

■ Boot-Camp学習プログラム

Boot-Camp学習プログラム 社員管理システム(re_emp_manager)について

Boot-Camp学習プログラム 社員管理システム(re_emp_manager)は、JavaEEの技術を使用してMVCデザインパターンで作成されたWEBアプリケーションです。

【M】 Model（モデル）の部分には、JavaEEのCDIという技術を、
【V】 View（ビュー）の部分には、Thymeleafというテンプレートエンジンを、
【C】 Controller（コントローラ）の部分には、JavaEEのJAX-RSという技術を使用してMVCの実装を行っています。

※ モジュールの構成は【モジュール構成図】シートを参照してください。

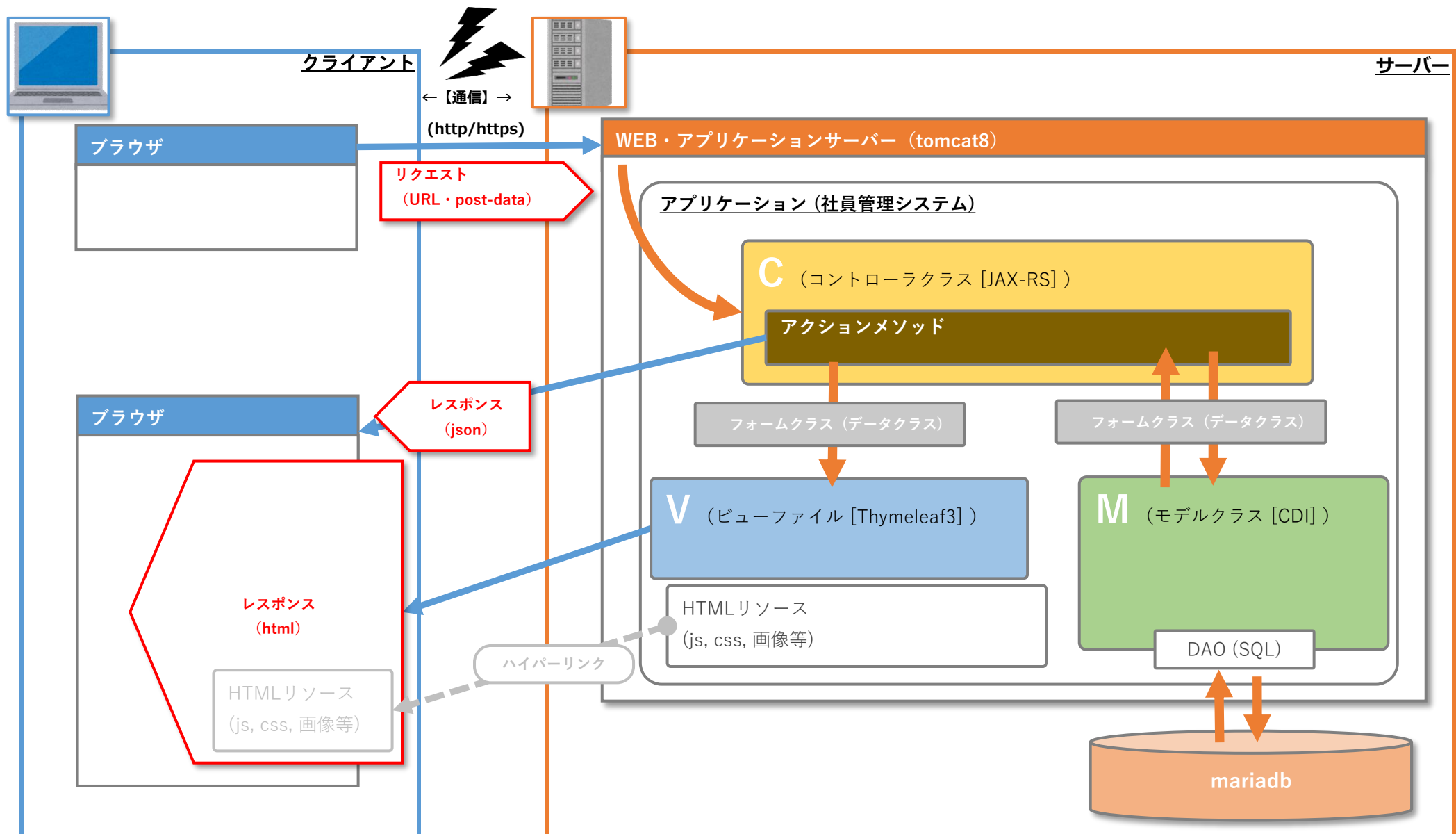
※ アプリケーションの構成は【アプリケーション構成図】シートを参照してください。

学習プログラムはJAVAE7や使用ライブラリの標準になるべく沿うような形で作成していますが、一部に独自の拡張を行っています。

プログラムを調査する際は、ライブラリの標準なのか独自仕様なのかをよく考えたうえで、調査してください。
※ 独自仕様については、この資料が公開しているJAVADOCをご確認ください。

Boot-Camp学習プログラム

アプリケーション構成図



主な用語

ブラウザ

ホームページを見るためのアプリケーション。 （詳しくはWEBで！）

WEBサーバー

HTTPに則り、クライアントソフトウェアのウェブブラウザに対して、HTMLやオブジェクト（画像など）の表示を提供するサービスプログラム。 （詳しくはWEBで！）

アプリケーションサーバー

ビジネスロジックなどを実装したアプリケーションソフトウェアの実行を管理補助するミドルウェア。 （詳しくはWEBで！）

コントローラクラス

画面のリクエスト・レスポンス制御処理を記述します。 （詳しくは【学習プログラム】のシートで！）

フォームクラス

画面のデータを格納するクラス。 （詳しくは【学習プログラム】のシートで！）

モデルクラス

画面の業務処理を行うクラス。 （詳しくは【学習プログラム】のシートで！）

ビューファイル（HTML）

画面のレイアウトの設定を行うファイル。

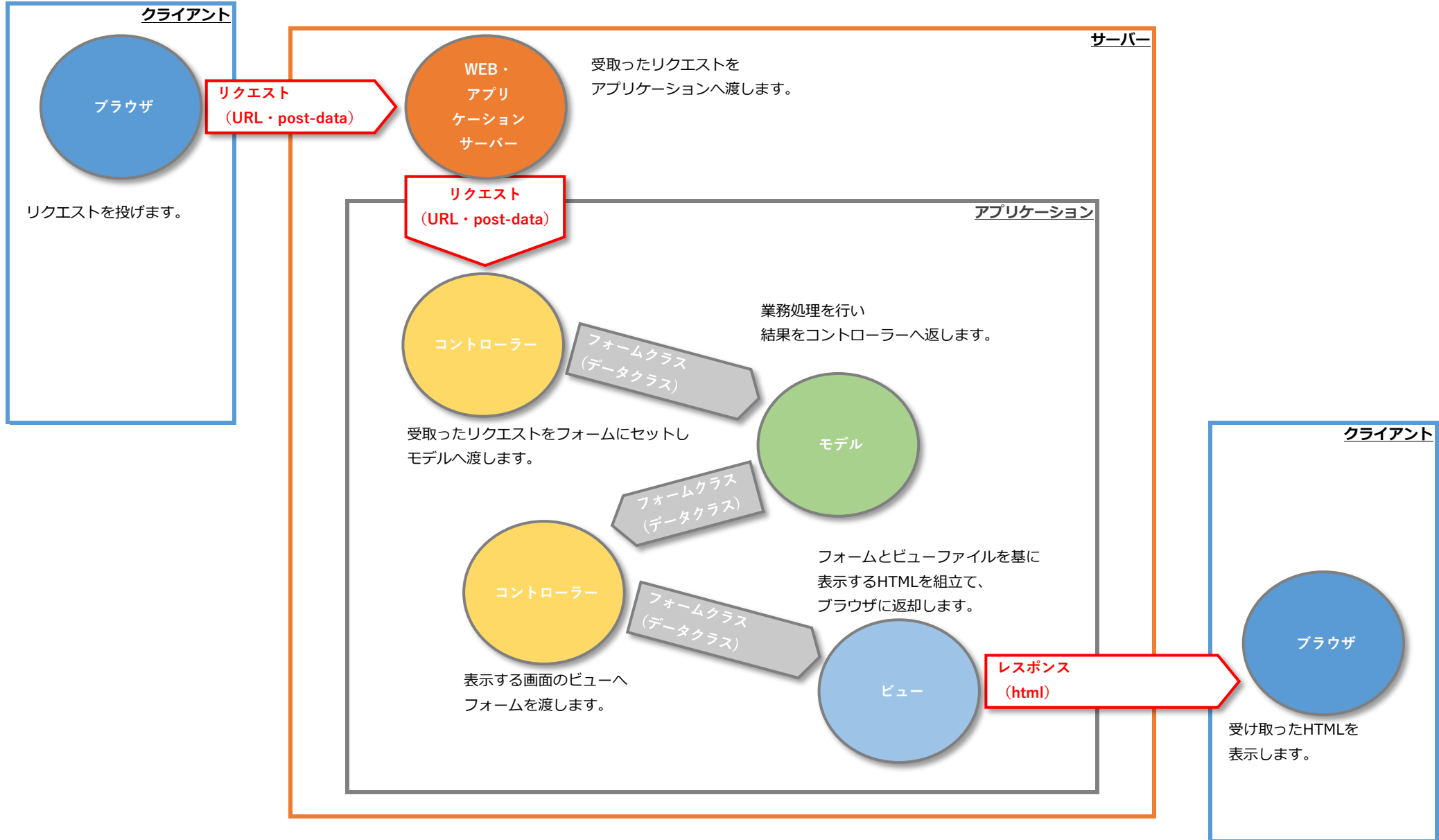
スクリプトファイル

ブラウザ側で処理する内容を記述します。

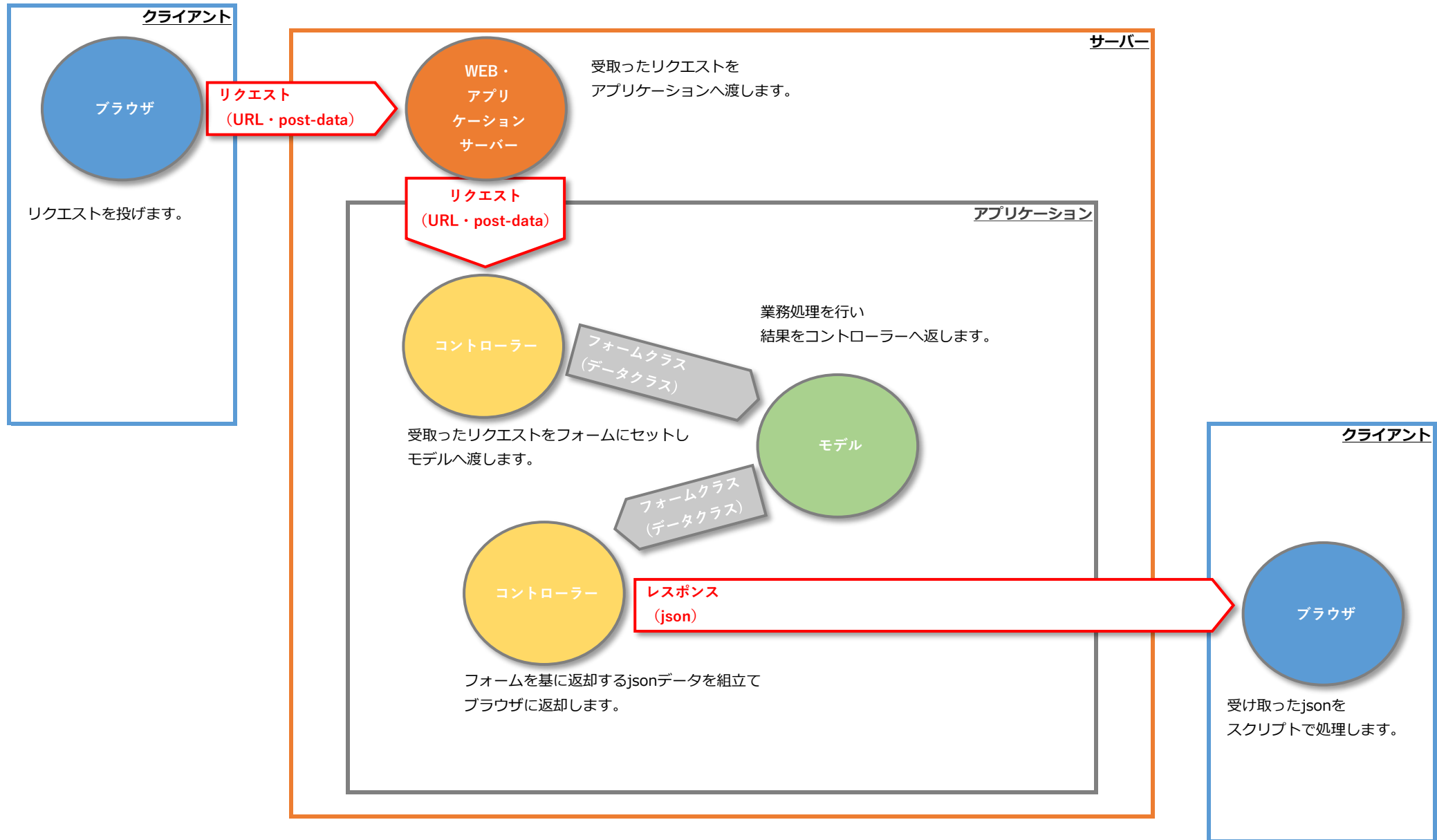
スタイルシートファイル

画面固有のスタイル設定を記述します。

画面処理(htmlレスポンス)の流れ



非同期処理(json)の流れ



Boot-Camp学習プログラム

Boot-Camp学習プログラム プログラムについて

各モジュールは以下のように作成しています。

1.コントローラクラス

画面のリクエスト・レスポンス制御処理を記述します。（業務ロジックは記述しません）

例) LoginController.java

@Path("/login")	…リクエストURLの設定 (JAX-RS)。
@Controller	…@Controller属性設定 (独自実装)。
@RequestScoped	…スコープの設定(JAX-RS)。

```
public class LoginController {  
  
    private static final String TEMPLATE = "login/index";  
  
    /** メイン業務処理. */  
    @Inject  
    private LoginModel model;  
  
    /** デフォルト. */  
    @GET  
    public HtmlResponse defaultAction() throws Exception {  
        return index();  
    }  
  
    /** 画面表示. */  
    @Path("/index")  
    @GET  
    public HtmlResponse index() throws Exception {  
        LoginForm form = model.init(FormUtil.make(LoginForm.class));  
        if (model.isLogin(form)) {  
            // 認証あり.  
            return new HtmlResponse().Redirect(form.getNext());  
        } else {  
            // 認証なし.  
            return new HtmlResponse(TEMPLATE).add("form", form);  
        }  
    }  
}
```

…業務処理(モデルクラス)をInjectする(CDI)。 (インスタンスを取得します。)
…アクションメソッド (JAX-RS)。 受け取る方式 (POST or GET) (JAX-RS)。 ※HtmlResponse (独自実装) 独自で拡張したレスポンス。 テンプレートを指定し、 テンプレートエンジンを使用して HTMLのレスポンスを返却します。 (リダイレクトでも使用します)
…アクションメソッド (JAX-RS)。
…リダイレクトアクション。
…画面表示アクション。

```

/** 次画面遷移 アクション. */
@Path("/next")
@POST
public HtmlResponse doNext(MultivaluedMap<String, String> params) throws Exception {
    LoginForm form = FormUtil.make(LoginForm.class, params);

    return new HtmlResponse().Redirect(form.getNext());
}
...省略
/** ログイン. */
@Path("/login")
@POST
@Produces(MediaType.APPLICATION_JSON)
public AjaxResponse doLogin(MultivaluedMap<String, String> params) throws Exception {
    LoginForm form = FormUtil.make(LoginForm.class, params);
    if (model.checkInput(form) && model.doLogin(form)) {
        // 認証OK.
        return new AjaxResponse(AjaxResponse.OK).setForm(form)
            .setMessageList(model.getMessageList());
    } else {
        // 認証NG.
        return new AjaxResponse(AjaxResponse.NG).setForm(form)
            .setMessageList(model.getMessageList());
    }
}

```

…アクションメソッド (JAX-RS)。

…リクエストデータの受け取り (JAX-RS)。

…リクエストデータを
フォームクラスに格納する (独自実装)。

…リクエストURLの設定 (JAX-RS)。

…Jsonでレスポンスする設定 (JAX-RS)。

…Jsonでレスポンスするアクション。

※AjaxResponse (独自実装)

- ・ 継承を必要としない。(POJO)
- ・ コントローラのアクセスパスを必ず指定する「@Path("/")」。

例)

コントローラパス : @Path("/login")

アクションパス : @Path("/index") の場合、

「コンテキストパス+"/"+コントローラパス+"/"+アクションパス」
なので、

「http://localhost:8080/re_emp_manager/login/index」で
アクセスが可能です。

- ・ スコープを設定する。(コントローラーは基本RequestScoped)
- ・ 業務ロジックは書かない。
(画面データを受け取り、モデルクラスを呼出し、業務処理を行います)

2. フォームクラス

画面のデータを格納するクラス。（業務ロジックは記述しません）

例) LoginForm.java

```
public class LoginForm {

    //----- properties [private].
    /** グループ識別コード. */
    @Param                                     ...リクエストデータを受取る際に設定 (独自実装)。
    private String code;
    /** ログインID. */
    @Param                                     ...リクエストデータを受取る際に設定 (独自実装)。
    private String id;
    /** パスワード. */
    @Param                                     ...リクエストデータを受取る際に設定 (独自実装)。
    private String key;
    ...省略

    //-- setter / getter. --//
    /** グループ識別コード を取得します. */
    public String getCode() { return code; }
    /** グループ識別コード を設定します. */
    public void setCode(String code) { this.code = code; }
    /** ログインID を取得します. */
    public String getId() { return id; }
    /** ログインID を設定します. */
    public void setId(String id) { this.id = id; }
    /** パスワード を取得します. */
    public String getKey() { return key; }
    /** パスワード を設定します. */
    public void setKey(String key) { this.key = key; }
    ...省略
}
```

- ・ 継承を必要としない。(POJO)
- ・ データ項目とセッター・ゲッターで構成される。
- ・ リクエストデータを受取る際は「@Param」アノテーションを設定する。
※オブジェクトで取得する際は「@DataParam」を
リストで受け取る際は「@ListParam」を指定します。
リストでデータを受け取る際はリストの件数も必要です。(リスト名+"_size")

3.モデルクラス

画面の業務処理を行うクラス。

例) LoginModel.java

@RequestScoped

…スコープの設定(JAX-RS)。

```
public class LoginModel implements IMessage {  
    …省略
```

```
/** メッセージ モデルクラス. */
```

…業務処理(モデルクラス)をInjectする(CDI)。
(インスタンスを取得します。)

@Inject

```
private MessageModel msgModel;
```

```
/** 共通処理 モデルクラス. */
```

@Inject

```
private CommonModel common;
```

```
    …省略
```

```
/** インジェクション完了後、コールバック. */
```

…Inject時のインスタンスの生成時と破棄時の
処理を指定する(CDI)。

@PostConstruct

```
public void setup() {  
    messageList = new ArrayList<>();  
}
```

```
/** インスタンスの破棄時、コールバック. */
```

@PreDestroy

```
public void destroy() {}  
    …省略
```

```
/**
```

…入力チェックの業務処理。

```
 * 入力チェックを行います。  
 * @param form 対象データ。  
 * @return 結果。  
 */
```

```
public boolean checkInput(final LoginForm form) {
```

```
    boolean ret = true;  
    String name = "";  
    String selector = "";  
    String value = "";  
    checker.setMessageList(messageList);
```

```
    //-----
```

```
    name = "グループ識別コード";  
    selector = "#code";  
    value = form.getCode();  
    if (!checker.checkCode(value, false, name, selector)) { ret = false; }  
    …省略
```

```

        return ret;
    }

    /**
     * ログイン処理を行います。
     * @param form 対象データ。
     * @return 処理結果。
     * @throws Exception
     */
    public boolean doLogin(final LoginForm form) throws Exception {
        try {
            dao.begin();

            ...省略

        } catch (SimpleDaoException e) {
            dao.rollback();
            throw e;
        }
    }

    ...省略
}

```

- ・ 継承を必要としない。(POJO)
- ・ スコープを設定する。
(@SessionScopedの場合は、Serializableの指定が必要。)
- ・ 引数無しのコンストラクタが必要。
(コンストラクタ自体無い方が良いかもしれません)
- ・ インスタンス生成時と破棄するときに処理が必要な場合は下記のアノテーションのメソッドを作成する。
 - @PostConstruct ... 生成時。
 - @PostConstruct ... 破棄する時。
- ・ データアクセス処理はこのクラスで行います。

4.ビューファイル (HTML)

画面のレイアウトの設定を行うファイル。

- ・ テンプレートエンジンThymeleaf3の規約に沿ったHTMLファイル。
参考) <http://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html>

5.スクリプトファイル

ブラウザ側で処理する内容を記述します。

6.スタイルシートファイル

画面固有のスタイル設定を記述します。

命名規約

Java

クラス (例 : TopModel)	パスカルケース
インタフェース (例 : IEntity)	I + パスカルケース
パッケージ (例 : com.res_system.re_employee_manager.model.top)	小文字のスネークケース
メソッド (例 : getId())	キャメルケース
ローカル変数	キャメルケース
メンバ変数	キャメルケース
定数 (例 : PROPERTY_FILE_NAME)	大文字のスネークケース
列挙定数	パスカルケース

※但しEntityクラス等で、DB項目名と紐づく場合はDB項目名をそのまま使用する。

JavaScript

クラス (例 : TopModel)	パスカルケース
メソッド (例 : getId())	キャメルケース
ローカル変数	キャメルケース
メンバ変数	キャメルケース
定数	大文字のスネークケース

※但しEntityクラス等で、DB項目名と紐づく場合はDB項目名をそのまま使用する。

HTML

id	小文字のスネークケース
----	-------------

※但しEntityクラス等で、DB項目名と紐づく場合はDB項目名をそのまま使用する。

CSS

クラス (例 : menu-box)	小文字のチェインケース
-----------------------	-------------

※但しEntityクラス等で、DB項目名と紐づく場合はDB項目名をそのまま使用する。

既存のライブラリのものはそのまま使用する。

Boot-Camp学習プログラム

Boot-Camp学習プログラム Thymeleaf拡張について

Boot-Camp学習プログラムでは、Thymeleaf3よりいくつかの機能を追加しています。
詳細については以下のパッケージのjavadocを参照してください。

(参照パッケージ)

```
com.res_system.commons.mvc.view.thexpressionobjects  
com.res_system.commons.mvc.view.thprocessors  
com.res_system.re_emp_manager.commons.view.thexpressionobjects
```

主なThymeleaf拡張にプロパティ 及びヘルパー関数。

ビューファイルで使用する、主なThymeleaf拡張にプロパティ 及びヘルパー関数は以下の通りです。

拡張プロパティ

re-th:input

[InputProcessor]

inputタグの属性(id,name,value)の設定を行います。
対象のタグに以下のプロパティを設定します。

使用例)

```
[ re-th:input = "~,~"    (フォーム名,フィールド名)]  
又は  
[ re-th:input = "~,~,~"  (フォーム名,データ名,フィールド名)]  
又は  
[ re-th:input = "~,~,~,~" (フォーム名,リスト名,リストIndex,フィールド名)]
```

re-th:checkbox

[CheckboxProcessor]

input[checkbox]タグの属性(id,name,value)の設定を行います。
対象のタグに以下のプロパティを設定します。

使用例)

```
[ re-th:checkbox = "~,~,~"    (フォーム名,フィールド名,フィールド値)]  
又は  
[ re-th:checkbox = "~,~,~,~"  (フォーム名,データ名,フィールド名,フィールド値)]  
又は  
[ re-th:checkbox = "~,~,~,~,~" (フォーム名,リスト名,リストIndex,フィールド名,フィールド値)]
```

re-th:radio

[RadioProcessor]

input[radio]タグの属性(id,name,value)の設定を行います。
対象のタグに以下のプロパティを設定します。

使用例)

```
[ re-th:radio = "~,~,~" (フォーム名,フィールド名,フィールド値)]  
又は  
[ re-th:radio = "~,~,~,~" (フォーム名,データ名,フィールド名,フィールド値)]  
又は  
[ re-th:radio = "~,~,~,~,~" (フォーム名,リスト名,リストIndex,フィールド名,フィールド値)]
```

re-th:select

[SelectProcessor]

selectタグの属性(id,name)の設定を行います。
optionについては「re-th:option」で対応する。
対象のタグに以下のプロパティを設定します。

使用例)

```
[ re-th:select = "~,~" (フォーム名,フィールド名)]  
又は  
[ re-th:select = "~,~,~" (フォーム名,データ名,フィールド名)]  
又は  
[ re-th:select = "~,~,~,~" (フォーム名,リスト名,リストIndex,フィールド名)]
```

re-th:option

[SelectOptionProcessor]

select[option]タグの属性(value,selected)の設定を行います。
対象のタグに以下のプロパティを設定します。

使用例)

```
[ re-th:option = "~,~,~,~"  
    (フォーム名,フィールド名,フィールド値,フィールド表示文字)]  
又は  
[ re-th:option = "~,~,~,~,~"  
    (フォーム名,データ名,フィールド名,フィールド値,フィールド表示文字)]  
又は  
[ re-th:option = "~,~,~,~,~,~"  
    (フォーム名,リスト名,リストIndex,フィールド名,フィールド値,フィールド表示文字)]
```

re-th:textarea

[TextareaProcessor]

textareaタグの属性(id,name)の設定を行います。
対象のタグに以下のプロパティを設定します。

使用例)

```
[ re-th:textarea = "~,~" (フォーム名,フィールド名)]  
又は
```

[re-th:textarea = "~,~,~" (フォーム名,データ名,フィールド名)]

又は

[re-h:textarea = "~,~,~,~" (フォーム名,リスト名,リストIndex,フィールド名)]

re-th:cltext

[CTextAttrProcessor]

改行を
タグに変換してテキストを表示します.
対象のタグに以下のプロパティを設定します。

使用例)

[re-th:cltext="~" (対象テキスト)]

re-th:listSize

[ListSizeProcessor]

入力リストのサイズを保存するhiddenタグを設定します.
対象のタグに以下のプロパティを設定します。

使用例)

[re-th:listSize = "~,~" (フォーム名,リスト名)]

re-th:uqhref

[UqHrefProcessor]

ユニークなhref属性の設定を行います.
対象のタグに以下のプロパティを設定します。

使用例)

[re-th:uqhref = "~" (ファイルパス)]

re-th:uqsrc

[UqSrcProcessor]

ユニークなsrc属性の設定を行います.
対象のタグに以下のプロパティを設定します。

使用例)

[re-th:uqsrc = "~" (ファイルパス)]

ヘルパー関数

[ReThHelper]

#h.name()

name属性を作成します。
Thymeleafに設定する値に以下のように設定します。

使用例)

```
[ th:name="${#h.name('~','~')}" (フォーム名,フィールド名)]  
又は  
[ th:name="${#h.name('~','~','~')}" (フォーム名,データ名,フィールド名)]  
又は  
[ th:name="${#h.name('~','~','~','~')}" (フォーム名,リスト名,リストIndex,フィールド名)]
```

#h.id()

id属性を作成します。
Thymeleafに設定する値に以下のように設定します。

使用例)

```
[ th:id="${#h.id('~','~')}" (フォーム名,フィールド名)]  
又は  
[ th:id="${#h.id('~','~','~')}" (フォーム名,データ名,フィールド名)]  
又は  
[ th:id="${#h.id('~','~','~','~')}" (フォーム名,リスト名,リストIndex,フィールド名)]
```

#h.idWithValue()

id属性を作成します(値付き)。
Thymeleafに設定する値に以下のように設定します。

使用例)

```
[ th:id="${#h.idWithValue('~','~','~')}"  
                                (フォーム名,フィールド名,フィールド値)]  
又は  
[ th:id="${#h.idWithValue('~','~','~','~')}"  
                                (フォーム名,データ名,フィールド名,フィールド値)]  
又は  
[ th:id="${#h.idWithValue('~','~','~','~','~')}"  
                                (フォーム名,リスト名,リストIndex,フィールド名,フィールド値)]
```

#h.list()

リストの値を取得します。(List<String>)
Thymeleafに設定する値に以下のように設定します。

使用例)

```
[ th:text="${#h.list(~,~)}" (リスト,リストIndex)]
```

#h.listValue()

リストの値を取得します。(List<IListItem>)

Thymeleafに設定する値に以下のように設定します。

使用例)

```
[ th:text="${#h.listValue(~,~)}" (リスト,リストIndex)]
```

#h.listText()

リストアイテムより対象の値の表示文字列を取得します。(List<ListItem>)

Thymeleafに設定する値に以下のように設定します。

使用例)

```
[ th:text="${#h.listText(~,~)}" (リスト,対象値)]
```

#h.isExists()

対象の値がリスト内に存在する事を確認します。(List<String>)

Thymeleafに設定する値に以下のように設定します。

使用例)

```
[ th:checked="${#h.isExists(~, ~)}" (リスト,対象値)]
```

#h.isEmpty()

対象の値がNULL又は空である事を確認します。(String,List,Map)

Thymeleafに設定する値に以下のように設定します。

使用例)

```
[ th:if="${#h.isEmpty(~)}" (対象値)]
```

#h.isNotEmpty()

対象の値がNULL又は空で無い事を確認します。(String,List,Map)

Thymeleafに設定する値に以下のように設定します。

使用例)

```
[ th:if="${#h.isNotEmpty(~)}" (対象値)]
```

#h.json()

JSON文字列に変換します。

Thymeleafに設定する値に以下のように設定します。

使用例)

```
[ th:text="${#h.json(~)}" (対象オブジェクト)]
```

[RmThHelper]

#rmh.toAge()

年齢を算出します。

Thymeleafに設定する値に以下のように設定します。

使用例)

```
[ th:text="${ #rmh.toAge(~)}" (生年月日)]
```

#rmh.showPostalCode()

郵便番号を表示します。

Thymeleafに設定する値に以下のように設定します。

使用例)

```
[ th:text="${ #rmh.showPostalCode(~)}" (郵便番号)]
```

Boot-Camp学習プログラム

Boot-Camp学習プログラム データアクセスオブジェクト(DAO)について

Boot-Camp学習プログラムでは、データアクセス処理にSimpleDaoを使用します。

com.res_system.commons.dao.SimpleDao

SimpleDaoはJDBCを使用して、データアクセスを行うDAOクラスです。

またSimpleDaoはSqlMakerというSQL自動作成クラスを持ち、データクラス(エンティティクラス)からSQLを自動で作成し実行することが可能です。(自動作成したSQLは内部にキャッシュされます。)

com.res_system.commons.dao.sqlmaker.SqlMaker

SqlMakerは、エンティティクラスよりSQLを自動で作成するクラスです。

SqlMakerでSQLを自動作成する際、エンティティクラスには、IEntityインターフェース実装と以下のアノテーションの設定が必要です。

@Table ... クラスのアノテーションです。
SQL自動作成時のテーブル名を設定します。SELECT時の別名も設定できます。
指定がない場合は、クラス名をテーブル名とします。

@Column ... フィールドのアノテーションです。
SQL上のカラム名の指定や、各SQL作成時の対象項目か否かの設定を行います。
(name)カラム名を設定します。指定がない場合は、フィールド名をカラム名とします。
(isSelected)Select対象の項目にtrue、そうでない場合はfalseを指定します。(デフォルトはtrue)
(isInsert)Insert対象の項目にtrue、そうでない場合はfalseを指定します。(デフォルトはtrue)
(isUpdate)Update対象の項目にtrue、そうでない場合はfalseを指定します。(デフォルトはtrue)

@AtSelect ... Select時に直接値や式を設定する場合に指定します。

@AtInsert ... Insert時に直接値や式を設定する場合に指定します。

@AtUpdate ... Update時に直接値や式を設定する場合に指定します。

@Key ... フィールドのアノテーションです。
キー項目のフィールドに設定します。

@Sql ... クラスのアノテーションです。
SQLを名前付きで保存します。
SQLを記述する際に取得項目やFROM句を省略できます。
「--SELECT」 Entityクラスで自動作成した取得項目に置き換わります。
「--FROM」 Entityクラスで自動作成したFROM句に置き換わります。
「--WHERE_WT」 Entityクラスで自動作成したWHERE句
(テーブル名付キー項目)に置き換わります。
「--INSERT」 Entityクラスで自動作成したInsert文(冒頭)に置き換わります。
「--UPDATE」 Entityクラスで自動作成したUpdate文(冒頭)に置き換わります。
「--DELETE」 Entityクラスで自動作成したDelete文(冒頭)に置き換わります。
「--WHERE」 Entityクラスで自動作成したWHERE句(キー項目)に置き換わります。
パラメータをフィールド名で指定するとEntityクラスの内容でパラメータを自動設定します。
「:フィールド名(例 :cd)」 "?"に置換され、SQL実行時にパラメータを同名の
フィールドの値を設定します。
(通常はパラメータの埋め込み部分は"?"です)

@Sqls

… クラスのアノテーションです。

@SQLを複数設定する場合に使用します。

```
例) @Sqls({
    @Sql(name="find_list_test_1",sql="--SELECT --FROM --WHERE_WT")
    ,@Sql(name="find_list_test_2",sql="--SELECT --FROM WHERE code LIKE 'code%' ")
    ,@Sql(name="find_list_test_3",sql="--SELECT --FROM WHERE code LIKE 'code%' ")
})
```

データアクセス処理記述例

【 SQLを直接記述して実行する 】

■ 1レコードを取得する

(パラメータなし)

```
String sql = "SELECT * FROM `test_dao_table` `tdt`";
TestEntityDaoNonEntity actual = simpleDao.executeQuery(TestEntityDaoNonEntity.class, sql);
```

(パラメータあり)

```
String sql = "SELECT * FROM `test_dao_table` `tdt` WHERE `tdt`.`id` = ?";
TestEntityDaoNonColumn actual = simpleDao.executeQuery(TestEntityDaoNonColumn.class
    , sql
    , (st) -> { st.setLong(1, 999990001L); });
```

- ・ `executeQuery()`にレコードイメージのデータクラス(エンティティクラス)とSQL文をパラメータに設定して実行します。
- ・ データが取得できた場合、戻り値にデータクラスのインスタンスが返却されます。取得できない場合、NULLが返却されます。
- ・ 取得したデータは、カラム名に一致するフィールドへ格納されます。
- ・ SQLにパラメータがある場合は、SQLのあとにパラメータ値の設定処理を追加します。
※ラムダ式の引数「st」にはPreparedStatementが設定されますので、そのクラスを使用してパラメータの値を設定します。

■ 複数レコードを取得する

(パラメータなし)

```
String sql = "SELECT * FROM `test_dao_table` `tdt`";
List<TestEntityDaoNonEntity> actual
    = simpleDao.executeQueryList(TestEntityDaoNonEntity.class, sql);
```

(パラメータあり)

```
List<TestEntityDaoNonColumn> actual
```

```
= simpleDao.executeQueryList(TestEntityDaoNonColumn.class  
    , sql  
    , (st) -> { st.setString(1, "00%"); });
```

- `executeQueryList()`にレコードイメージのデータクラス(エンティティクラス)とSQL文をパラメータに設定して実行します。
- データが取得できた場合、戻り値に取得できた件数分のデータクラスのリストが返却されます。取得できない場合、0件のリストが返却されます。
- 取得したデータは、カラム名に一致するフィールドへ格納されます。
- SQLにパラメータがある場合は、SQLのあとにパラメータ値の設定処理を追加します。
※ラムダ式の引数「st」にはPreparedStatementが設定されますので、そのクラスを使用してパラメータの値を設定します。

■ レコードを更新する

(パラメータなし)

```
String sql = "UPDATE test_dao_table SET memo = 'テスト' WHERE id = 1";  
int actual = simpleDao.executeUpdate(sql);
```

(パラメータあり)

```
String sql = "UPDATE test_dao_table SET memo = ? WHERE id = ?";  
int actual = simpleDao.executeUpdate(sql,  
    (st) -> {  
        st.setString(1, "テスト");  
        st.setLong(2, 1L);});
```

- `executeUpdate()`にSQL文を設定して実行します。戻り値に処理件数が返却されます。
- SQLにパラメータがある場合は、SQLのあとにパラメータ値の設定処理を追加します。
※ラムダ式の引数「st」にはPreparedStatementが設定されますので、そのクラスを使用してパラメータの値を設定します。

【 SQLを自動作成して実行する 】

■ 1レコードを取得する

```
TestEntityDao entity = new TestEntityDao();  
entity.setId("999999");  
TestEntityDao actual = simpleDao.find(entity);
```

- `find()`にレコードイメージのエンティティクラスのインスタンスをパラメータに設定して実行します。
- データが取得できた場合、戻り値にデータクラスのインスタンスが返却されます。取得できない場合、NULLが返却されます。

- ・ 取得したデータは、カラム名に一致するフィールドへ格納されます。

■ キーで1レコードを取得する

```
TestEntityDao actual = simpleDao.findByKey(TestEntityDao.class, "999999");
```

- ・ `findByKey()`にレコードイメージのエンティティクラスとキー値をパラメータに設定して実行します。
- ・ データが取得できた場合、戻り値にデータクラスのインスタンスが返却されます。
取得できない場合、NULLが返却されます。
- ・ 取得したデータは、カラム名に一致するフィールドへ格納されます。

■ 複数レコードを取得する

```
TestEntityDao entity = new TestEntityDao();  
TestEntityDao actual = simpleDao.findList(entity);
```

- ・ `findList()`にレコードイメージのエンティティクラスのインスタンスをパラメータに設定して実行します。
- ・ データが取得できた場合、戻り値に取得できた件数分のデータクラスのリストが返却されます。
取得できない場合、0件のリストが返却されます。
- ・ 取得したデータは、カラム名に一致するフィールドへ格納されます。

■ レコードを追加する

```
TestEntityDao entity = new TestEntityDao();  
entity.setId("999999");  
entity.setCd("code999999");  
entity.setNm("name999999");  
int actual = simpleDao.insert(entity);
```

- ・ `insert()`にレコードイメージのエンティティクラスのインスタンスをパラメータに設定して実行します。
戻り値に処理件数が返却されます。

■ レコードを更新する

```
TestEntityDao entity = new TestEntityDao();  
entity.setId("999999");  
entity.setCd("code999999");  
entity.setNm("nameXXXXXX");  
int actual = simpleDao.update(entity);
```

- ・ `update()`にレコードイメージのエンティティクラスのインスタンスをパラメータに設定して実行します。

戻り値に処理件数が返却されます。

■ レコードを削除する

```
TestEntityDao entity = new TestEntityDao();
entity.setId("999999");
int actual = simpleDao.delete(entity);
```

- delete()にレコードイメージのエンティティクラスのインスタンスをパラメータに設定して実行します。
戻り値に処理件数が返却されます。

【 エンティティクラスに設定されたSQLを使用して実行する 】

■ 1レコードを取得する

(パラメータなし)

```
TestEntityDao
...
@Sqls({
    @Sql(name="find_1",sql="--SELECT --FROM")
})
public class TestEntityDao implements IEntity {
...
    TestEntityDao actual = simpleDao.find(TestEntityDao.class, "find_1");
```

(パラメータあり)

```
TestEntityDao2
...
@Sqls({
    @Sql(name="find_1",sql="--SELECT --FROM WHERE code LIKE ? ")
})
public class TestEntityDao2 implements IEntity {
...
    TestEntityDao2 actual = simpleDao.find(TestEntityDao2.class
        , "find_1"
        , (st) -> { st.setString(1, "00%"); });
```

(パラメータあり(フィールド名指定))

```
TestEntityDao3
...
@Sqls({
    @Sql(name="find_1",sql="--SELECT --FROM WHERE code LIKE :cd "),
})
```

```
public class TestEntityDao3 implements IEntity {
...
    TestEntityDao3 entity = new TestEntityDao3();
    entity.setCode("01%");
    TestEntityDao3 actual = simpleDao.find(entity, "find_1");
}
```

- ・ find()にレコードイメージのデータクラス(エンティティクラス)又はエンティティクラスのインスタンスとSQL名をパラメータに設定して実行します。
- ・ データが取得できた場合、戻り値にデータクラスのインスタンスが返却されます。取得できない場合、NULLが返却されます。
- ・ 取得したデータは、カラム名に一致するフィールドへ格納されます。
- ・ SQLにパラメータがある場合は、SQLのあとにパラメータ値の設定処理を追加します。
※ラムダ式の引数「st」にはPreparedStatementが設定されますので、そのクラスを使用してパラメータの値を設定します。
- ・ SQLのパラメータがフィールド名の場合、エンティティクラスのインスタンスより指定されたフィールドの値をパラメータに設定します。(第一引数がエンティティクラスのインスタンスの場合のみ)

■ 複数レコードを取得する

(パラメータなし)

```
TestEntityDao
...
@Sqls({
    @Sql(name="find_1",sql="--SELECT --FROM")
})
public class TestEntityDao implements IEntity {
...
    List<TestEntityDao> actual = simpleDao.findList(TestEntityDao.class, "find_1");
}
```

(パラメータあり)

```
TestEntityDao2
...
@Sqls({
    @Sql(name="find_1",sql="--SELECT --FROM WHERE code LIKE ? ")
})
public class TestEntityDao2 implements IEntity {
...
    List<TestEntityDao> actual = simpleDao.findList(TestEntityDao2.class
        , "find_1"
        , (st) -> { st.setString(1, "00%"); });
}
```

(パラメータあり(フィールド名指定))

```
TestEntityDao3
...
@Sqls({
```



```

    @Sql(name="find_1",sql="--SELECT --FROM WHERE code LIKE :cd "),
    })
public class TestEntityDao3 implements IEntity {
    ...
    TestEntityDao3 entity = new TestEntityDao3();
    entity.setCode("01%");
    List<TestEntityDao> actual = simpleDao.findList(entity, "find_1");
}

```

- ・ findList()にレコードイメージのデータクラス(エンティティクラス)又はエンティティクラスのインスタンスとSQL名をパラメータに設定して実行します。
- ・ データが取得できた場合、戻り値に取得できた件数分のデータクラスのリストが返却されます。取得できない場合、0件のリストが返却されます。
- ・ 取得したデータは、カラム名に一致するフィールドへ格納されます。
- ・ SQLにパラメータがある場合は、SQLのあとにパラメータ値の設定処理を追加します。
※ラムダ式の引数「st」にはPreparedStatementが設定されますので、そのクラスを使用してパラメータの値を設定します。
- ・ SQLのパラメータがフィールド名の場合、エンティティクラスのインスタンスより指定されたフィールドの値をパラメータに設定します。(第一引数がエンティティクラスのインスタンスの場合のみ)

■ レコードを追加する

(パラメータなし)

```

TestEntityDao2
...
@Sqls({
    @Sql(name="insert_test_1"
        ,sql="INSERT INTO test_dao_table (id,code,name) "
        +"VALUES ('9999990001','code0001','name0001')")
    })
public class TestEntityDao2 implements IEntity {
    ...
    int actual = simpleDao.insert(TestEntityDao2.class, "insert_test_1");
}

```

(パラメータあり)

```

TestEntityDao2
...
@Sqls({
    @Sql(name="insert_test_2"
        ,sql="INSERT INTO test_dao_table (id,code,name) VALUES (?, ?, ?)")
    })
public class TestEntityDao2 implements IEntity {
    ...
    TestEntityDao2 entity = new TestEntityDao2()
        .setId("9999990002").setCd("code0002").setNm("name0002");
    int actual = simpleDao.insert(TestEntityDao2.class, "insert_test_2"
        , (st) -> { simpleDao.setStatementParam(st, 1, entity, "id", "cd", "nm"); });
}

```

(パラメータあり(フィールド名指定))

```
TestEntityDao2
...
@Sqls({
    @Sql(name="insert_test_3"
        ,sql="INSERT INTO test_dao_table (id,code,name) VALUES (:id,:cd,:nm)")
    })
public class TestEntityDao2 implements IEntity {
...
    TestEntityDao2 entity = new TestEntityDao2()
        .setId("999990003").setCd("code0003").setNm("name0003");
    int actual = simpleDao.insert(entity, "insert_test_3");
}
```

- ・ insert()にレコードイメージのデータクラス(エンティティクラス)又はエンティティクラスのインスタンスとSQL名をパラメータに設定して実行します。
戻り値に処理件数が返却されます。
- ・ SQLにパラメータがある場合は、SQLのあとにパラメータ値の設定処理を追加します。
※ラムダ式の引数「st」にはPreparedStatementが設定されますので、
そのクラスを使用してパラメータの値を設定します。
- ・ SQLのパラメータがフィールド名の場合、エンティティクラスのインスタンスより指定されたフィールドの値をパラメータに設定します。(第一引数がエンティティクラスのインスタンスの場合のみ)

■ レコードを更新する

(パラメータなし)

```
TestEntityDao2
...
@Sqls({
    @Sql(name="update_test_1"
        ,sql="UPDATE test_dao_table SET code = 'code0001x', name = 'name0001x' "
        +"WHERE id = '999990001'")
    })
public class TestEntityDao2 implements IEntity {
...
    int actual = simpleDao.update(TestEntityDao2.class, "update_test_1");
}
```

(パラメータあり)

```
TestEntityDao2
...
@Sqls({
    @Sql(name="update_test_2"
        ,sql="UPDATE test_dao_table SET code = ?, name = ? WHERE id = ? ")
    })
public class TestEntityDao2 implements IEntity {
}
```

```

...
TestEntityDao2 entity = new TestEntityDao2()
    .setId("999990002").setCd("code0002x").setNm("name0002x");
int actual = simpleDao.update(TestEntityDao2.class, "update_test_2"
    , (st) -> { simpleDao.setStatementParam(st, 1, entity, "id", "cd", "nm"); });

```

(パラメータあり(フィールド名指定))

```

TestEntityDao2
...
@Sqls({
    @Sql(name="update_test_3"
        ,sql="UPDATE test_dao_table SET code = :cd, name = :nm WHERE id = :id")
})
public class TestEntityDao2 implements IEntity {
...
    TestEntityDao2 entity = new TestEntityDao2()
        .setId("999990003").setCd("code0003").setNm("name0003");
    int actual = simpleDao.update(entity, "update_test_3");

```

- update()にレコードイメージのデータクラス(エンティティクラス)又はエンティティクラスのインスタンスとSQL名をパラメータに設定して実行します。
戻り値に処理件数が返却されます。
- SQLにパラメータがある場合は、SQLのあとにパラメータ値の設定処理を追加します。
※ラムダ式の引数「st」にはPreparedStatementが設定されますので、
そのクラスを使用してパラメータの値を設定します。
- SQLのパラメータがフィールド名の場合、エンティティクラスのインスタンスより指定されたフィールドの値をパラメータに設定します。(第一引数がエンティティクラスのインスタンスの場合のみ)

■ レコードを削除する

(パラメータなし)

```

TestEntityDao2
...
@Sqls({
    @Sql(name="delete_test_1"
        ,sql="DELETE FROM test_dao_table WHERE id = '999990001'")
})
public class TestEntityDao2 implements IEntity {
...
    int actual = simpleDao.delete(TestEntityDao2.class, "delete_test_1");

```

(パラメータあり)

TestEntityDao2

```

...
@Sqls({

```

```

    @Sql(name="delete_test_2",sql="DELETE FROM test_dao_table WHERE id = ? ")
    })
public class TestEntityDao2 implements IEntity {
...
    TestEntityDao2 entity = new TestEntityDao2().setId("999990002");
    int actual = simpleDao.delete(TestEntityDao2.class, "delete_test_2"
        , (st) -> { simpleDao.setStatementParam(st, 1, entity, "id"); });
}

```

(パラメータあり(フィールド名指定))

```

TestEntityDao2
...
@Sqls({
    @Sql(name="delete_test_3",sql="DELETE FROM test_dao_table WHERE id = :id ")
    })
public class TestEntityDao2 implements IEntity {
...
    TestEntityDao2 entity = new TestEntityDao2().setId("999990003");
    int actual = simpleDao.delete(entity, "delete_test_3");
}

```

- delete() にレコードイメージのデータクラス(エンティティクラス)又はエンティティクラスのインスタンスと SQL 名をパラメータに設定して実行します。
戻り値に処理件数が返却されます。
- SQL にパラメータがある場合は、SQL のあとにパラメータ値の設定処理を追加します。
※ラムダ式の引数「st」には PreparedStatement が設定されますので、
そのクラスを使用してパラメータの値を設定します。
- SQL のパラメータがフィールド名の場合、エンティティクラスのインスタンスより指定されたフィールドの値をパラメータに設定します。(第一引数がエンティティクラスのインスタンスの場合のみ)

【トランザクションの制御】

トランザクションの制御は下記のように行います。

```

try {
    dao.begin();           ... トランザクション開始
    ...                   ... データアクセス処理。
    dao.commit();          ... コミット
} catch (SimpleDaoException e) {
    dao.rollback();        ... ロールバック
    throw e;
}

```

Boot-Camp学習プログラム

Boot-Camp学習プログラム デプロイについて

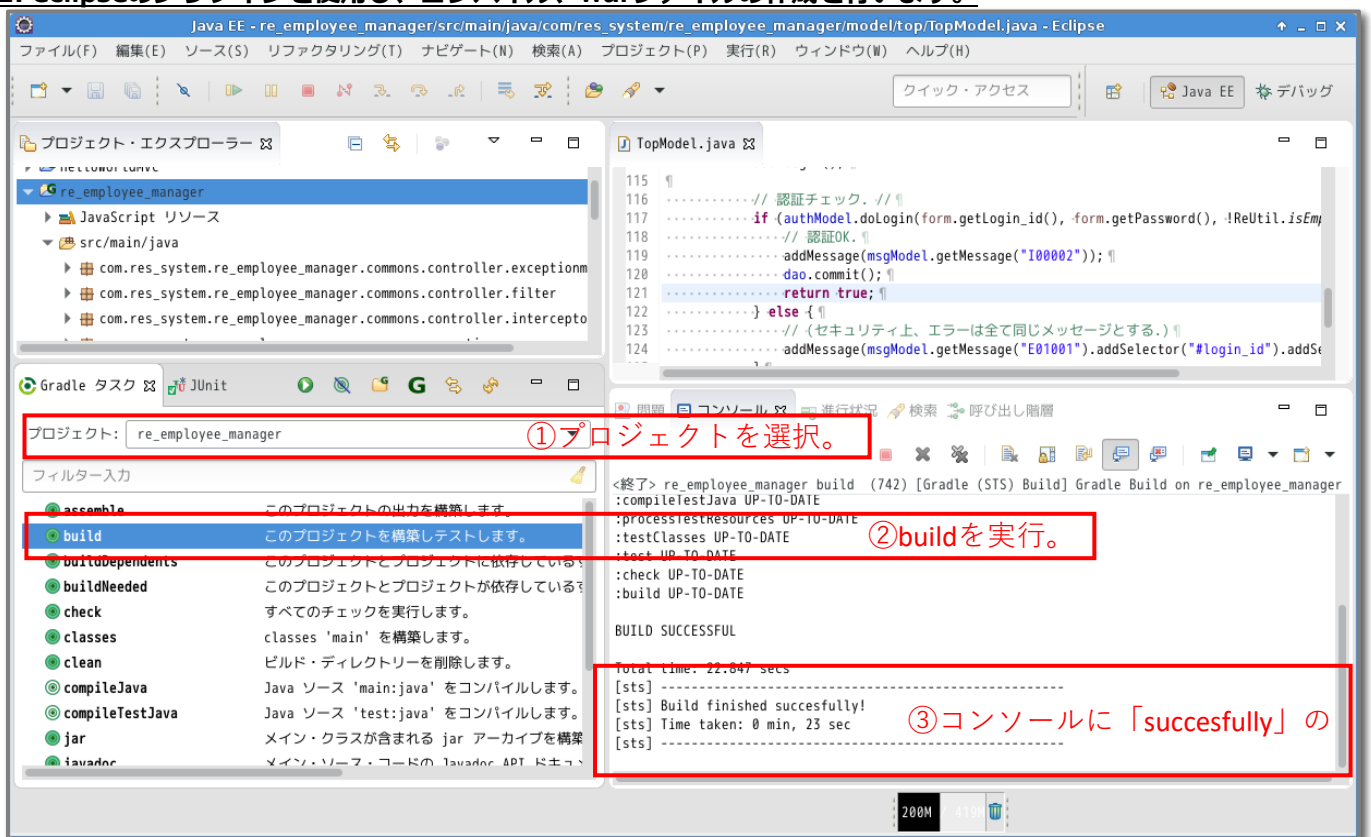
Boot-Camp学習プログラムの製造環境では、コンパイルツールにGradleを使用します。

eclipseよりGradleを使用し、コンパイル、warファイルの作成を行います。

作成されたwarファイルをtomcatのwebアプリケーションフォルダ「/opt/lampp/tomcat8/webapps」に設置し、デプロイをします。

デプロイ方法

1. eclipseのプラグインを使用し、コンパイル、warファイルの作成を行います。



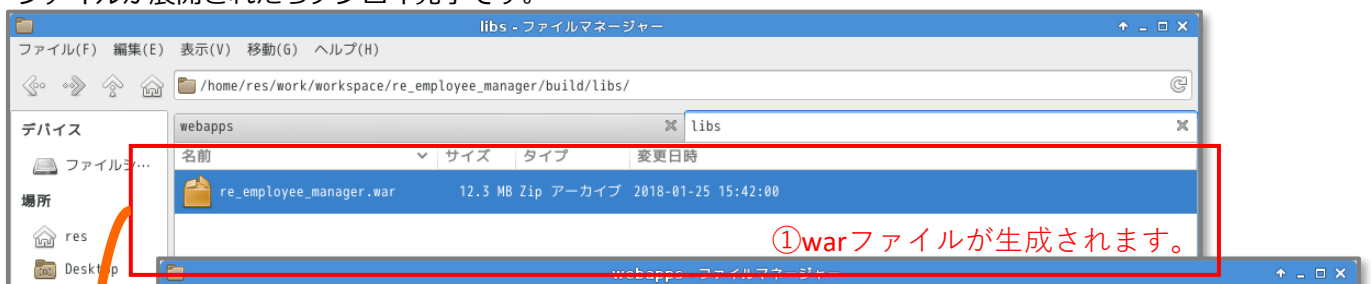
2. warの配置。

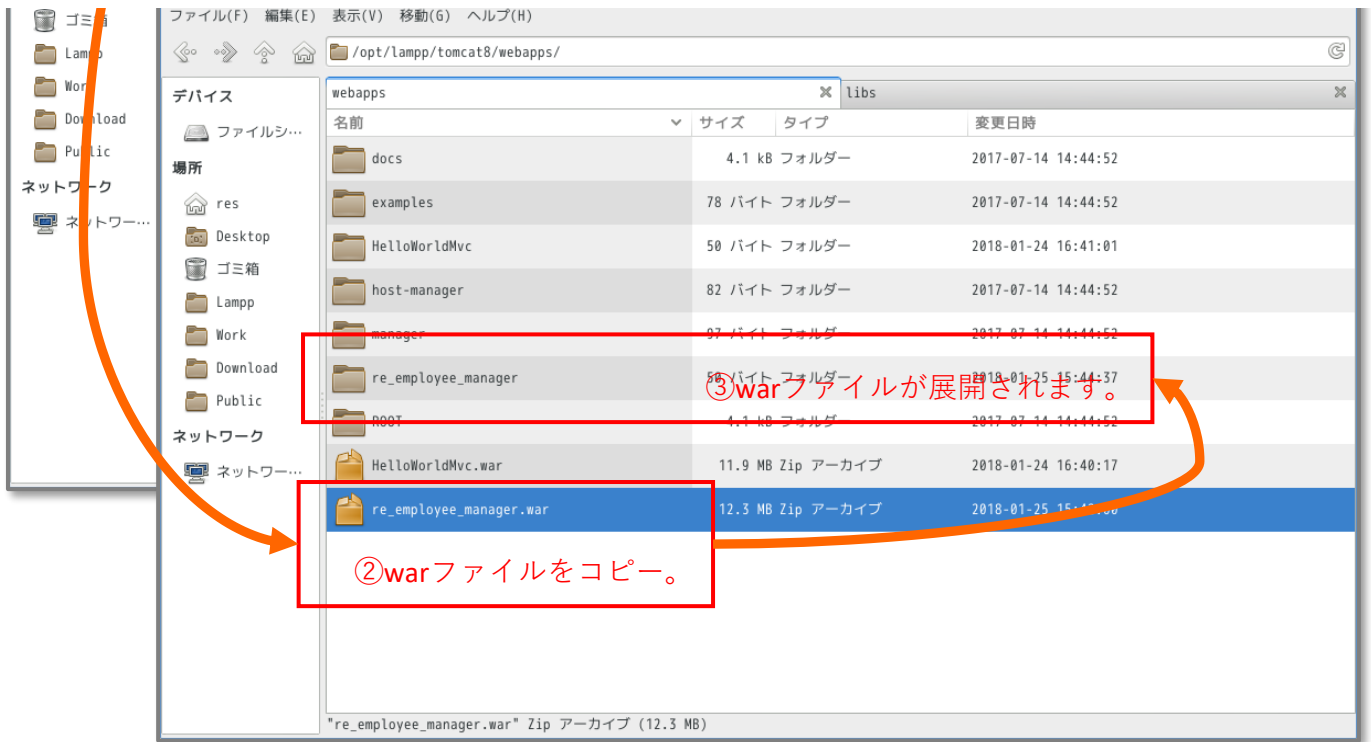
Gradleでbuildするとプロジェクトフォルダの「/build/libs/」にwarファイルが生成されます。

そのwarファイルをTomcatの「webapps」フォルダにコピーします。

tomcatが起動された状態ですと、実行可能な状態へwarファイルが展開されます。

ファイルが展開されたらデプロイ完了です。





Boot-Camp学習プログラム

Boot-Camp学習プログラム ログイン・データについて

ログインURL(開発環境):

http://localhost:8080/re_emp_manager/

テストデータのログインについて、以下のテストユーザーを用意しています。

システム権限

ユーザー名:	グループ識別コード:	ログインID:	パスワード:
システム		system@res-system.com	test1234

一般権限 (オーナー)

ユーザー名:	グループ識別コード:	ログインID:	パスワード:
RESオーナーユーザー		test@res-system.com	test1234
グループBオーナーユーザー		group-b@res-system.com	test1234
テストユーザー X		group-ab@res-system.com	test1234

グループマネージャー権限

ユーザー名:	グループ識別コード:	ログインID:	パスワード:
山田 太郎	res-system.com	test@res-system.com	test1234
鈴木 太郎	group-b.com	group-b@res-system.com	test1234

グループリーダー権限

ユーザー名:	グループ識別コード:	ログインID:	パスワード:
山田 次郎	res-system.com	member02	test1234
山田 四郎	res-system.com	member04	test1234
鈴木 次郎	group-b.com	member02	test1234

グループメンバー権限

ユーザー名:	グループ識別コード:	ログインID:	パスワード:
山田 三郎	res-system.com	member03	test1234
山田 五郎	res-system.com	member05	test1234
山田 六郎	res-system.com	group-ab@res-system.com	test1234
鈴木 三郎	group-b.com	member03	test1234
鈴木 四郎	group-b.com	member04	test1234
鈴木 五郎	group-b.com	member05	test1234
鈴木 六郎	group-b.com	group-ab@res-system.com	test1234

※ re_emp_manager/src/work/sql/のSQLファイルを実行すると、初期値に戻ります。

re_emp_manager_all.sql … テーブル定義

re_emp_manager_data.sql … 初期データ

※ 開発環境のデータベース情報はPhpMyAdminで確認できます。

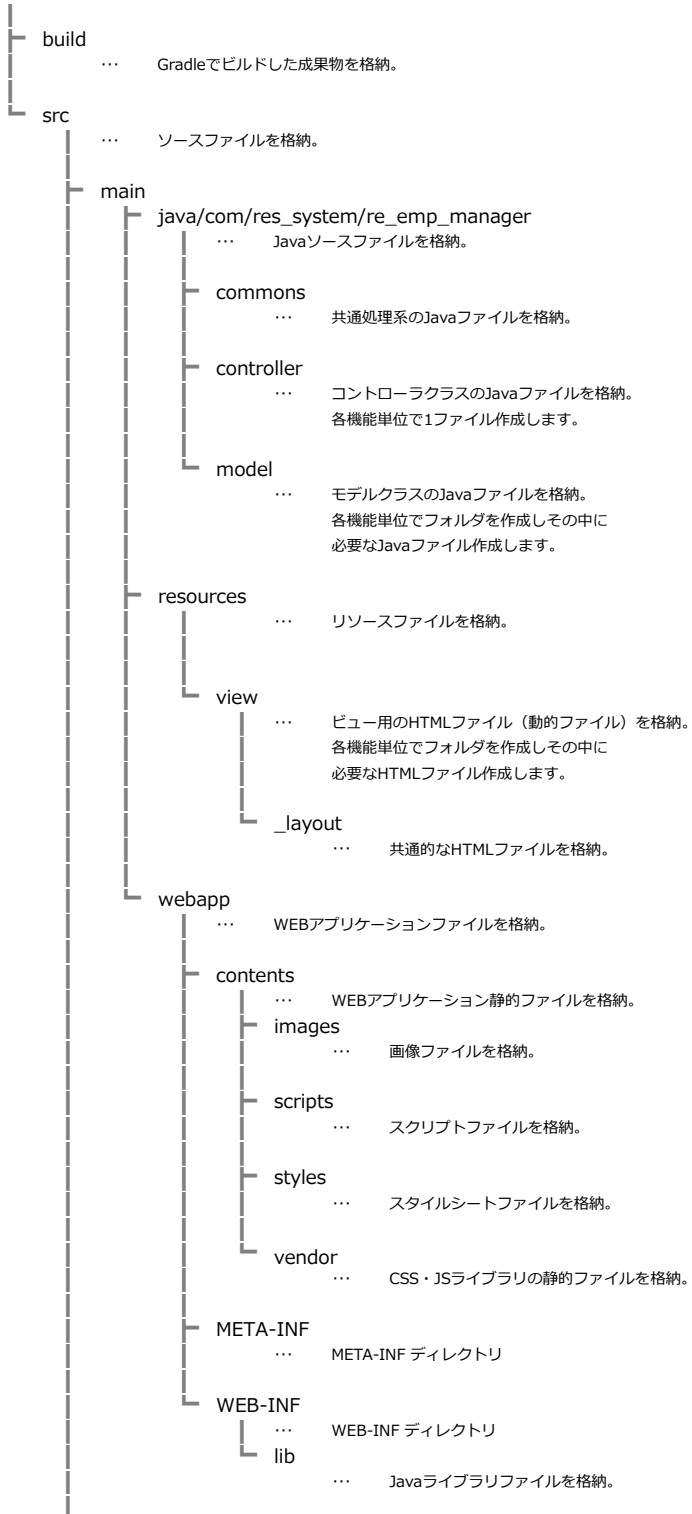
■ Boot-Camp学習プログラム

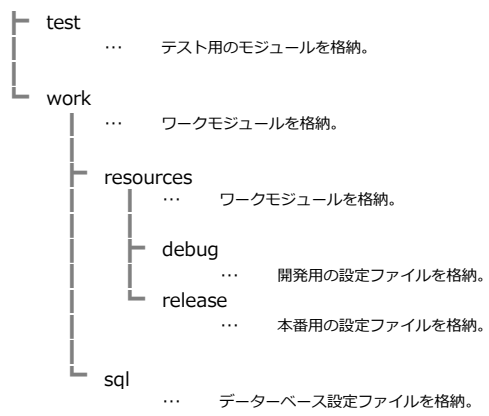
モジュール構成図

『社員管理システム』

(WEBアプリケーション ソース)

re_emp_manager

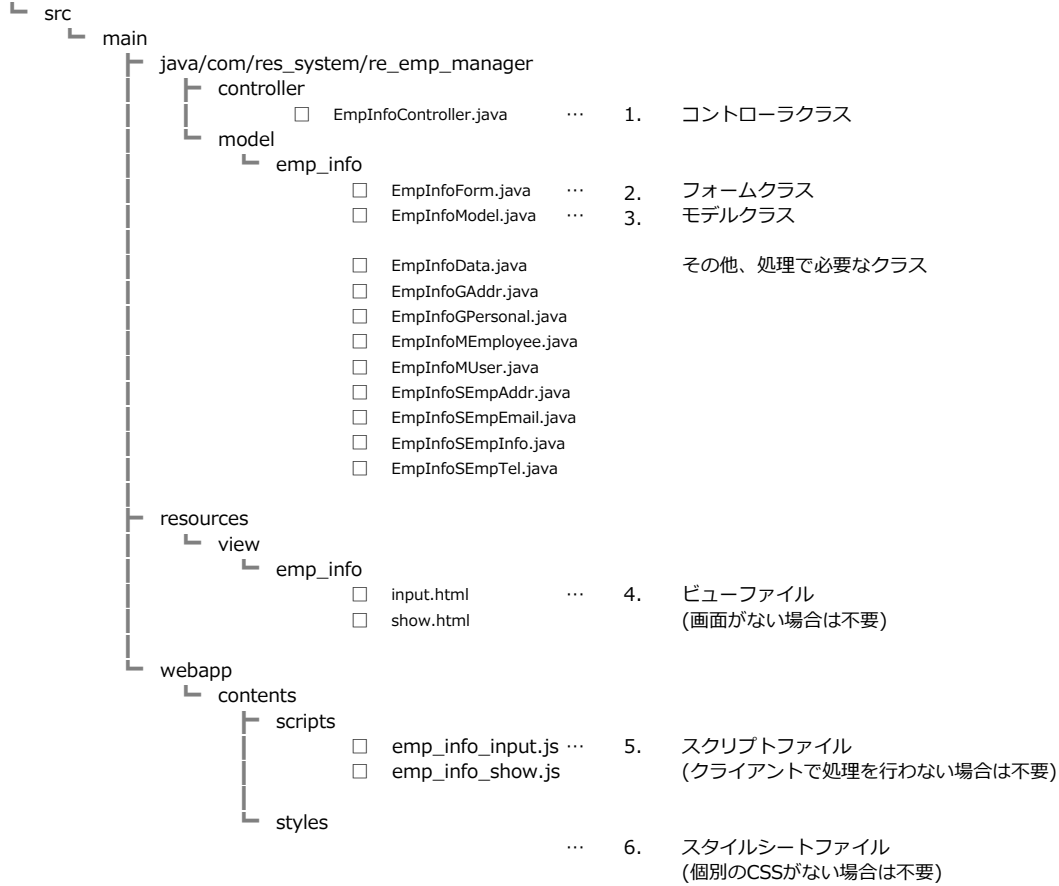




『1業務画面のモジュール構成』

例) 社員個人情報管理

re_emp_manager



Boot-Camp学習プログラム

学習環境

OS

CentOS7 (Xfce)

言語

Java8

IDE

eclipse Mars.2 Release (4.5.2)

WEB (アプリケーションサーバー)

tomcat8

コンパイルツール

Gradle

データベース

mariadb10.1. (MySQL互換)

フレームワーク

JAVAE7相当

- JAX-RS(実装 jersey2)
- CDI(実装 Weld3)
- Thymeleaf3 (テンプレートエンジン)

その他使用ライブラリ

- Bootstrap v4.0.0-beta.3 (CSSテンプレート)
- jQuery v3.2.1 (Javascriptライブラリ)
- JUnit4.12(モジュールテスト)