {
  private Boolean privacy;
  ...
}
```

Thymeleaf

```
<form action= th:object=${items}>
   <input type="checkbox" th:field="*{privacy}">
    プライバシーポリシーに同意する
   ...
```

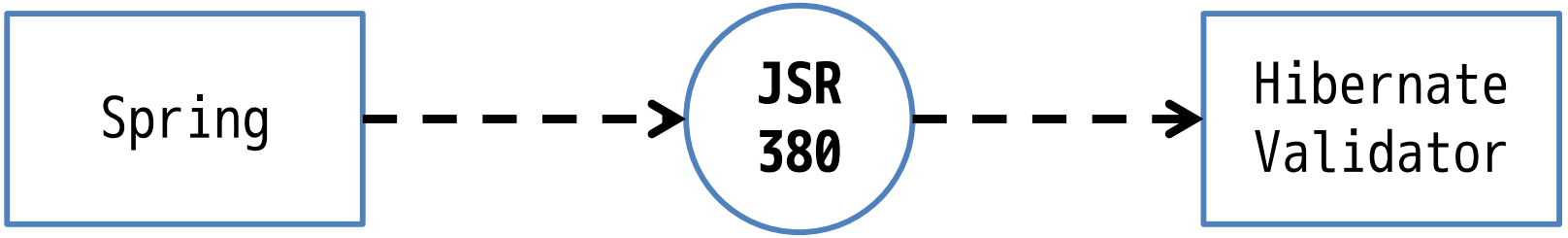
練習

- 練習09-1

バリデーション

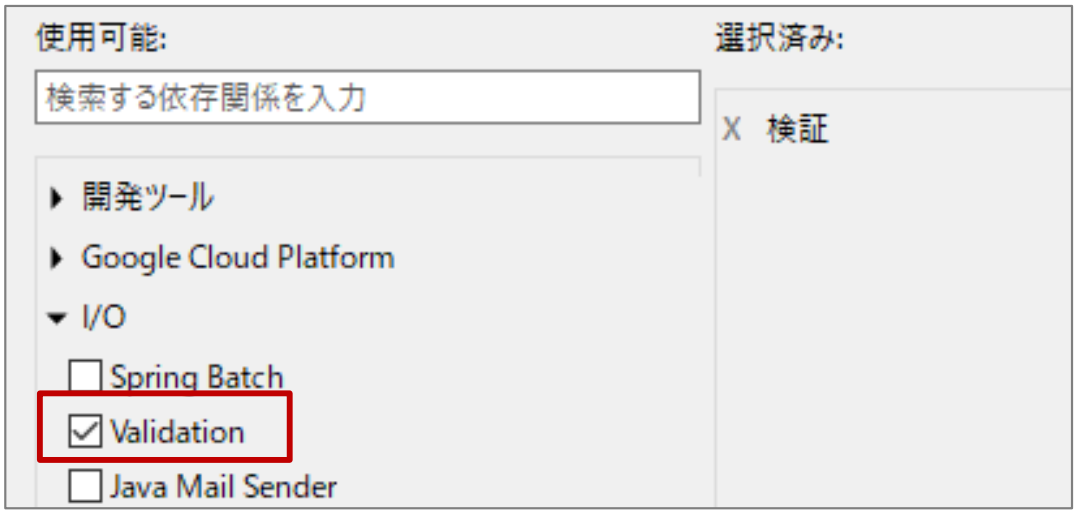
Hibernate Validator

- Hibernateというオープンソースコミュニティが作成したアノテーションによるバリデーションのライブラリ
  - Javaのアノテーションによるバリデーション仕様として、**JSR 380: Bean Validation**というものがあり、これを実装している
  - Springはこの「JSR 380: Bean Validation」と連携する仕組みを持つ



Hibernate Validatorの導入

- 新規プロジェクト作成時に、依存関係に Validation を加える



- プロジェクト作成済みの場合、以下をpom.xmlに追記

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

バリデーションの方法

- ドメインクラスに対し、アノテーションを記述することで、バリデーションが行われる

```
public class Item {

  private Integer id;

  @NotBlank
  @Size(min = 5)
  private String name;

  @Min(0)
  private Integer price;
  ...
```

空白を許可しない

5文字以上

0以上

基本的にインポートは、**jakarta.validation.constraints**パッケージから行う
一部、org.hibernate.validator.constraintsからインポートする場合もある

バリデーション用アノテーション

アノテーション	チェック内容	例
@NotNull	nullではないこと	
@Max	数値の最大値	@Max(100)
@Min	数値の最小値	@Min(0)
@Size	文字数やCollectionの要素数	@Size(min=1, max=20)
@Future @FutureOrPresent	未来であること	
@Past @PastOrPresent	過去であること	
@AssertTrue	trueであること	
@AssertFalse	falseであること	
@Pattern	正規表現にマッチすること	@Pattern(regexp="¥¥d{3}-¥¥d{4}")

バリデーション用アノテーション

アノテーション	チェック内容	例
@NotEmpty	文字列やCollectionがnullまたは空ではないこと(文字列の場合、空白文字があればEmptyとみなされない)	
@NotBlank	文字列がnullまたは空ではないこと(空白文字だけの場合はBlankとみなされる)	
@Length	文字数の範囲	@Length(min=0, max=100)
@Range	数値の範囲	@Range(min=0, max=100)
@Email	文字列がEメール形式であること	
@CreditCardNumber	文字列がクレジットカード番号形式であること	
@URL	文字列がURL形式であること	
@Valid	ネストされた要素のアノテーションを有効にする	

数値の入力欄の場合、NotEmptyやNotBlankではなく、NotNullを使用する

バリデーション処理の実装

- コントローラー用メソッドのドメインオブジェクト引数の前に@Validを付ける

```
@GetMapping("/addItem")
public String addGet(Model model) {
        model.addAttribute("item", new Item());
        return "add";
}

@PostMapping("/addItem")
public String addPost(@Valid @ModelAttribute("item") Item item) {
      … バリデーション処理の記述 …
}
```

この３か所がドメインクラス名のキャメルケースになっている場合、青字部分は省略可能

インポートは、**jakarta.validation**パッケージから行う

バリデーション処理の実装

- エラー情報はErrors型(または、BindingResult型)の引数で受け取る
  - **@Validの次の引数として記述する**

```
@PostMapping("/addItem")
public String addItem(@Valid @ModelAttribute("item") Item item,
                      Errors errors,
                      Model model) {

  if (errors.hasErrors()) {
    return "errorPage";
  }
  ...
```

@Validの次に記述

Errorsオブジェクトを使い、
バリデーション処理を実装する

インポートは、**org.springframework.validation**パッケージから行う

バリデーション処理の実装

- Errors / BindingResultのおもなメソッド

メソッド	説明
boolean hasErrors()	バリデーションでエラーがあればtrueを返す
int getErrorCount()	エラーの総数を取得する
List<FieldError> getFieldErrors(String field)	フィールドエラー(各入力項目ごとのエラー)を取得する
List<ObjectError> getGlobalErrors()	グローバルエラーのリストを取得する (「ログインIDとパスワード」のように複数の入力項目の組み合わせによって生じるエラー)
List<ObjectError> getAllErrors()	フィールドエラーとグローバルエラーを取得する
void reject(String errorCode)	グローバルエラーを追加する
void rejectValue(String field, String errorCode)	フィールドエラーを追加する

エラーメッセージの出力

- エラーメッセージの出力を行う場合は、th:errors属性を使用する
  - 属性値はth:fieldと同じ
  - 複数のメッセージがある場合、brタグで区切られる
    ⇒ メッセージの順番は決まっていない

```
<form action="" method="post" th:object="${item}">
    <p th:errors="*{name}"></p>
    <p>名前：<input type="text" th:field="*{name}"></p>
    <input type="submit">
</form>
```

HTML変換後の出力例

```
<p>空白は許可されていません
<br />5 から 2147483647 の間のサイズにしてください</p>
```

エラーメッセージの出力

- メッセージをbrタグで区切りたくない場合や１つだけ表示させたい場合は、暗黙オブジェクト#fieldsを使用する
  - #fields.hasErrors() でエラーの有無を確認する
  - #fields.errors() でエラーメッセージを配列で取得する

全てのメッセージを出力

```
<p th:if="${#fields.hasErrors('name')}"
     th:each="err : ${#fields.errors('name')}"
     th:text="${err}"></p>
```

メッセージを１つだけ出力

```
<p th:if="${#fields.hasErrors('name')}"
     th:text="${#fields.errors('name')[0]}"></p>
```

ネストされた要素のバリデーション

- ネストされた要素のアノテーションを有効にするには、@Validアノテーションを使用する

親となる要素

```
public class Item {

  @NotBlank
  private String name;

  @Valid
  private ItemDetail detail;
  ...
```

ネストされた要素

```
public class ItemDetail {

  @Min(0)
  private Integer price;
  ...
```

Thymeleafの記述例

```
<form action="" method="post" th:object="${item}">
   <span th:errors="*{detail.price}"></span>
   <input type="number" th:field="*{detail.price}">
   ...
```

練習

- 練習09-2

エラーメッセージの指定

message属性による指定

- アノテーションにmessage属性を記述することで、エラーメッセージをカスタマイズすることができる
  - 特に指定がない場合は、あらかじめ用意されているエラーメッセージが設定される

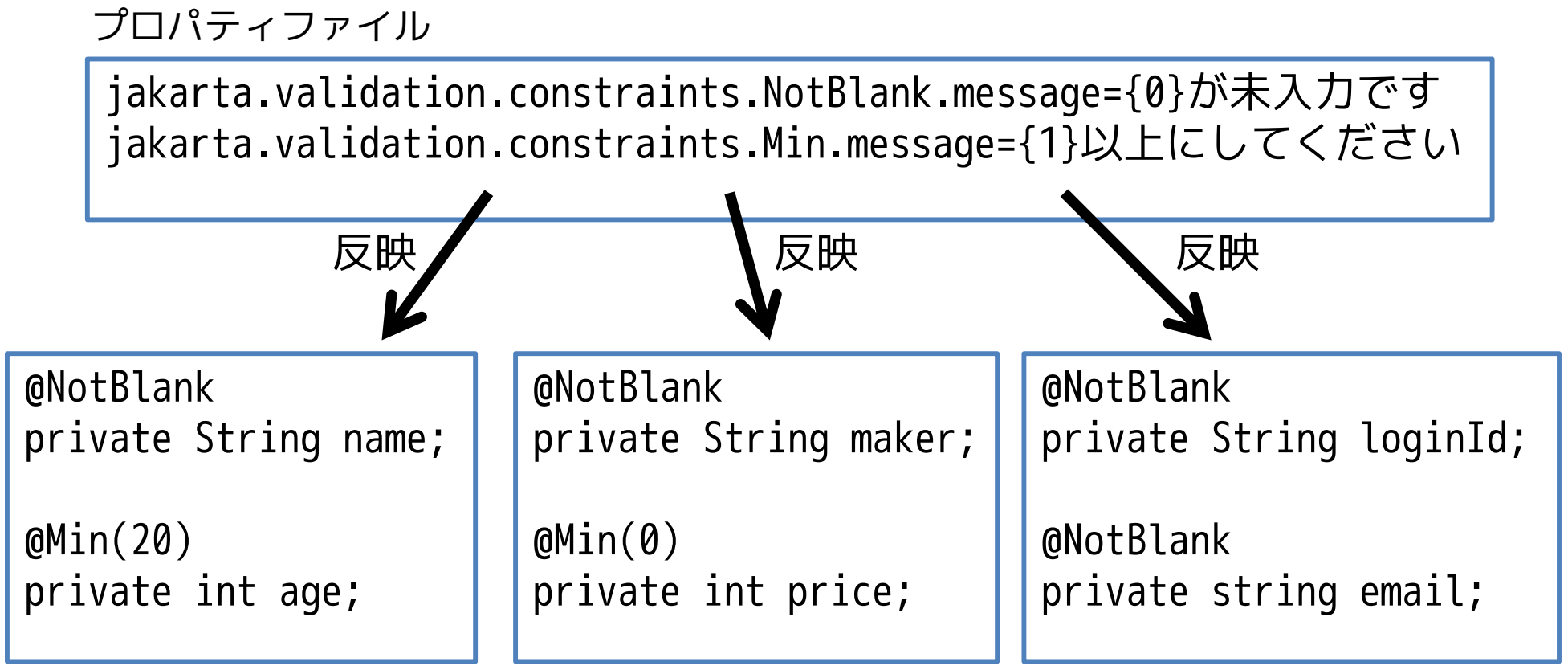
```
public class Item {

  private Integer id;

  @NotBlank(message="必須項目です")
  @Length(max=20, message="{max}文字以内で入力してください")
  private String name;
  ...
```

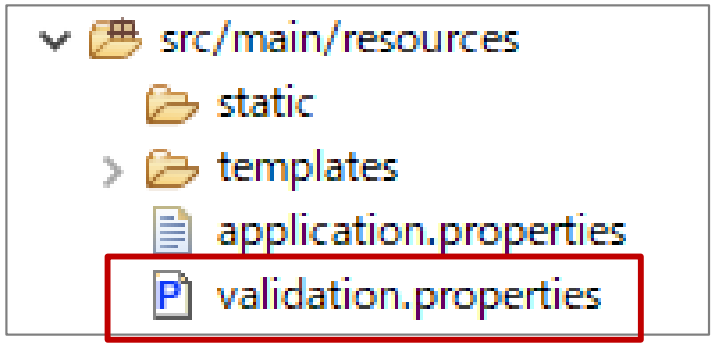
プロパティファイルによる指定

- 各アノテーションではなく、プロパティファイルでエラーメッセージを設定することで、メッセージの一元管理ができるようになる



プロパティファイルによる指定

- バリデーションメッセージ用のプロパティファイルは、src/main/resources以下に配置する（ファイル名は任意）



- プロパティファイルには、アノテーションとそれに対応するメッセージを記述する

@NotBlankに対するメッセージの設定例

```
jakarta.validation.constraints.NotBlank.message=入力必須項目です
```

プロパティファイルの有効化

- WebMvcConfigurerを実装した設定ファイルを作成し、以下を記述する

```
@Configuration
public class ValidationConfig implements WebMvcConfigurer {

    @Override
    public Validator getValidator() {
        var validator = new LocalValidatorFactoryBean();
        validator.setValidationMessageSource(messageSource());
        return validator;
    }

    @Bean
    MessageSource messageSource() {
        var messageSource = new ResourceBundleMessageSource();
        messageSource.setBasename("validation");
        return messageSource;
    }

}
```

org.springframework.validation/パッケージからインポート

validation.propertiesというファイル名の場合

プロパティファイルの記述例

SpringBootバージョン2系を使用する場合は、jakarta⇒javaxに変更する

{0}：フィールド名が入る
{value}：@Minや@Maxで設定した値が入る

```
jakarta.validation.constraints.NotBlank.message={0}が未入力です
jakarta.validation.constraints.Min.message={value}以上にしてください
jakarta.validation.constraints.Max.message={value}以下にしてください
jakarta.validation.constraints.Size.message={0}は{min}～{max}文字で入力してください
typeMismatch.java.lang.Integer={0}は整数で入力してください

item.name=商品名
user.name=氏名
age=年齢
```

エラーコードによる指定も可能

{0} のフィールド名に対応する日本語の文字列を指定

エラーコード

- エラーコードについては、Errors#getAllErrors()で取得できるObjectErrorから知ることができる

```
@PostMapping
public String formPost(@Valid FormItem formItem, Errors errors) {
    if(errors.hasErrors()) {
        // エラー内容の捕捉
        List<ObjectError> objList = errors.getAllErrors();
        for(ObjectError obj : objList) {
            System.out.println(obj.toString());
        }
    }
   …
```

出力例 (太字部分がエラーコード)

```
Field error in object 'formItem' on field 'name': rejected value []; codes
[NotBlank.bookingFormItem.name,NotBlank.name,NotBlank.java.lang.String,NotBla
nk];
```

Errorsオブジェクトの利用

エラー内容の出力

- Errorsオブジェクトを使用することで、生じているエラーコードなどの詳細情報を調べることができる

```
@PostMapping
public String contactPost(@Valid User user,  Errors errors, Model
model) {
   System.out.println("エラー数：" + errors.getErrorCount());

   // エラーの詳細を調べる
   List<ObjectError> allErrors = errors.getAllErrors();
   for(ObjectError error : allErrors) {
      System.out.println(error);
   }

   ...
```

カスタムエラーの追加

- 独自のエラー情報を追加することも可能

特定フィールドのエラー

```
errors.rejectValue(フィールド名, エラーコード);
```

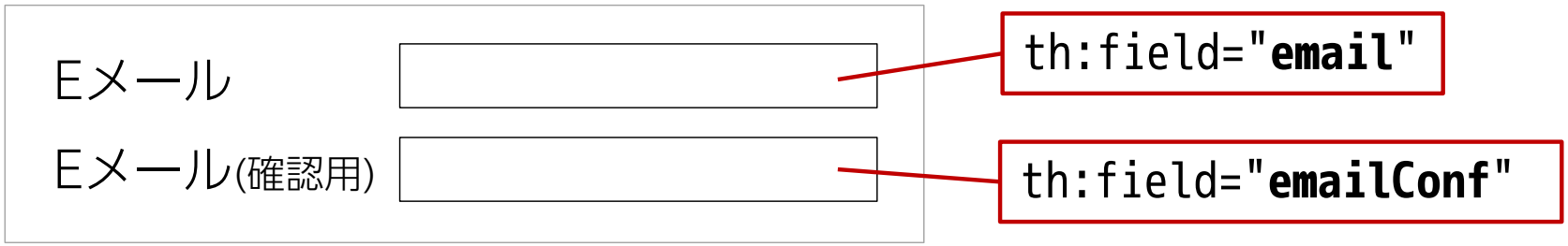
グローバルエラー

```
errors.reject(エラーコード);
```

  - フィールド名はバリデーション対象オブジェクトのフィールド名を文字列として指定する
  - エラーコードは任意の文字列で設定する
    例：`error.category.invalid`　など
    ⇒ エラーコードに対応するメッセージをプロパティファイルに設定
  - グローバルエラーは、特定のフィールドに紐づかないエラー
    ⇒ 例：ログインIDとパスワードの組み合わせが正しくない

フィールドエラーの追加例

入力欄



コントローラー

```
@PostMapping("/registerUser")
public String registerUser(
                    @Valid User user, Errors errors, Model model) {
    if(!user.getEmail().equals(user.getEmailConf())) {
        errors.rejectValue("email", "error.email.inequal");
    }
    ...
```

emailのフィールドエラーを追加
⇒ th:errors="*{email}" でエラーメッセージが表示される

プロパティファイル

```
error.email.inequal=確認用のメールアドレスと一致していません
```

グローバルエラーの出力

- #fields.hasGlobalErrors() でグローバルエラーの有無を確認する
  - 代わりに #fields.hasErrors('global') や #fields.hasAnyErrors() で確認することもできる
- #fields.globalErrors() でグローバルエラーのリストを取得することができる

グローバルエラーの出力例

```
<div th:object="${user}">
  <p th:errors="*{email}"></p>
  <th:block th:if="${#fields.hasGlobalErrors()}">
    <p th:each="error : ${#fields.globalErrors()}"
       th:text="${error}"></p>
  </th:block>
</div>
```

練習

- 練習09-3

バリデーショングループ

バリデーショングループ

- バリデーションの内容をグループ分けする機能
  - あるオブジェクトのバリデーションについて、特定の場面でだけ特定のバリデーションを行いたいという時に便利

例：バリデーショングループを利用したい場面の例

```
public class User {

    private String loginId;

    private String loginPass;

    private String name;

    private int age;
    …
```

会員登録時は、未入力チェックと文字数チェックを行いたいが、ログイン時は未入力チェックのみ行いたい

会員登録時は、未入力チェックや文字数チェックなどを行いたいが、ログイン時はチェックする必要がない

バリデーショングループの使い方

- 準備としてグループを表すインターフェースを作成する
  - マーカーインターフェースとして作成できる
    (メソッドの定義は不要)

例：ユーザ登録時のグループとログイン時のグループ

RegisterGroup.java

```
public interface RegisterGroup {
    //メソッドの定義は不要
}
```

LoginGroup.java

```
public interface LoginGroup {
    //メソッドの定義は不要
}
```

バリデーショングループの使い方

- ドメインクラスでは、バリデーション用アノテーションに**groups属性**を付与し、グループ指定をする

例：User.java

```
public class User {
    @Size(max=20, groups={RegisterGroup.class}))
    @NotBlank(groups={RegisterGroup.class, LoginGroup.class})
    private String loginId;

    @Size(max=20, groups={RegisterGroup.class}))
    @NotBlank(groups={RegisterGroup.class, LoginGroup.class})
    private String loginPass;

    @NotBlank(groups={RegisterGroup.class})
    private String name;
    …
```

文字数チェックは、会員登録時に行う

未入力チェックは、会員登録時とログイン時の両方にを行う

バリデーショングループの使い方

- コントローラーでは、@Validの代わりに@Validatedアノテーションを使ってグループを指定する

例：会員登録のURLに対応するメソッドでは、RegisterGroupを指定

```
@PostMapping("/add")
public String addUser(@Validated(RegisterGroup.class) User user,
                      Errors errors) {
  …
```

例：ログインのURLに対応するメソッドでは、LoginGroupを指定

```
@PostMapping("/login")
public String login(@Validated(LoginGroup.class) User user,
                    Errors errors) {
  …
```

練習

- 練習09-4
- 練習09-5