Seasar Conference
2008 Autumn

SAStrutsの開発Tips

出羽 健一

はじめに

- このセッションの内容
  - SAStrutsを使った開発において、悩みそうなトピックに絞って解説
  - SAStrutsと一緒に使用されることが多いS2JDBCについても扱う
  - SAStrutsの基本的な内容については扱わない
    - 公式ドキュメント(※1)や以前のカンファレンス資料(※2)をどうぞ！

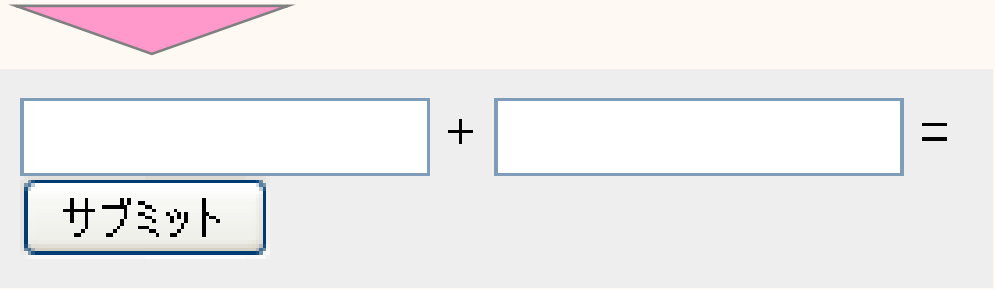
※1　SAStrutsの公式ページ
http://sastruts.seasar.org/
※2　StrutsからSAStrutsへ
http://event.seasarfoundation.org/sc2008spring/Session#s4

SAStrutsの特徴

- Strutsベースのフレームワーク
  - Strutsの利点を活かせる
    - ノウハウ、開発者人口の多さ、実行速度、安定性
  - Strutsのマイナス要素が排除されている
    - 設定ファイル地獄からの開放
- ホットデプロイ対応によるサクサク開発
  - ブラウザのリロードでソースコード修正が即反映される
- 脱CoC
  - CoCはマッピングなど必要最小限に限定
  - 明示的なアノテーションベースの開発
- エンタープライズはもちろん、
  Strutsが苦手なアジャイルもOK！

- 足し算(JSP)

```
<html:errors/>
<s:form>
  <html:text property="arg1"/> +
  <html:text property="arg2"/>
  = ${f:h(result)}<br />
  <input type="submit" name="submit"
   value="サブミット"/>
</s:form>
```



- 足し算（アクションフォーム）

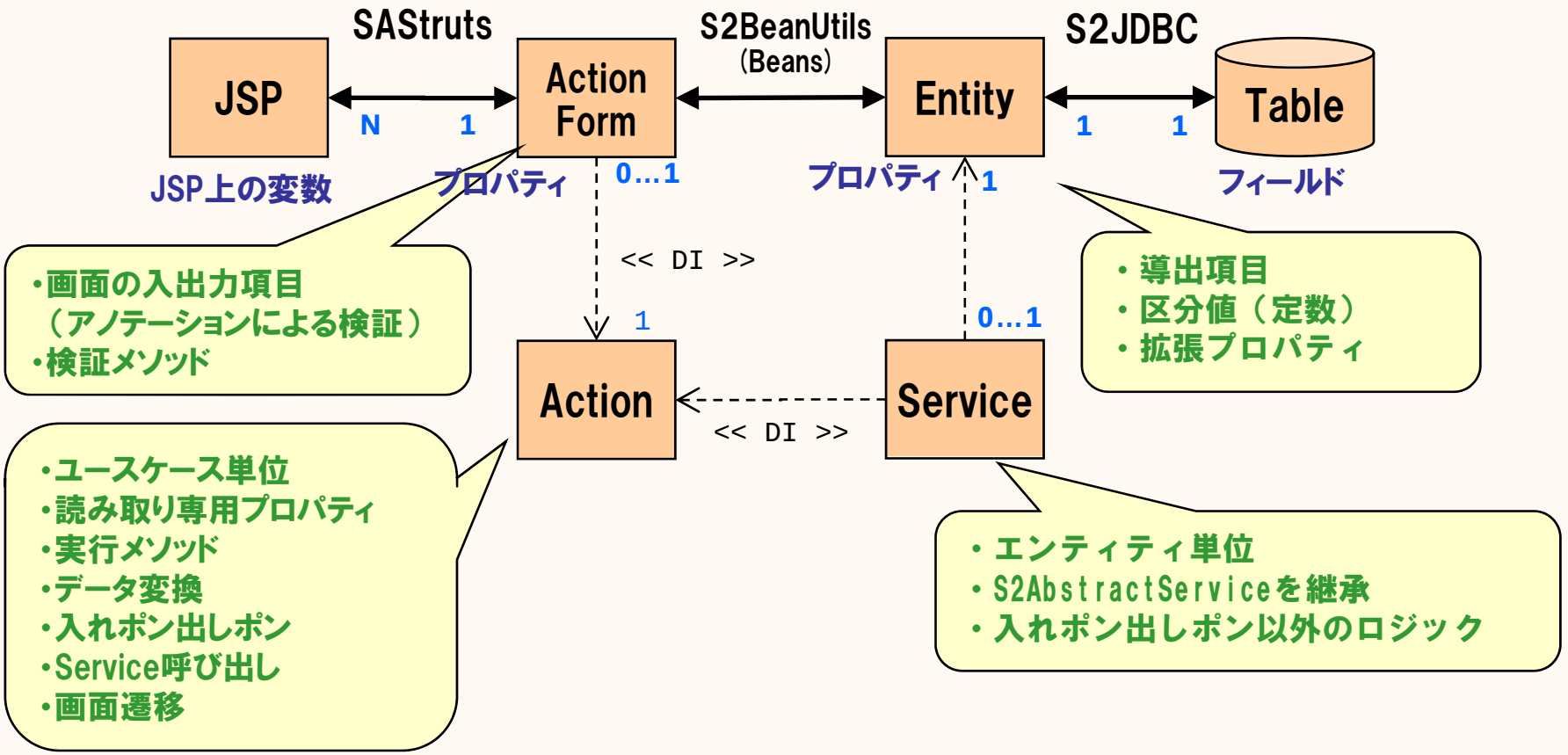
```
public class AddForm {

    @Required
    @IntegerType
    public String arg1;

    @Required
    @IntegerType
    public String arg2;
}
```

SAStrutsのおさらい

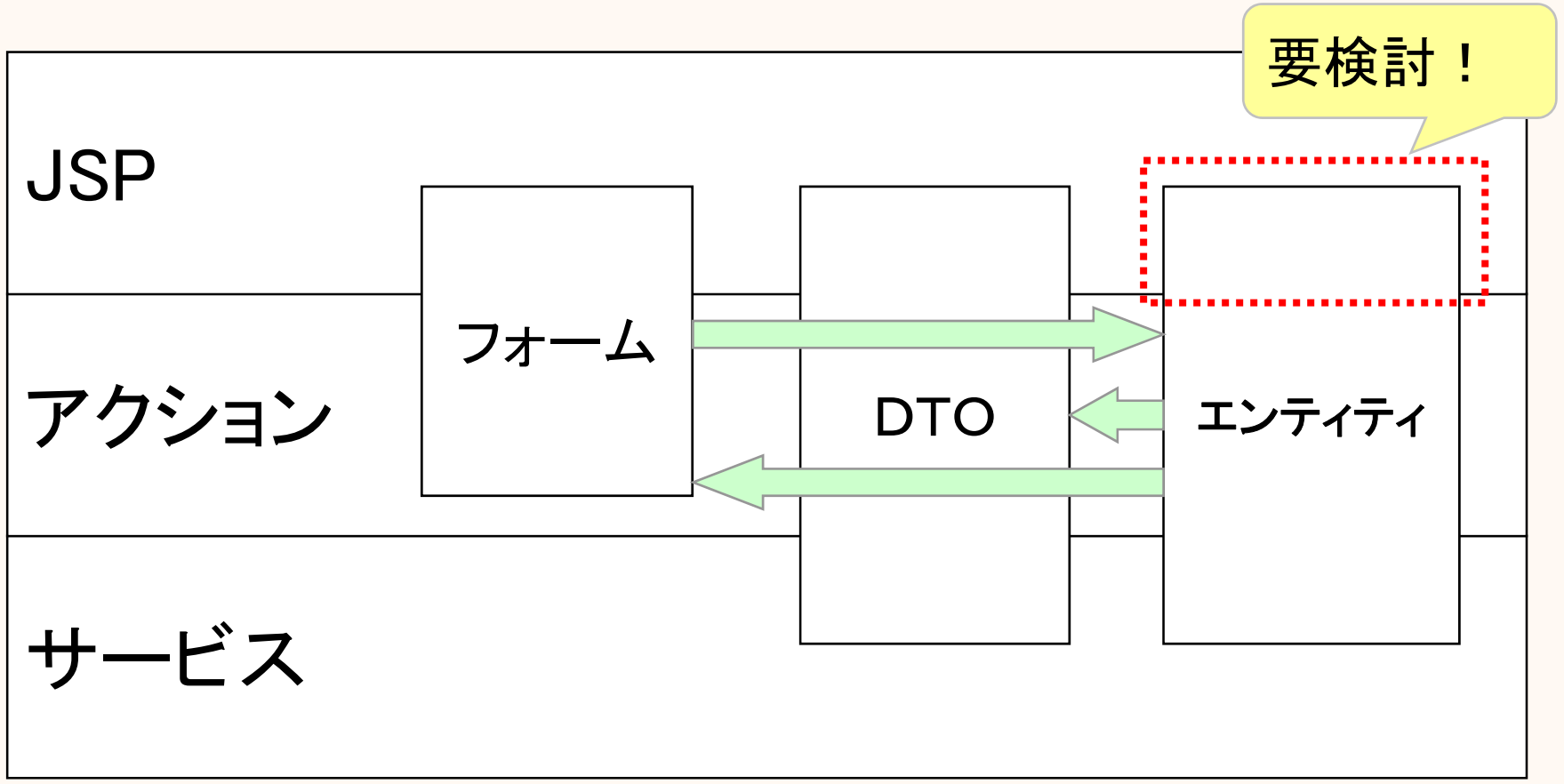
- 足し算（アクション）

```
public class AddAction {
    @ActionForm
    @Resource
    protected AddForm addForm;

    public Integer result;

    @Execute(validator = false)
    public String index() {
        return "index.jsp";
    }

    @Execute(input = "index.jsp")
    public String submit() {
        result = Integer.valueOf(addForm.arg1)
            + Integer.valueOf(addForm.arg2);
        return "index.jsp";
    }
}
```

アーキテクチャ



レイヤ – モデル図



※ モデルの詰め替え処理はアクションで行う ← 重要
※ JSPにエンティティを持ち込まないアーキテクチャも検討に値する

レイヤ – モデルのアンチパターン①

- サービスメソッドの引数にアクションフォームを渡す
  - 下位レイヤは上位レイヤのモジュールに依存すべきでない
  - 検索条件はアクションフォームからDtoに詰め替えたものをサービスへ渡そう

レイヤ – モデルのアンチパターン②

- アクションフォームからDtoに詰め替えたものをサービスに渡し、サービス内でDtoからエンティティに詰め替える
  - 画面入力値をDBに入れるだけであれば、INとOUTをテストすれば良いので、途中の無駄な詰め替え処理は減らしたい
  - アクション内でアクションフォームからエンティティに詰め替えて、それをサービスへ渡すようにする

アクションフォーム

- 責務
  - 画面の入出力項目
  - 入力値の検証（アノテーション、メソッド）

アクションフォーム

- ポイント
  - 入力チェック対象プロパティの型：
    String型、 String[]型、 boolean型
  - 型変換を保証するアノテーション：
    @IntegerType，@DateType など

※ プロパティをString型で持つと、
HTTPプロトコルとの相性が良くてハマりにくい

バリデータの全体像

JSP

エラーメッセージ出力

```
<html:errors/>
```

or

```
<html:errors property="xxx"/>
<html:errors property="yyy"/>
<html:errors property="zzz"/>
```

フォーム

```
<s:form>
  ...
</s:form>
```

アクションフォーム

検証用のアノテーション

・ターゲット指定
・独自作成可能

検証メソッド

アクション

実行メソッド

・エラー時の遷移先指定
・検証メソッドの指定
（複数指定可）
・エラー時の継続制御
(stopOnvalidationError)

applicaton_ja.properties

・ラベル指定（labels.xxx= … ）
・メッセージテンプレートの変更
・エラーメッセージ出力(<html:errors/>)の書式カスタマイズ

アクション

- 責務
  - リクエスト処理
    - 画面系データとDB系データの変換
    - 入れポン出しポン系データアクセスロジック
    - 入れポン出しポン以外の業務ロジック
    - 出力用プロパティ
    - 画面遷移

アクション:「実行メソッド」

- 実行メソッドは2種類に分けると良い
  - 入力系処理の実行メソッド
  - 出力系処理の実行メソッド

- 上記のように分けておくと、ボタン追加や画面遷移などの仕様変更時の修正コストが少なくなる

アクション:「実行メソッド」

- 入力系処理の実行メソッド
  - 呼ばれるタイミング: サブミットが呼ばれた時
  - 主処理: 画面入力項目をDBへ格納する
  - バリデーション: あり
  - 粒度: サブミットボタンの数だけ作る
  - 命名: doから始まる名前にしておくと分かりやすい
  - 戻り値: 出力系処理の実行メソッドを呼ぶ

アクション:「実行メソッド」

- 出力系処理の実行メソッド
  - 呼ばれるタイミング: 画面を表示する時
  - 主処理: DBのデータを画面に出力する
  - バリデーション: なし
  - 粒度: 1つのJSPに付き1メソッド
  - 命名: メソッド名はJSP名と同じ
  - 戻り値: JSPファイル

アクションフォームのDI

- 短いアクションフォームの名前で扱う

```
@Resource
@ActionForm
protected HogeFugaForm hogeFugaForm;
```

name要素で短い名前を指定

```
@Resource(name = “hogeFugaForm")
@ActionForm
protected HogeFugaForm form;
```

サービス

- S2JDBCを使う場合は、S2AbstractServiceの使用を推奨します
- 1つのエンティティに対し、1つのサービスを作成する
- S2AbstractServiceを使うとJdbcManagerの薄いラッパーとして扱える

Serviceのクラス図

FW開発者

<< abstract >>
S2AbstractService

アーキテクト

<< abstract >>
AbstractService

プログラマ

EmpService

DeptService

…Service

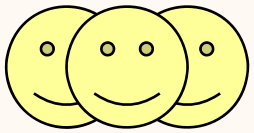
S2AbstractServiceの中身(抜粋)

```
public abstract S2AbstractService<T> {

    @Resource
    protected JdbcManager jdbcManager;

    protected Class<T> entityClass;


        ...

    public AutoSelect<T> select() {
        return jdbcManager.from(entityClass);
    }

    public int insert(T entity) {
        return jdbcManager.insert(entity).execute();
    }
        ...
}
```

アプリケーション側のサービス

アーキテクト

```
public abstract class AbstractService<ENTITY>
        extends S2AbstractService<ENTITY> {
}
```



プログラマ

```
public class EmpService extends AbstractService<Employee> {
}
```

※ この時点では、サービスの中身は空っぽ

S2AbstractServiceの利用イメージ

呼び出しイメージ:

```
Employee employee = employeeService.findById(5);

employee.name = “Dewa”;
employeeService.update(employee);
```

※ findByIdメソッドやupdateメソッドはS2AbstractServiceクラスで
定義されている

S2AbstractServiceの利用イメージ

- 「流れるようなインターフェース」もOK

```
@Resource
protected EmployeeService employeeService;

public List<Employee> employees;

@Execute(validate = false)
public String list() {
    employees = employeeService
            .select()
                .innerJoin(“department”)
            .where(new SimpleWhere()
                .ge(“age”, form.age_ge)
                .le(“age”, form.age_le))
            .orderBy(“sort”)
            .resultList();
    return “list.jsp”;
}
```

AbstractServiceにはどんなメソッドを定義する？

- 現時点では、S2AbstractServiceを継承したAbstractServiceや各サービスクラスは空っぽ

- では、何のために存在するのか？
  - どんなメソッドを定義するべきか？

AbstractService

- JdbcManagerの薄いラッパーとして機能する
  - 利用例： insertメソッドのオーバーライド
    - 以下のメソッドをAbstractServiceに定義しておくと、DB側で規定値をセットするケースに有効。

```
/**
 * エンティティを挿入します。
 * ただし、MODIFY_TIMESTAMP, VERSION のフィールドについては
 * エンティティの値は使用しません。
 */
@Override
public int insert(ENTITY entity) {
    return jdbcManager.insert(entity)
    .excludes("modifyTimestamp", "version")
    .execute();
}
```

※ updateメソッドにも有効

個別サービスクラスの実装

- 個別のサービスクラスにはどんなメソッドを実装すべきか？

⇒ 『入れポン出しポン』以外のデータアクセスロジック

データアクセスロジック

- データアクセスロジックの分類
  - 入れポン出しポン
    - 画面入力値をDBへ入れる
    - DBの値を画面項目として表示する
    - **サービスにはメソッドを定義しないが、サービスを使ってアクションから呼び出す**
  - 入れポン出しポン以外
    - 各種チェックロジック
      - 例：　ログイン認証チェック
    - **サービスにメソッド定義してアクションから呼び出す**

入れポン出しポン以外のデータアクセスロジックの例

- ログイン認証処理は、「入れポン出しポン」処理ではない

```
public class LoginAuthService extends AbstractService<LoginAuth> {
    public boolean checkLogin(String loginId, String password) {
        LoginAuth loginAuth =
            select()
            .where(new SimpleWhere()
                .eq("loginId", userId)
                .eq(“password", password))
            .getSingleResult();
        return loginAuth == null ? false : true;
    }
}
```

入れポン出しポン以外のデータアクセスロジックの例

```
/* ログインボタンが押された時に呼ばれる */
@Execute(input = “login")
public String doLogin() {
    boolean isLoginOK = loginAuthService
        .checkLogin (form.loginId, form.password);
    if (!isLoginOK) {
        // ログイン失敗した時の処理
        . . .
    }
    // ログイン成功した時の処理
    . . .
}
```

「入れポン出しポン」以外のデータアクセス処理はサービスクラスに自分で定義したメソッドを呼び出す

「入れポン出しポン」のまとめ

- Webアプリの多くが「入れポン出しポン」系
- 入れポン出しポンのデータアクセスロジックはサービスを活用して、アクションに記述する
- 入れポン出しポン以外のデータアクセスロジックは、サービスにメソッド定義したものをアクションから呼び出す

テスト

- アクションの実行メソッドに対して、INPUTとOUTPUTを意識したテストを実施
  - モックを使わないでDB経由するテスト
  - INPUT
    - アクションフォームのプロパティ
  - OUTPUT
    - アクションフォームのプロパティ
    - アクションのプロパティ
    - DB

テスト

- 入れポン出しポン以外のデータアクセスロジック
  - サービスクラスのpublicメソッドに対するテスト
    - モックを使用しないDB経由のテスト

データ詰め替え

- 主に「フォーム」と「エンティティ」の詰め替え
- 詰め替えはアクションにて行う
- 詰め替え処理はメソッド化しておくと、単体テストしやすい
  - スコープは protected
  - convert ではじまるメソッド名とかに統一しておくと分かりやすい
  - 例:
    - Employee employee = convertEmployee(xxxForm);
    - convert(xxxForm, employee);

エンティティ

- 共通の親クラス
- 区分値（定数）
- 拡張プロパティ
- （導出項目）

エンティティ:「共通の親クラス」

- 全てのエンティティに共通して保持する
  プロパティは親エンティティに持たせる

エンティティ:「共通の親クラス」

- 共通の親クラス(サンプル)

```
@MappedSuperclass
public abstract class AbstractEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer id;

    @Temporal(TemporalType.TIMESTAMP)
    public Date createTimestamp;

    @Temporal(TemporalType.TIMESTAMP)
    public Date modifyTimestamp;

    @Version
    public Integer version;
}
```

エンティティ:「共通の親クラス」

- 子クラス(サンプル)

```
@Entity
public class User extends AbstractEntity {

    public String name;

    public Integer height;

    public Integer weight;

    @Temporal(TemporalType.DATE)
    public Date birthday;
}
```

エンティティ:区分値(定数)

- DB上のフィールドの区分値は
  エンティティに定数として定義しておくと便利
  - ハードコーディングを避けれる
  - 汎用性の高い区分値は enum の使用も検討すべし!

エンティティ:区分値(定数)

- サンプルコード

```
public class User extends AbstractEntity {

    //=========================================
    //                                   定数
    //                                   ====
    public static final Integer STATUS_WORKING  = 1;
    public static final Integer STATUS_EATING   = 2;
    public static final Integer STATUS_SLEEPING = 3;

    //=========================================
    //                                   プロパティ
    //                                   =======
    public String name;

    public Integer status;
}
```

エンティティ:区分値(定数)

- サンプルコード
  - 寝ているユーザーの一覧を取得する

```
List<User> users = userService
    .select()
    .where(new SimpleWhere()
        .eq("status", User.STATUS_SLEEPING))
    .getResultList();
```

※ userService は下記のUserServiceクラスの
インスタンスであり、DIされているものとする

```
public class UserService extends AbstractService<User>{
}
```

エンティティ:「拡張プロパティ」

- エンティティにDBとは無関係のプロパティ（publicフィールド）を追加したい
  - しかし、不用意に追加すると、SQL文の自動生成時に追加した列が含まれてエラーになる

- 解決案:
  永続化対象外であることを指定する
  @Transient を使う

  ※「金額 = 単価 × 数量 」のような導出項目は
  読み取り専用プロパティとしてgetterで定義する

エンティティ:「拡張プロパティ」

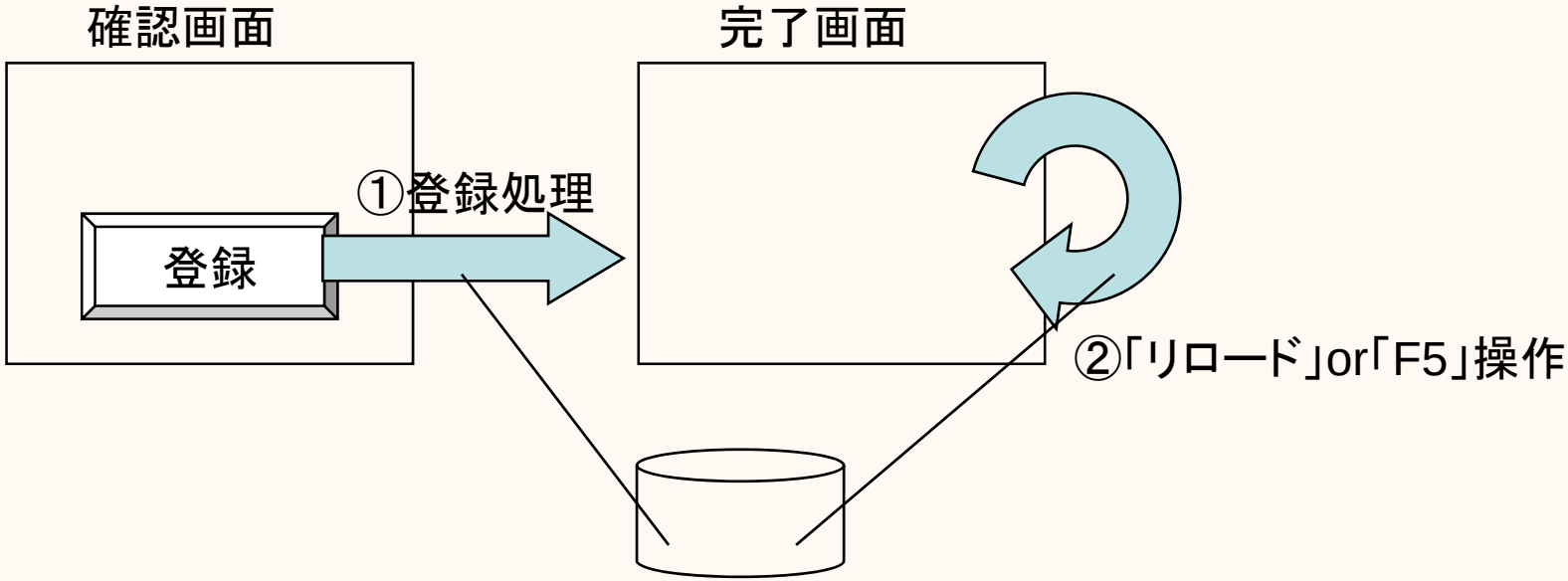
- サンプル

```
@Entity
public class User extends AbstractEntity {

    . . .

    //=========================================
    //                                拡張プロパティ
    //                                ==========
    /**
     * 回答済かどうか
     * @return true 回答済, false 未回答
     */
    @Transient
    public boolean isResponding;
}
```

@Transient が無いと
S2JDBCのSQL自動生成＆
実行時にエラーとなる。

登録後のF5による2重登録問題

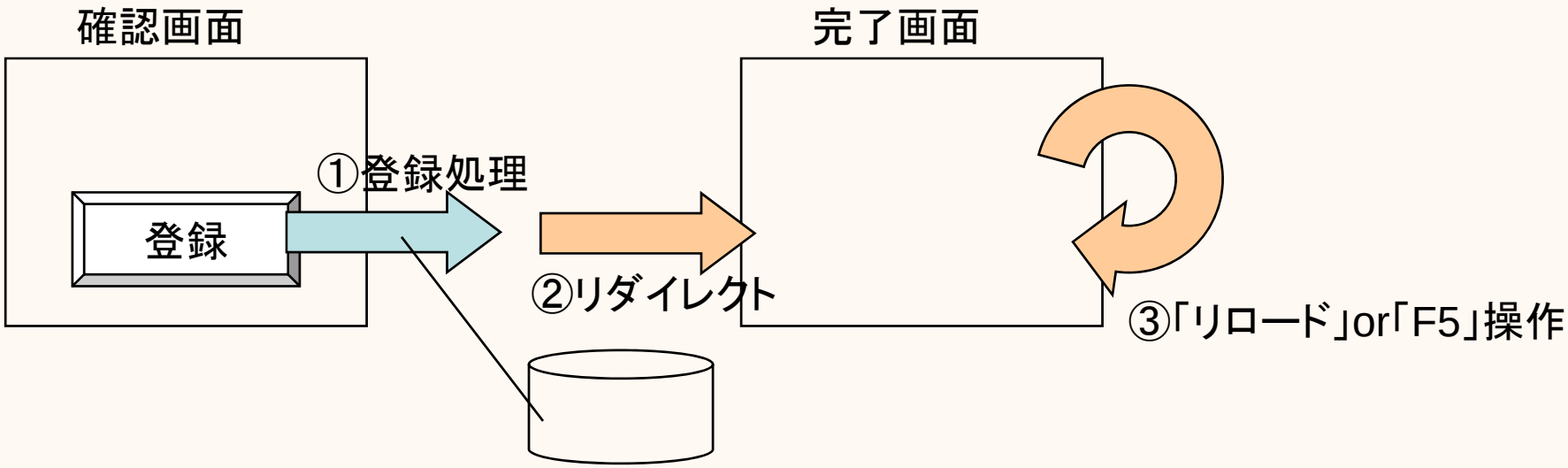
- 問題



リロード(F5)時に直前のリクエストが発行される
⇒ 二重登録されてしまう！

登録後のF5による2重登録問題

- 対策
  - 登録処理の後にリダイレクトさせる



リロード(F5)しても、直前のリクエスト(=リダイレクトのURL)が発行されるため、登録処理は行われず、完了画面が表示される

まとめ

ご清聴
ありがとう
ございました