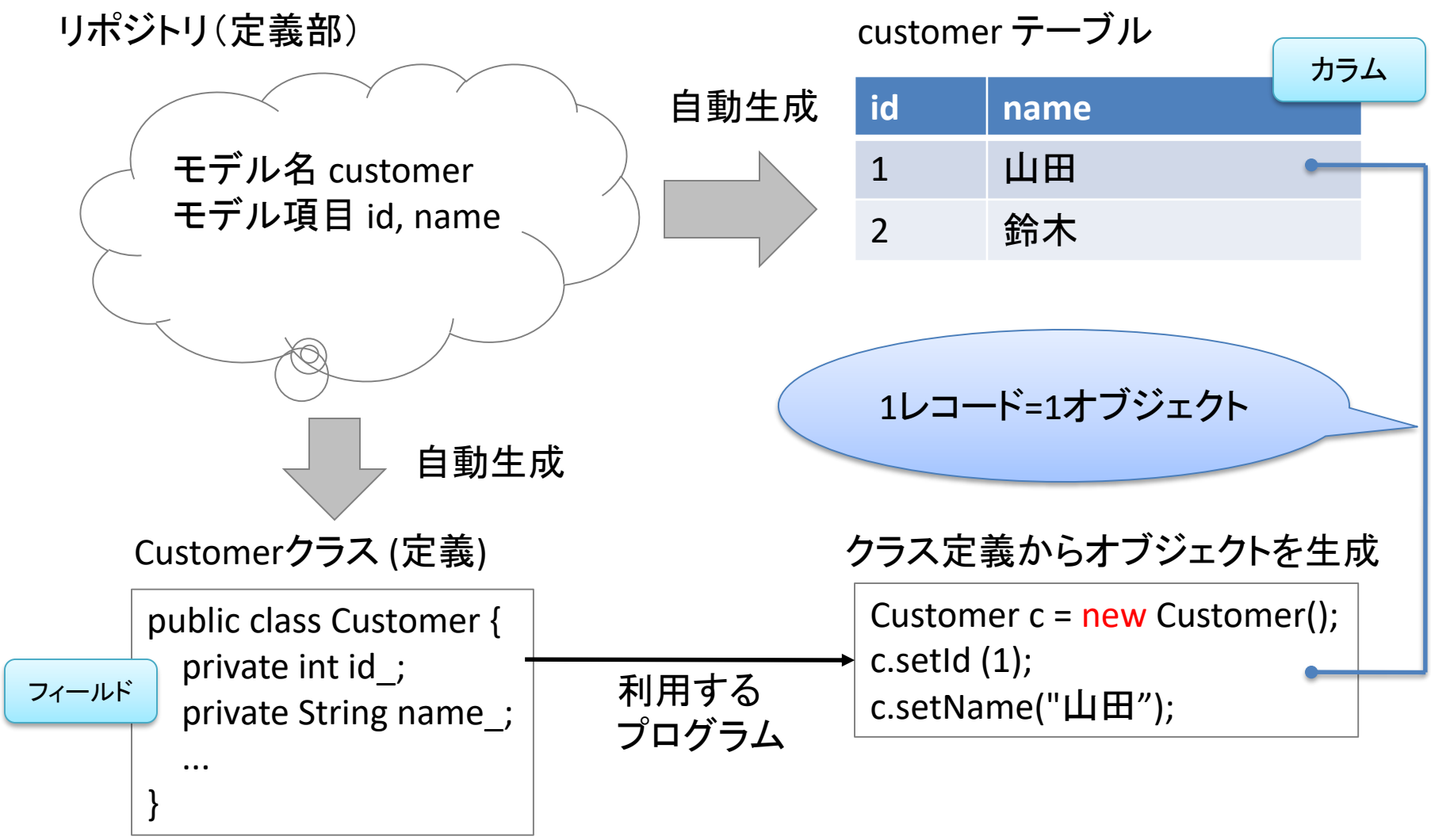
Wagby R8 カスタマイズ開発者向け 基本ガイド

株式会社ジャスミンソフト

WagbyのためのJava基礎知識

モデル定義とテーブル、クラスの関係



Javaクラス =「データの入れ物」として機能します。

データの流れを制御するクラス

利用者

[リクエスト] (要求)
/showCustomer.do?id=1000

id=1000の顧客データの詳細表示画面を見たい。

HTML

顧客モデル詳細表示処理クラス

idは？

1000。

id=1000のデータを取得するオブジェクトを呼び出そう。

Javaオブジェクトがきた。

Viewに渡そう。

顧客モデル詳細表示ビュークラス

オブジェクトの内容を元に、HTMLを作ろう。

表示用オブジェクトを受け取った。

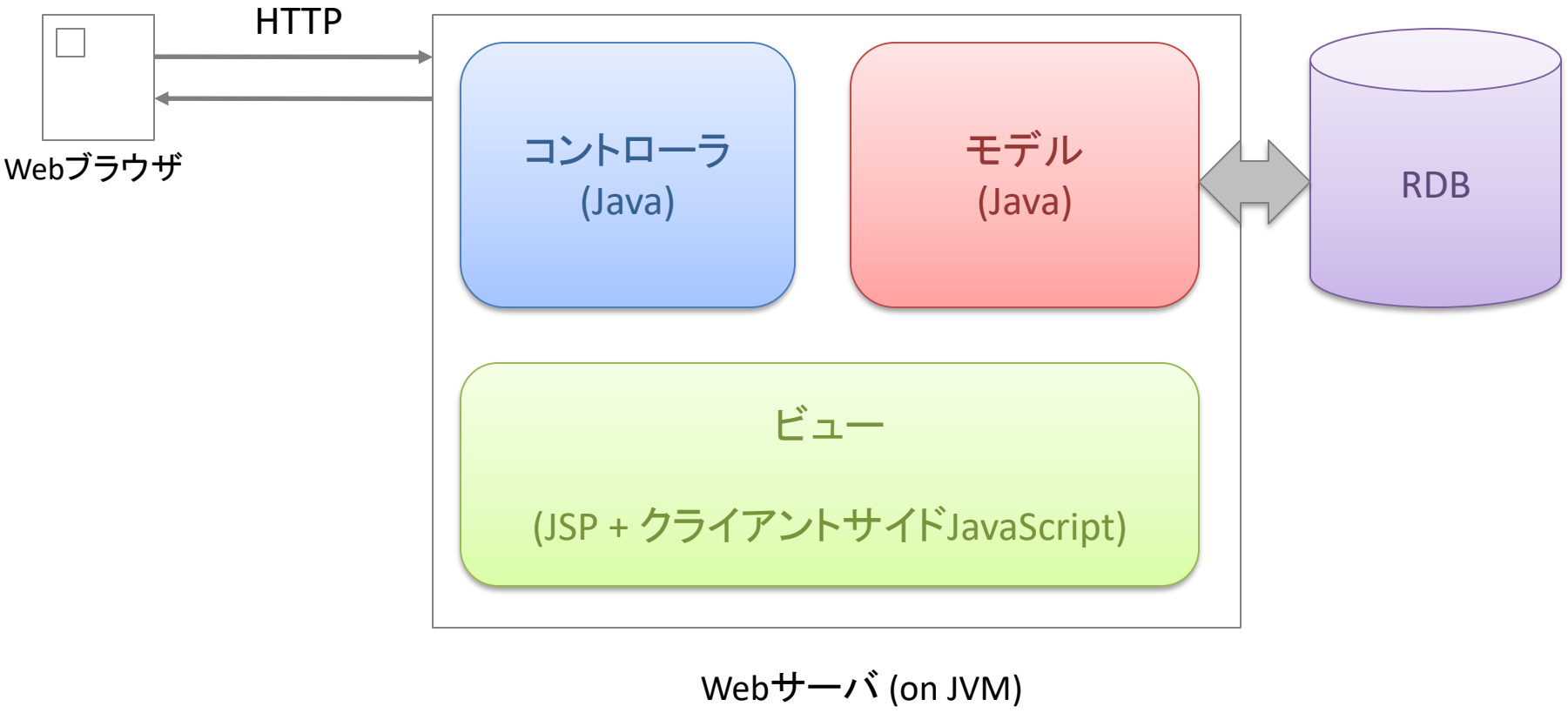
顧客モデルデータ取得クラス

id=1000のデータをデータベースから取得して…

これをJavaのオブジェクトの形で返します。

Javaクラス=「データの取得・変換」や「利用者とのやりとり」といった機能を担います。

モデル / ビュー / コントローラの関係

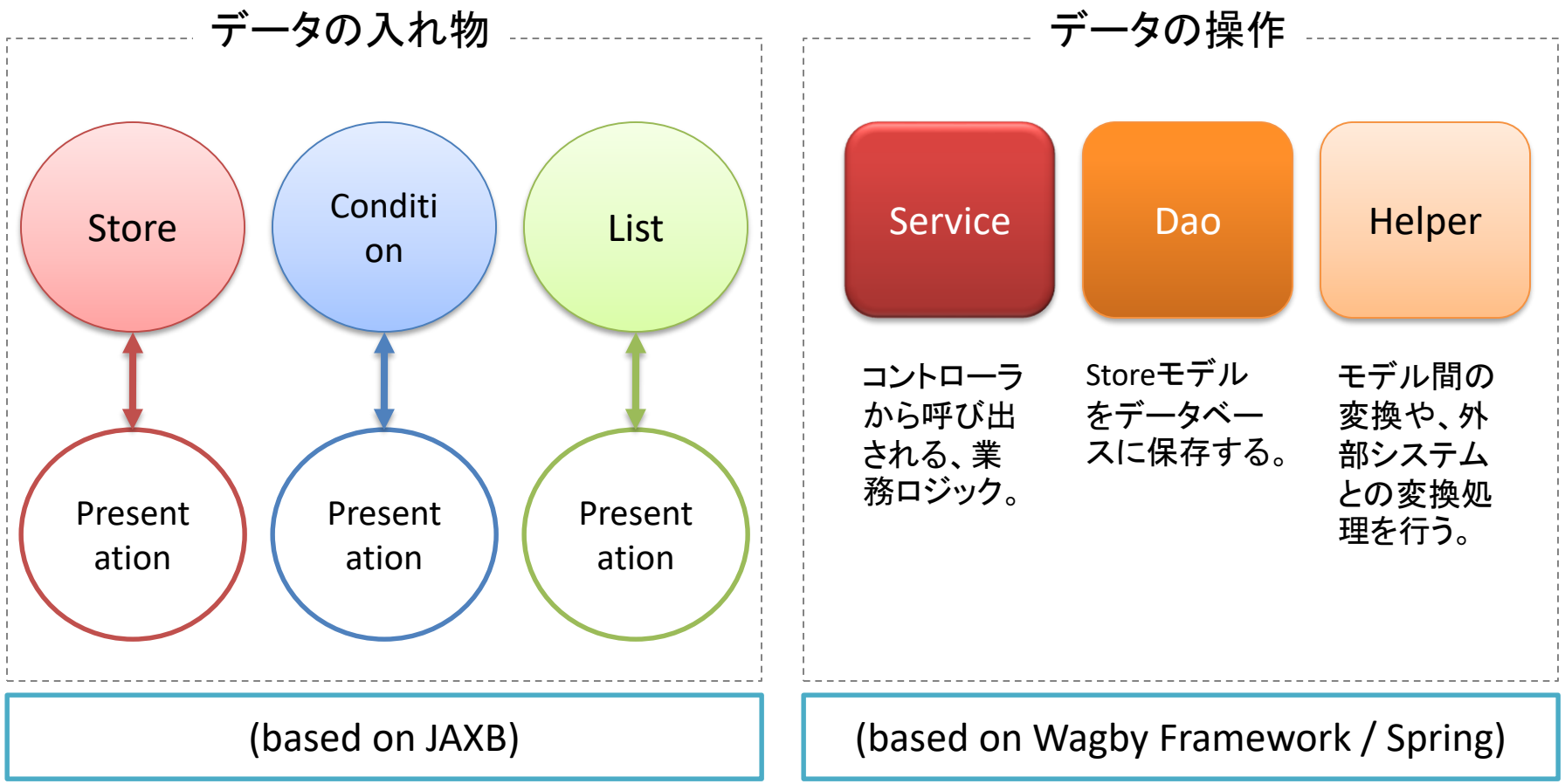


※ HTTP ... インターネット標準規格。Request と Response から構成される。
※ JVM ... Java実行環境。Wagby R8 では Java 8 以上が必要。

Wagbyが取り決めた「クラス」の役割

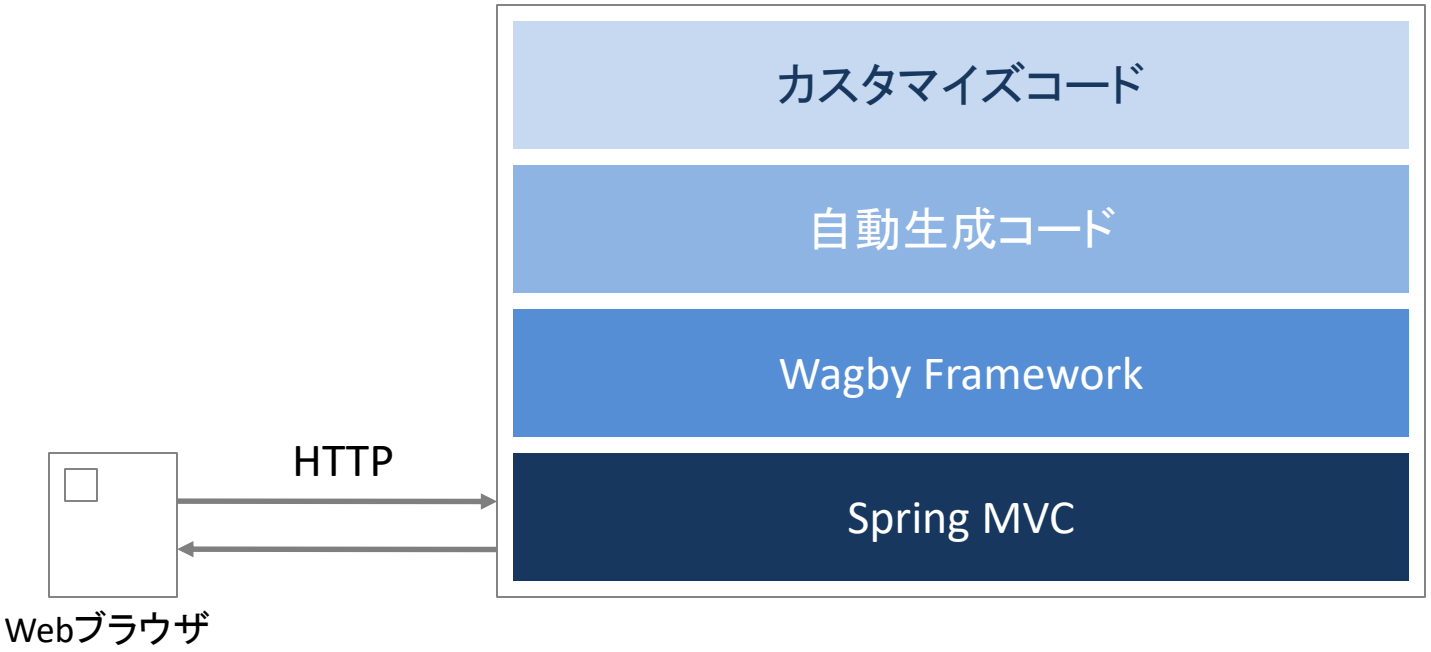
名称	役割
モデル	データ（テーブル上の1レコード）を保持する「入れ物」
DAO	Data Access Object の略。モデルの1データを取得する。特定の1件を取得するために主キーを用いる。検索条件に合致した複数件を取得することもできる。データの登録・更新・削除にも対応している。
サービス	コントローラから呼び出される単位。内部では必要なDAOを呼び出してデータを取得する。トランザクションの境界になっている。
コントローラ	利用者（Webブラウザ) からのリクエスト（要求）を受け付け、必要なサービスを利用する。結果として得られたJavaオブジェクトをビューに渡す。
ビュー	JSPという形式で表現される。これもクラスの一種。コントローラから受け取ったJavaオブジェクトの情報を使って、HTMLを作成する。
ヘルパ	モデルにはさらに複数の種類がある（後述）。モデルとモデルの相互変換を行ったり、初期値設定や計算処理を行う。

モデル



モデル部は「データの入れ物」と「データの操作」に分かれます。すべて自動生成されます。

コントローラ



コントローラ部は、Spring MVC を基盤とし、基本画面（詳細表示、一覧表示、登録、更新、削除、カレンダビュー、集計ビュー、マップビュー、一覧更新、ダウンロード、アップロード）を Wagby Framework 上に実装しています。自動生成コードは、Wagby Framework を前提としたコードになっています。

ビュー

画面毎に生成されるJSPファイルには、「HTML」「JSTL+EL」「JavaScript」「CSS」の4つの技術が含まれています。

モデル

ストアモデル [1] 基本構造

- (RDB上の)「テーブル」の「1レコード」の情報を格納するための Java オブジェクトです。
- Javaオブジェクトとテーブルの関係（マッピング）は、Hibernate というミドルウェアが管理します。
  - Hibernate設定ファイル (.hbmファイル)が自動生成されます。
- JavaオブジェクトとXMLの関係（マッピング）は、JAXB というライブラリが管理します。
  - ストアモデルのソースコード内に、JAXBのためのアノテーションが含まれています。

ストアモデル [2] サンプル

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "customer")
@XmlType(propOrder={ "customerid_", "name_", "namekana_", "email_", "tel_", "fax_",
"companyname_", "companynamekana_", "companytel_", "companyfax_",
"companyzipcode_", "companyaddress_", "companyurl_", "officename_",
"officenamekana_", "officetel_", "officefax_", "officezipcode_", "officeaddress_",
"keyperson_", "retiredate_", "note_" })
public class Customer extends jp.jasminesoft.jfc.app.ContainerBase<Customer>
implements java.io.Serializable, Cloneable {
    @XmlElement(name="customerid")
    private int customerid_;
    @XmlElement(name="name")
    private String name_;
    ...
}
```

「項目名」がフィールドに対応します。

フィールドはprivateで宣言されます。

アクセッサ (setter/getter) が用意されます。

DAOからストアモデルを読み込む

- 1つのストアモデルに対して、1つのDAO (Data Access Object)が用意されます。
- DAOが提供する get メソッドを使って、データベースから1つのデータを Java オブジェクトとして読み込むことができます。

```
// 顧客ID（主キー)が1000の顧客データを読み込む。
Customer customer = customerDao.get(1000);
```

クラス名の命名規則

- モデル名（英語）の先頭を大文字にする。
  - 元々が大文字の場合は、そのまま。
- アンダースコア (_) は除去し、その次の文字を大文字にする。
  - sales_slip は SalesSlip になる。

項目名の命名規則

- 項目名に対応したフィールドが用意される。
- 項目名の末尾にアンダースコアを含める。
  - id は id_ になる。
- アンダースコア (_) を取り除き、その次の文字を大文字にする。
  - customer_id は、customerId_ になる。
- データを取得する getter と、値をセットする setter メソッドが用意される。
  - getCustomerId()
  - setCustomerId(引数)

利用できる型

Wagby Designerで指定した型	Javaの型
整数型	Integer (非・必須) / int (必須)
1,2,4,8バイト整数	Byte, Short, Integer, Long (非・必須) byte, short, integer, long (必須)
4バイト小数	Float (非・必須) / float (必須)
8バイト小数	Double (非・必須) / double (必須)
文字列、ファイル、メール、URL、郵便番号	String
日付	java.sql.Date
時刻	java.sql.Time
日付時間	java.sql.Timestamp

※ 日付、時刻は java.util パッケージではなく java.sql パッケージです。
※ Javaのboolean型に相当する(Wagby Designerの)定義はありません。intで代替します。
※ JavaのBigInteger/BigDecimalクラスに相当する(Wagby Desginerの)定義はありません。
Wagbyが提供する関数ADD,SUB,MUL,DIVは内部でBigIntegerを使った演算を行います。その結果をlongやdouble型で保存するようにします。

整数・小数のバイト数と表現できる数の関係

型とバイト数	表現できる数
1バイト整数	-128 ～ 127
2バイト整数	-32,768 ～ 32,767
4バイト整数	-2,147,483,648 ～ 2,147,483,647
8バイト整数	-9,223,372,036,854,775,808～ 9,223,372,036,854,775,807
4バイト小数	約6～7桁の精度
8バイト小数	15桁の精度

※ 小数の演算には「誤差」が含まれます。Wagbyが提供する関数ADD,SUB,MUL,DIVを使って誤差を最小化してください。

整数型の必須項目の扱い

Designerの型情報	必須	Javaの型情報	初期値	未入力の確認方法
整数型	非・必須	Integer	null	checkメソッド (※)
整数型	必須	int	0	なし

※ check項目名() というメソッドが用意されます。戻り値は boolean 型となり、値は true か false のいずれかです。これを使って未入力かどうかの確認を行うことができます。

※ getメソッドを使った場合、nullの場合は "-1" が返されます。

繰り返し項目の扱い

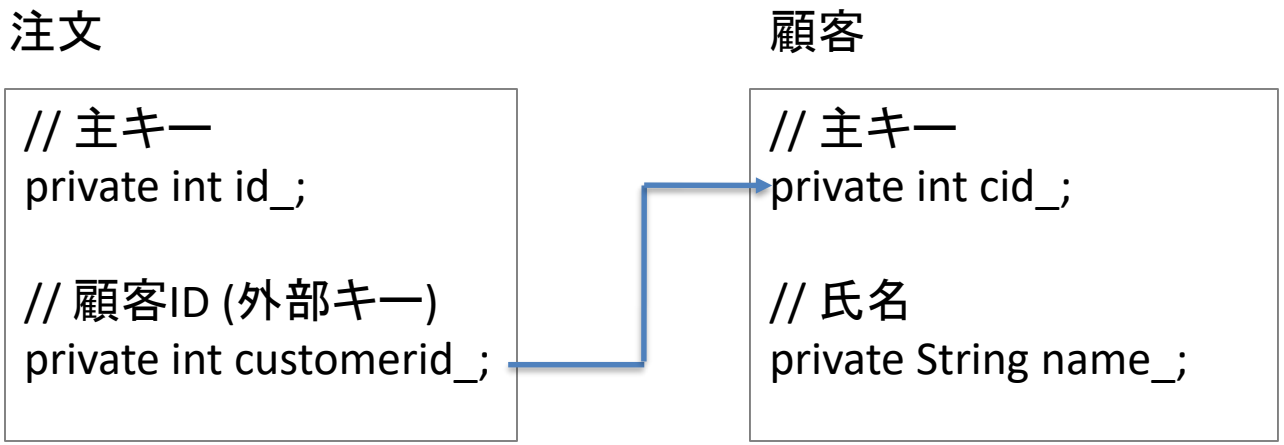
- 繰り返し項目は「文字列」「数字」「日付、時間、日付時間」に指定できます。
- 内部では List というコレクションに複数の値を保持します。
- RDB上では別テーブルが用意されます。自動的に外部キー関係になります。
- DAOからオブジェクトを取得したとき、Listに値は入った状態です。（つまり別テーブルからのデータ取得は必ず行われます。）
- モデルを削除したとき、関連する繰り返し項目も同じタイミングで削除されます。強い結合です。

繰り返しコンテナの扱い

- 繰り返しコンテナは、独立した（コンテナ名の）クラスになります。
- 内部では List というコレクションに複数のコンテナを保持します。
- RDB上では別テーブルが用意されます。自動的に外部キー関係になります。
- DAOからオブジェクトを取得したとき、Listに値は入った状態です。（つまり別テーブルからのデータ取得は必ず行われます。）
- 繰り返しコンテナの入れ子はできません。深いネスト構造によるパフォーマンスの低下、という問題は起こりません。
- モデルを削除したとき、関連する繰り返し項目も同じタイミングで削除されます。強い結合です。

モデル参照 [1]

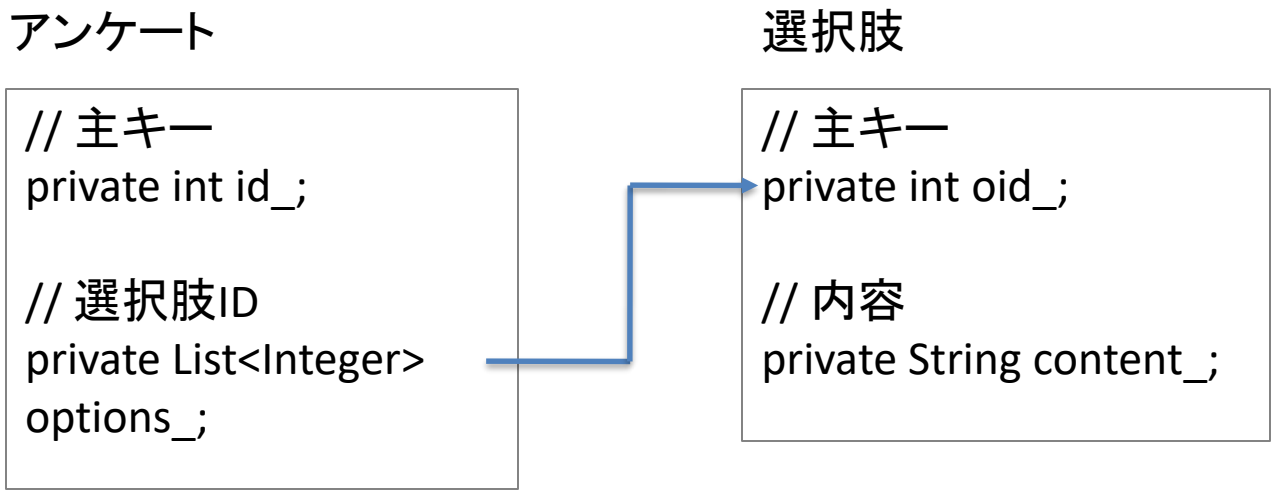
- モデル参照（リストボックス、ラジオボタン、検索ウィンドウ）では、参照先モデルの主キーを保持します。



「注文DAO」からモデルを読み込んだとき、外部キーである「顧客ID」の値は含まれていますが、「顧客」の情報そのものは含まれません。開発者は必要に応じて、「顧客DAO」を使って読み込んでください。

モデル参照 [2]

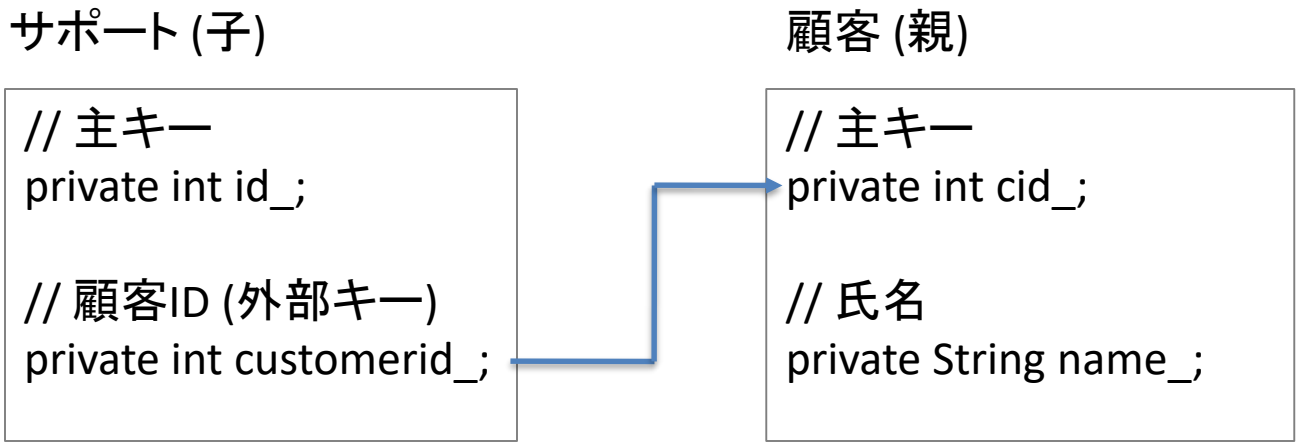
- モデル参照（チェックボックス）では、参照先モデルの主キーをListに保持します。



「アンケートDAO」からモデルを読み込んだとき、外部キーである「選択肢ID」の値を含むリストは用意されますが、「選択肢」の情報そのものは含まれません。開発者は必要に応じて、「選択肢DAO」を使って読み込んでください。

外部キー

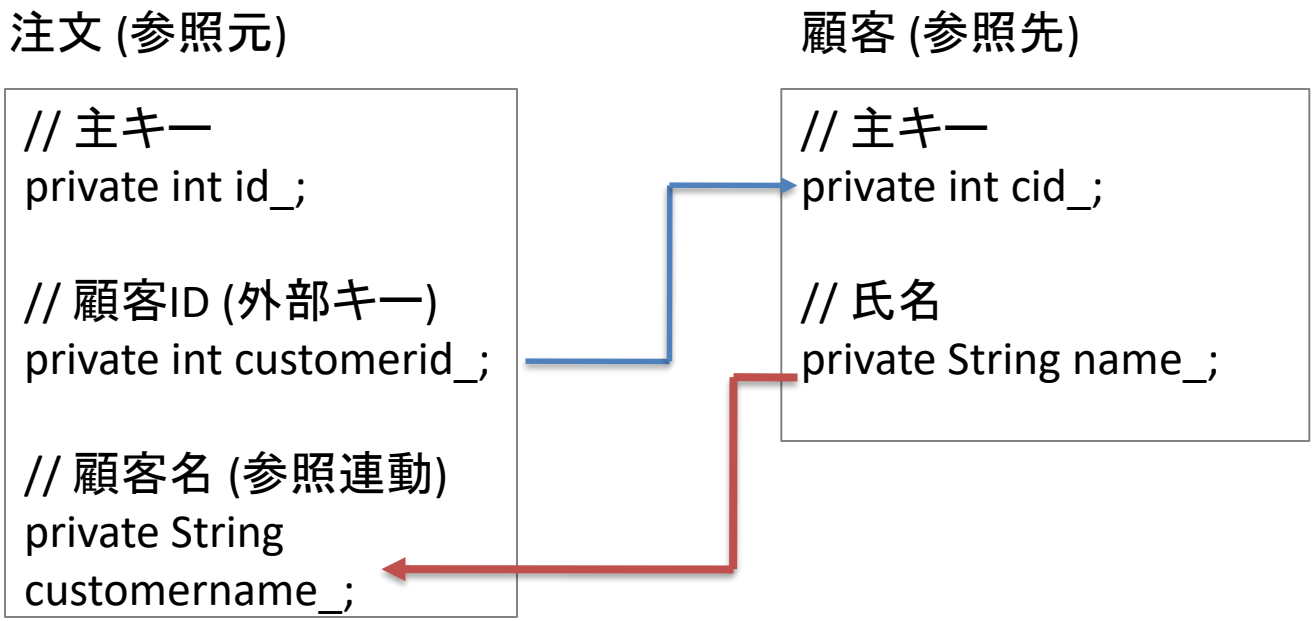
- 親子関係では、親 (1) に対して子 (N) という関係が成立します。親側への記述はなく、子が親のキーを保持します。



このことから、Wagbyの「モデル参照」と「外部キー」に、クラス定義上の差異はありません。ただ「外部キー」と指定することで、親の詳細画面に子の一覧が表示されるといった機能が自動的に付与されます。

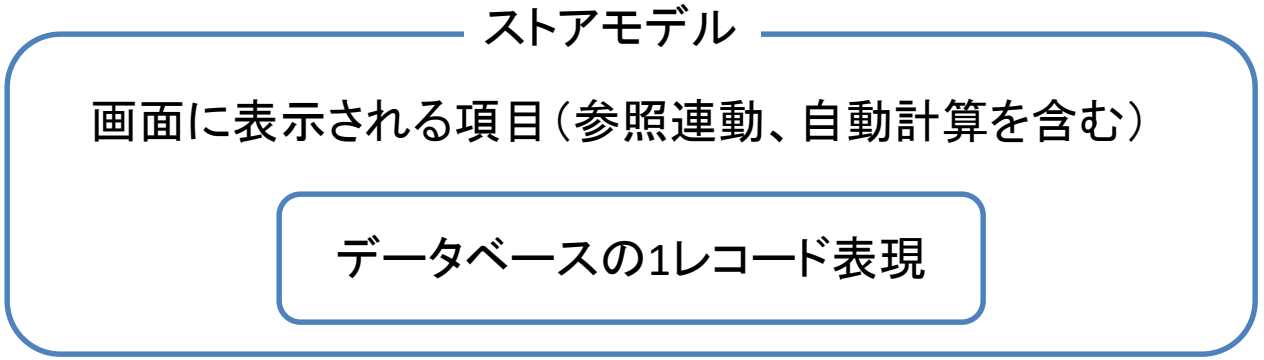
参照連動の動作

- 参照連動は、参照先モデルの値を、参照元に転記する機能です。内部では自動的に顧客DAOからデータを読み込み、値のセット処理を行います。



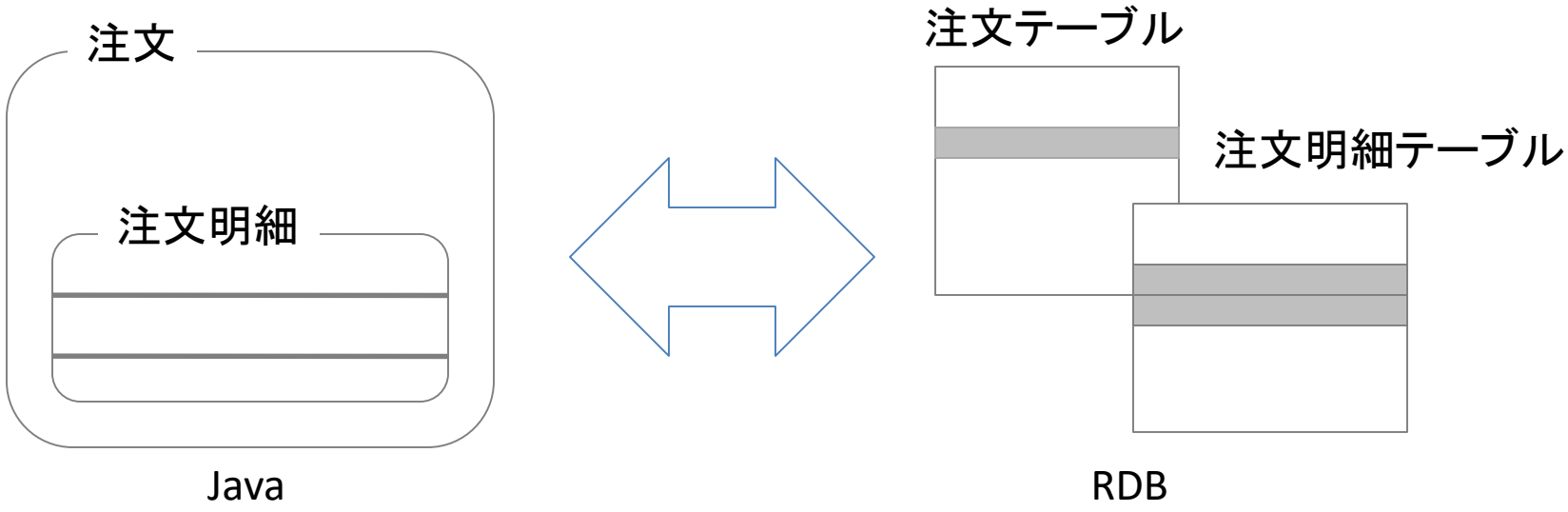
参照連動の目的

- ストアモデルは「データベースに保持される情報」だけでなく「画面に表示される情報」も含めることができます。参照連動はそのために用意された仕組みです。
- もし参照連動機能がないと、ストアモデルとは別に画面表示専用のデータモデルを定義し、そこへ値をセットするコードを手動で作成することになります。



繰り返しコンテナは強い結合を表現する

- 繰り返しコンテナは(RDB上では)別テーブルですが、ストアモデルでは一つのデータ内に明細情報が含まれるようになっています。
- 下の例では、注文モデルを削除すると、注文明細も同時に削除されます。



繰り返しコンテナのレコード管理

- テーブル名は「モデル名$繰り返しコンテナ名」。
- 繰り返しコンテナのテーブルには、繰り返しコンテナID項目が含まれます。
- （親モデルへの外部キーとなる）モデルの主キーと、繰り返しコンテナIDの複合キーでレコードを管理します。

繰り返しコンテナのトランザクション管理

- モデルの取得、登録、更新、削除といったすべての処理で、繰り返しコンテナは自動的にトランザクションの対象になります。

メール、URL、郵便番号型

- ストアモデル上の定義ならびにテーブル定義では、文字列型と同じ扱いになります。

ファイル型

- ファイル型は内部では二つのフィールド（ならびにテーブル上でも二つのカラム）から構成されます。
- 定義上の項目名には、ファイル名が入ります。
- 自動的に用意される「項目名_jshfilename」は、ファイルの実パス情報（文字列）が格納されます。
- ファイル情報そのものはBLOBとして管理しません。あくまでファイル名と実パスとして管理しています。
  - 物理ファイルが削除されてしまうと、参照エラーになります。

標準で提供されるメソッド

- デフォルトコンストラクタ
- コピーコンストラクタ
- cloneメソッド
- makeDocument メソッド
- makeElement メソッド
- makeTextDocument メソッド
- makeTextElement メソッド
- toString メソッド
- getter/setter メソッド（注: final 修飾されています）

項目に提供されるメソッド

- check項目名() ... 非・必須項目のみ
  - boolean (true/false) を返す。
- get項目名AsXXX
  - 例：getUnitAsInteger()
- set項目名(基本型)
- set項目名(オブジェクト型)
  - nullを格納するため。

繰り返し項目/チェックボックス型に提供されるメソッド

- getterメソッドは配列を返す。
- setXXX(配列)
- setXXX(一つの値)
  - 現在の値をクリアして、一つの値をセットする。
- setXXX(index, 一つの値)
  - 配列のindex番目に、値をセットする。（上書き）
- addXXX(一つの値)
  - 現在の値を保持し、一つの値を追加する。
- removeXXX(index)
- removeXXX(一つの値)
- clearXXX()
- sizeXXX()

繰り返しコンテナに提供されるメソッド

- getterメソッドは（繰り返しコンテナ型の）配列を返す。
- setXXX(繰り返しコンテナの配列)
- setXXX(一つの繰り返しコンテナ)
  - 現在の値をクリアして、一つの値をセットする。
- setXXX(index, 一つの繰り返しコンテナ)
  - 配列のindex番目に、値をセットする。（上書き）
- addXXX(一つの繰り返しコンテナ)
  - 現在の値を保持し、一つの値を追加する。
- removeXXX(index)
- removeXXX(一つの繰り返しコンテナ)
- clearXXX()
- sizeXXX()

Java – XML マッピング

- ストアモデルはtoStringメソッドを呼び出すことで自身のXML形式表現へ変換することができます。
- フレームワークに含まれるjp.jasminesoft.util.JaxbUtilクラスのunmarshalメソッドを使うことで、XML文字列からストアモデルを生成することができます。
- XMLマッピングルールはJAXBのアノテーションを使って（ストアモデルのソースコード内に）表現されています。

直列化 (Serializable)

- すべてのストアモデルはjava.io.Serializableインタフェースによってマーキングされています。
- ストアモデルのシリアライズ、デシリアライズにも標準で対応しています。
  - キャッシュされたオブジェクトをテンポラリファイルに保存するなどの用途に使えます。

O/R Mapping ルール

- ストアモデルと（RDB上の）テーブルとのマッピングはHibernateマッピングファイル（拡張子 .hbm）で規定されています。
- Hibernateマッピングファイルも自動生成されます。

複合キー

- 参照元項目では、参照先項目の主キー名を含めたフィールド（およびRDB上のカラム）が用意されます。

初期値

- 初期値の設定は、新規登録画面を開いたとき、およびCSV・Excelファイルアップロード更新時に適用されます。
- 初期値を設定するコードは、ストアモデルに対応したヘルパの initialize メソッド内に自動生成されます。

登録時・更新時

- 登録時および更新時における値の設定は、新規登録ならびに更新処理、およびCSV・Excelファイルアップロード更新時に適用されます。
- 設定するコードは、ストアモデルに対応したヘルパの beforeInsert および beforeUpdate メソッド内に自動生成されます。
- beforeInsertメソッドは通常、主キーの値をセットするコードが生成されます。
- beforeUpdateメソッドは通常、更新日時の値を上書きするコードが生成されます。

InitLoaderによるインポート

- InitLoaderコマンド（または管理者による「インポート・エクスポート」画面からのインポート処理）では、データ1件のインポート直前に、ヘルパの beforeImport メソッドが呼び出されます。

計算式

- リポジトリに記述した計算式は、ヘルパの_calcメソッドに展開されます。
- 「データベースに保存する」場合、表示時はデータベースの値をそのまま使います。登録、更新時は再計算されます。
- 「データベースに保存しない」場合、すべてのタイミングで再計算されます。
- Wagbyが提供する関数の多くは、フレームワークのjp.jasminesoft.util.ExcelFunctionにクラスメソッドとして提供されています。

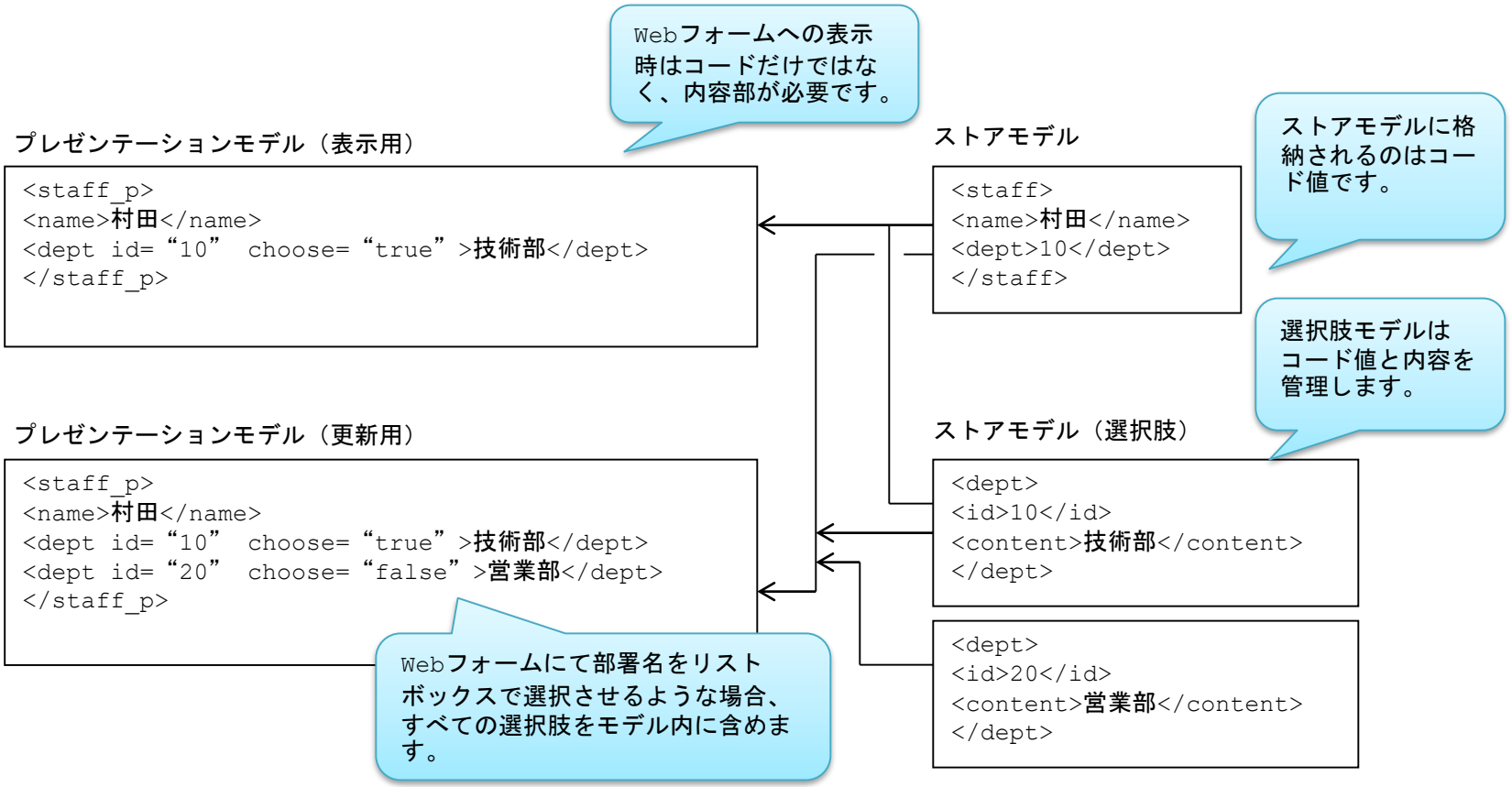
ストアモデルのカスタマイズ方針

- ストアモデル本体は修正しません。
  - 項目の追加や削除などはリポジトリを変更して対応する。
- Hibernateマッピングファイルはできるだけ修正しません。
  - 既存テーブルとのマッピングに関する情報も、リポジトリで設定できる。
- 初期値や計算式は、ヘルパの該当メソッドのオーバーライドにより対応します。
  - "My"を接頭語に付与したクラス定義を用意する。

プレゼンテーションモデル

- ストアモデルにおけるモデル参照項目は、参照先モデルの主キーのみを保持します。これに「内容部」を含めたものがプレゼンテーションモデルです。
  - 便宜上、これを「ID部」「内容部」と呼びます。
- プレゼンテーションモデルには、画面に表示されるモデルに関するすべての情報が含まれます。
- プレゼンテーションモデルは「表示モード」と「更新モード」があります。更新モードは、選ばれなかった他の選択肢もすべて含まれたものです。

プレゼンテーションモデルの生成イメージ



createObject メソッド

- Webフォームからプレゼンテーションモデルを生成するために、（プレゼンテーションモデルの）ヘルパクラスに含まれるcreateObject メソッドを利用します。
- プレゼンテーションモデルに含まれる値はすべて文字列型です。

s2p メソッドと p2s メソッド

- ストアモデルからプレゼンテーションモデルへ変換するためのメソッドは、（プレゼンテーションモデルの）ヘルパクラスに自動生成された s2p メソッドを呼び出します。
- プレゼンテーションモデルからストアモデルへ変換するためのメソッドは、（プレゼンテーションモデルの）ヘルパクラスに自動生成された p2s メソッドを呼び出します。

p2p メソッド

- createObjectメソッドによって用意されたプレゼンテーションモデルは、選択値（ID部）のみを保持しています。
- 入力チェック式において、内容部を判断材料に加えることがあるため、ヘルパの p2p メソッドを呼び出してから、input_check メソッドを呼び出すようになっています。

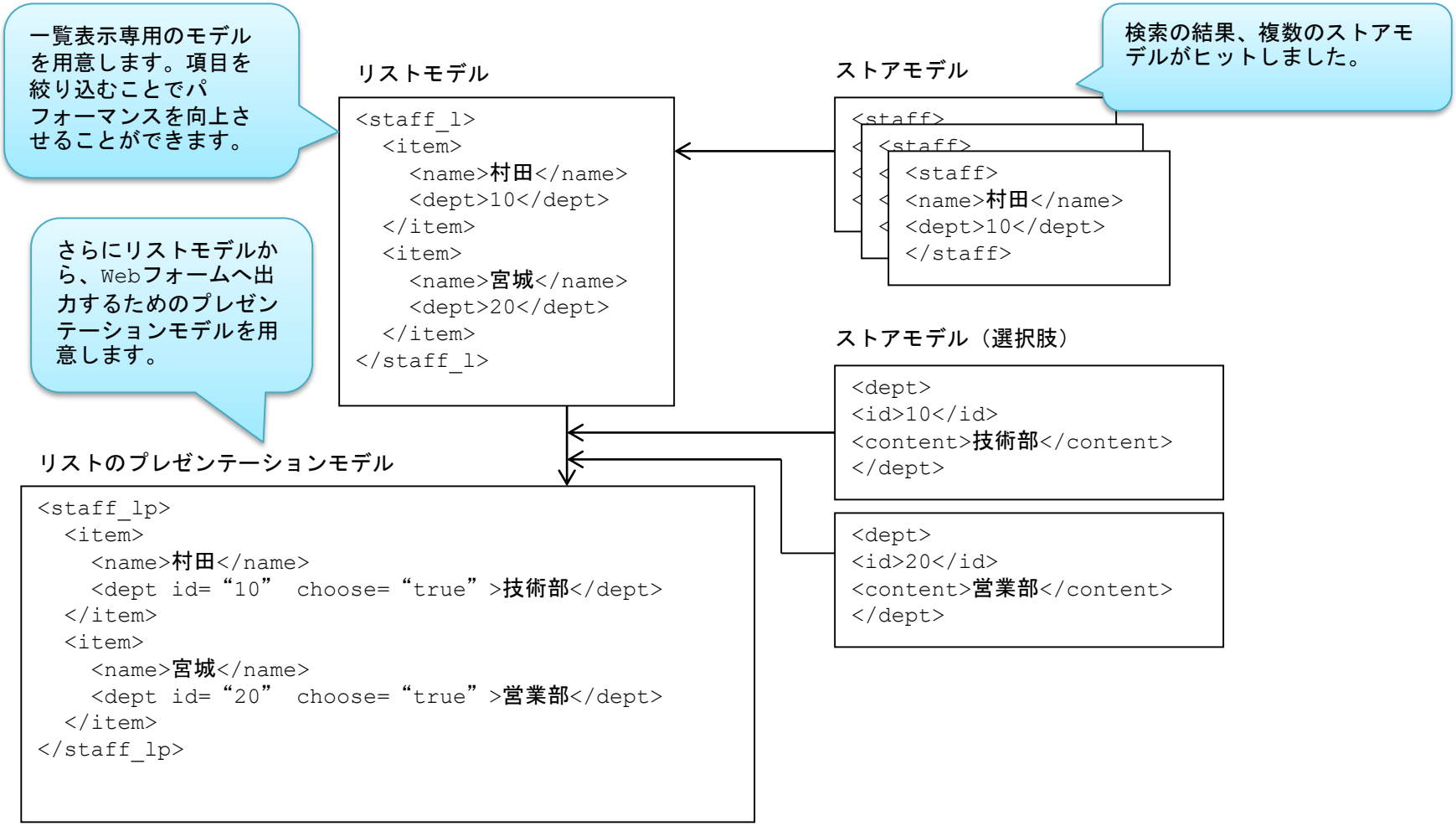
コンディションモデル

- コンディションモデルはストアモデルとほぼ同じ形ですが、データベースに保存されません。
- 数値型と日付型の範囲検索では、二つの入力欄を管理します。
- コンディションモデルは「セッション」に保存されます。検索画面毎にコンディションモデルは記憶されます。

リストモデル

- 検索結果（ストアモデルの集合）から、一覧表示画面に出力させる項目を選択し、「Item」という入れ物に格納しています。
- 表示側では、リストモデルに格納されているItemをループで処理することで一覧表示を実現します。

リストモデルの生成イメージ



コントローラ

Spring MVC

Spring MVCとは

- Spring Framework標準のWebフレームワーク
- MVC(Model-View-Controller)フレームワークです
- バージョン2.5以降でアノテーションが導入されました。
- バージョン3以降でRESTに対応しています。
- バージョン4以降でHTML5/WebSocketに対応しています。

Spring MVCコード例

```
@Controller
public class ShowCustomerController
            extends DbShowController<Customer, CustomerP, Integer>
{
   @RequestMapping(value="/rest/customer/entry/{pkey}")
   public String get(
          @PathVariable("pkey") String pkey,
          HttpServletRequest request, HttpServletResponse response)
         throws IOException, ServletException {
      return super.getRequest(pkey, request, response);
   }
…
```

@Controllerは、このクラスがコントローラクラスであることを示すアノテーション。
@RequestMappingは、http://xxx/xxx/rest/customer/entry/[主キー]にアクセスされた際に、このメソッドを呼び出すことを示すアノテーション。
@PathVariableはURL内の{pkey}で指定した部分をpkeyパラメータに格納するアノテーション。

アノテーション

- あるデータに対して関連する情報（メタデータ）を注釈として付与すること。
- Javaの**アノテーション**は、クラスやメソッド、パッケージに対してメタデータとして注釈を記入します。

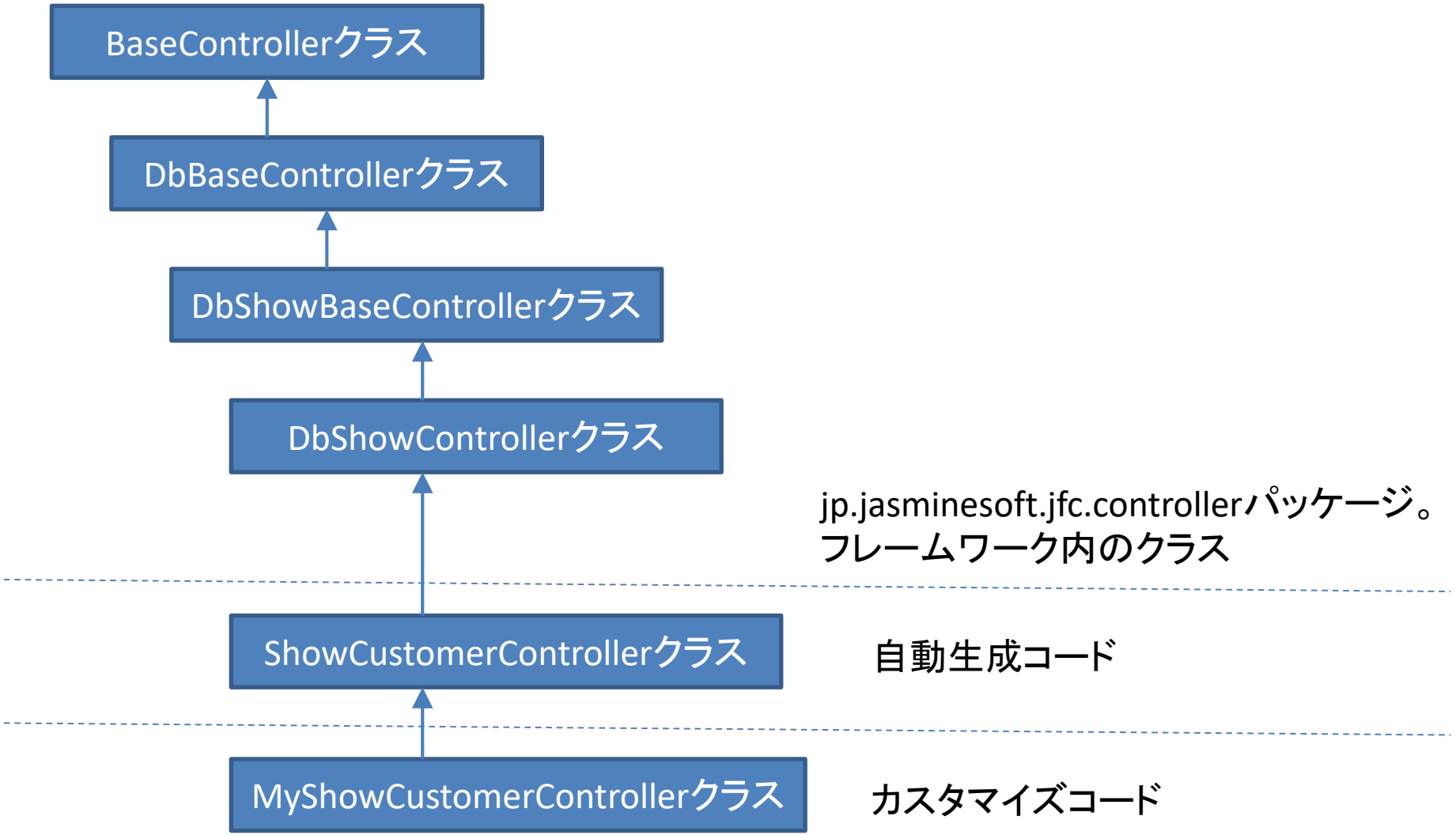
RequestMappingアノテーション (1)

- Springが提供する、URLとControllerメソッドのマッピングを設定するアノテーションです。
- value属性でURLを指定する。[必須]
- method属性で対応するHTTPメソッドを指定します。
  - @RequestMapping(value=“xxx”,method=HttpMethod.POST)

RequestMappingアノテーション (2)

- headers属性でリクエストに含まれるHTTPヘッダー内容を指定します。
  - @RequestMapping(…,headers=”Content-Type=application/x-www-form-urlencoded”)
- その他consumes属性(リクエストのメディアタイプ)、produces属性(レスポンスのメディアタイプ)、params属性(リクエストパラメータ)があります。

クラス階層(詳細表示)


BaseControllerクラス
DbBaseControllerクラス
DbShowBaseControllerクラス
DbShowControllerクラス
jp.jasminesoft.jfc.controllerパッケージ。
フレームワーク内のクラス
ShowCustomerControllerクラス
自動生成コード
MyShowCustomerControllerクラス
カスタマイズコード

wagby-servlet.xml

- Spring MVCの設定ファイルです。
- 次の内容を設定します。
  - Controllerクラスの定義
    - Serviceクラスの設定
  - 画像やCSS、JavaScript(js)ファイルをリソースとして定義
  - ViewResolverの定義
    - Controllerが返した値に対して、どのようなViewにて処理するかを決定する。
    - 例：Controllerが文字列”showCustomer”を返すと、Viewとして showCustomer.jspにて処理する。
  - MessageConverterの定義
    - REST処理にて利用する。（後述）

REST API

RESTとは

- HTTP通信を利用してXMLやJSON形式の情報を送受信するWebアプリケーションです。
- RESTとはRepresentational State Transferの略です。
- 例：http://xxx/wagby/rest/customer/entry/1000にGETメソッドでアクセスすると、主キー1000番に対応したcustomerモデルのデータをJSON形式で返します。

URL設計

例：customerモデルのURL

REST API	画面名	画面操作のURL
/rest/customer/list	検索・一覧表示画面	/showListCustomer.do
/rest/customer/entry/[主キー]	詳細表示画面	/showCustomer.do
/rest/customer/new	新規登録画面	/insertCustomer.do
/rest/customer/edit/[主キー]	編集、削除画面	/updateCustomer.do /deleteCustomer.do

JSONオブジェクト

- JavaScriptにおけるオブジェクトの表記法をベースとした軽量なデータ記述言語です。
- XML形式に比べると、厳密性はないが、軽量です。
- JavaScriptで扱いやすいです。
- JavaでもJackson 等のOSSライブラリを用いることで簡単に扱うことができます。

JSONオブジェクト内の値

```
{
…
   "entity": {
      "customerid_": 1000,
      “name_”: “村田光伸",
      “kananame_”: “ムラタミツノブ”,
      “companyname_”: “株式会社ジャスミンソフト”,
      “companykananame_”: “ジャスミンソフト”,
…
      "updateuserid_": "admin",
      "insertrec_": 1373876508706,
      "updaterec_": 1373876508706
   }
}
```

HTTPステータスコード

- REST処理の成否はHTTPステータスコードにて返します。
- 200番台はSuccess(成功)を示します。
- 400番台はClient Error(クライアント側の問題によるエラー)を示します。
  - 401(Unauthorized) : 許可なし、リクエストはユーザ認証を必要とする。もしくは認証に失敗した場合。
- 500番台はServer Error(サーバ側の問題によるエラー)を示します。

REST API一覧

URL	HTTP メソッド	説明
/rest/session	PUT	ログオン処理を実行
/rest/session	DELETE	ログオフ処理を実行
/rest/session	GET	ログオン状態を確認
/rest/ [モデル名]/list	GET	一覧表示データを取得
/rest/ [モデル名]/list	POST	検索条件を指定してデータを取得
/rest/ [モデル名]/entry/[主キー]	GET	主キーで指定したデータを取得
/rest/ [モデル名]/new	GET	データの新規登録を開始
/rest/ [モデル名]/new	POST	データの新規登録を実行
/rest/ [モデル名]/edit/[主キー]	GET	主キーで指定したデータの更新を開始
/rest/ [モデル名]/edit/[主キー]	PUT	主キーで指定したデータの更新を実行
/rest/ [モデル名]/edit/[主キー]	DELETE	主キーで指定したデータを削除

REST API ログオン (1)

- http://xxx/wagby/rest/sessionにJavaScript等からアクセスします。
  - HTTPメソッド PUT
  - Content-Type: application/x-www-form-urlencoded
    - リクエストのコンテンツタイプ
  - Accept:application/json
    - レスポンスのコンテンツタイプ
  - パラメータ:user=[ユーザ名]&pass=[パスワード]

REST API ログオン (2)

- ログオンに成功した場合、HTTPステータス 200(OK)を返し、ログオンしたユーザの情報などをJSONオブジェクトで渡します。
- ログオンに失敗した場合、HTTPステータス 401(Unauthorized)を返します。
- レスポンスヘッダ内のSet-Cookieで設定されているJSESSIONIDの値を保存します。
  - ログオン以外の呼び出しではJSESSIONIDの値をCookieでリクエストヘッダに設定する必要がある。

REST API 検索・一覧 (1)

- 検索条件を設定せず、1ページ分の検索結果を取得します。
- http://xxx/wagby/rest/customer/listにJavaScript等からアクセスします。
  - HTTPメソッド GET
  - Accept:application/json
    - レスポンスのコンテンツタイプ
  - Cookie:JSESSIONID=[セッションID]
    - セッションID

REST API検索・一覧 (2)

- 検索に成功した場合、HTTPステータス 200(OK)を返します。このとき、検索条件や検索結果の数、一覧のデータをJSONオブジェクトで渡します。
- 検索に失敗した場合、HTTPステータス 400,500番台のコードを返します。

REST API 検索条件付き一覧(1)

- 検索条件を設定して、1ページ分の検索結果を取得します。
- http://xxx/wagby/rest/customer/listにJavaScript等からアクセスします。
  - HTTPメソッド POST
  - Content-Type: application/x-www-form-urlencoded
    - リクエストのコンテンツタイプ
  - Accept:application/json,Cookie:JSESSIONID=[セッションID]
  - パラメータ：これまでのHTMLフォームと同様にパラメータを指定
    - customer_cp$002fcustomerid1jshparam=1010
    - customeridが1010以上のデータを検索する。
- 検索結果は（前ページの説明と同様に）取得できます。

REST API 検索条件付き一覧(2)

- JSONオブジェクトで検索条件を指定します。
- http://xxx/wagby/rest/customer/listにJavaScript等からアクセスします。
  - HTTPメソッド POST
  - Content-Type: application/json
    - リクエストのコンテンツタイプ
  - Accept:application/json,Cookie:JSESSIONID=[セッションID]
  - パラメータ：コンディションのプレゼンテーションモデルのJSON表現

```
{
    "customerid1jshparam_": {
      "content_": "1010"
    }
}
```

REST API 詳細表示

- 主キーをURLに記述します。
- http://xxx/wagby/rest/customer/entry/1000にJavaScript等からアクセスします。
  - HTTPメソッド GET
  - Accept:application/json,Cookie:JSESSIONID=[セッションID]
- 主キー1000に対応するデータをJSONオブジェクトで取得します。

REST API 新規登録(1)

- http://xxx/wagby/rest/customer/newにJavaScript等からアクセスします。
  - HTTPメソッド GET
  - Accept:application/json,Cookie:JSESSIONID=[セッションID]
- 新規登録の初期値データをJSONオブジェクトで返します。

REST API 新規登録(2)

- http://xxx/wagby/rest/customer/newにJavaScript等からアクセスします。
  - HTTPメソッド POST
  - Content-Type:リクエストのコンテンツタイプ
    - application/x-www-form-urlencodedの場合、HTMLフォームと同様のパラメータ
    - application/jsonの場合、プレゼンテーションモデルのJSON表現
  - Accept:application/json,Cookie:JSESSIONID=[セッションID]
- 新規登録処理を実行し、登録後のデータをJSONオブジェクトで返します。
- 初期値の取得が不要であれば、(1)の呼び出しなしに、新規登録を可能とします。

REST API 更新(1)

- http://xxx/wagby/rest/customer/edit/1000にJavaScript等からアクセスします。
  - HTTPメソッド GET
  - Accept:application/json,Cookie:JSESSIONID=[セッションID]
- 更新データ(主キー1000番のデータ)をJSONオブジェクトで返します。
- データのロックが行われます。

REST API 更新(2)

- http://xxx/wagby/rest/customer/edit/1000にJavaScript等からアクセスします。
  - HTTPメソッド PUT
  - Content-Type:リクエストのコンテンツタイプ
    - application/x-www-form-urlencodedの場合、HTMLフォームと同様のパラメータ
    - application/jsonの場合、プレゼンテーションモデルのJSON表現
  - Accept:application/json,Cookie:JSESSIONID=[セッションID]
- 更新処理を実行し、登録後のデータをJSONオブジェクトで返します。
- データの取得やロックが不要であれば、(1)の呼び出しなしに、更新を可能とします。

REST API 削除

- http://xxx/wagby/rest/customer/edit/1000にJavaScript等からアクセスします。
  - HTTPメソッド DELETE
  - Accept:application/json,Cookie:JSESSIONID=[セッションID]
- 更新の場合とURLが同じだが、HTTPメソッドにより判定されます。

REST APIのSpring MVCコード

- ResponseBodyアノテーション
  - Map<String, Object>の値とレスポンスHTTPコードを格納しています。
  - Map<String, Object>の値はMessageConverterにより、JSONオブジェクトに変換されます。

```
@RequestMapping(value="/rest/customer/new", method = POST,
      headers="Content-Type=application/x-www-form-urlencoded")
@ResponseBody
public ResponseEntity<Map<String, Object>> insertRest(
      HttpServletRequest request, HttpServletResponse response)
      throws IOException, ServletException {

   return super.insertRest(request, response);
}
```

REST APIのSpring MVCコード

- RequestBodyアノテーション
  - リクエストに格納されているJSONオブジェクトをプレゼンテーションモデル(CustomerP)に変換して、引数に渡します。
  - 変換はMessageConverterが行います。

```
@RequestMapping(value="/rest/customer/new", method = POST,
      headers="Content-Type=application/json")
@ResponseBody
public ResponseEntity<Map<String, Object>> insertRest(
      HttpServletRequest request, HttpServletResponse response,
      @RequestBody CustomerP entityp)
      throws IOException, ServletException {

   return super.insertRest(request, response, entityp);
}
```

MessageConverter

- wagby-servlet.xmlで指定されています。
- *org.springframework.http.converter.json.MappingJackson2HttpMessageConverter*
  - JSONオブジェクトとJavaオブジェクトの相互変換を行うライブラリJacksonが必要。
  - Jacksonを用いて、リクエストのJSONオブジェクトからJavaオブジェクトへの変換を行う。
  - レスポンスのJavaオブジェクトからJSONオブジェクトへの変換を行う。

呼び出し方 (1)

- JavaScriptによる呼び出し
  - XMLHttpRequestによる呼び出し
  - dojoやjquery、prototype.js等のJavaScriptライブラリを用います。
- Androidからの呼び出し
  - Spring for android：REST呼び出しのライブラリ
    - AndroidHttpClientやDefaultHttpClient等の標準のクラスもある。
  - Jackson:JSONオブジェクトとJavaオブジェクトの変換を行うライブラリ。

呼び出し方 (2)

- その他さまざまな環境からの呼び出し
  - HTTPリクエストを送ることができれば、どのような環境でも可能です。
  - JSONオブジェクトのパース方法を検討する必要があります。
    - 取得する情報を限定すれば、文字列処理でも可能です。
      indexOf,substring等

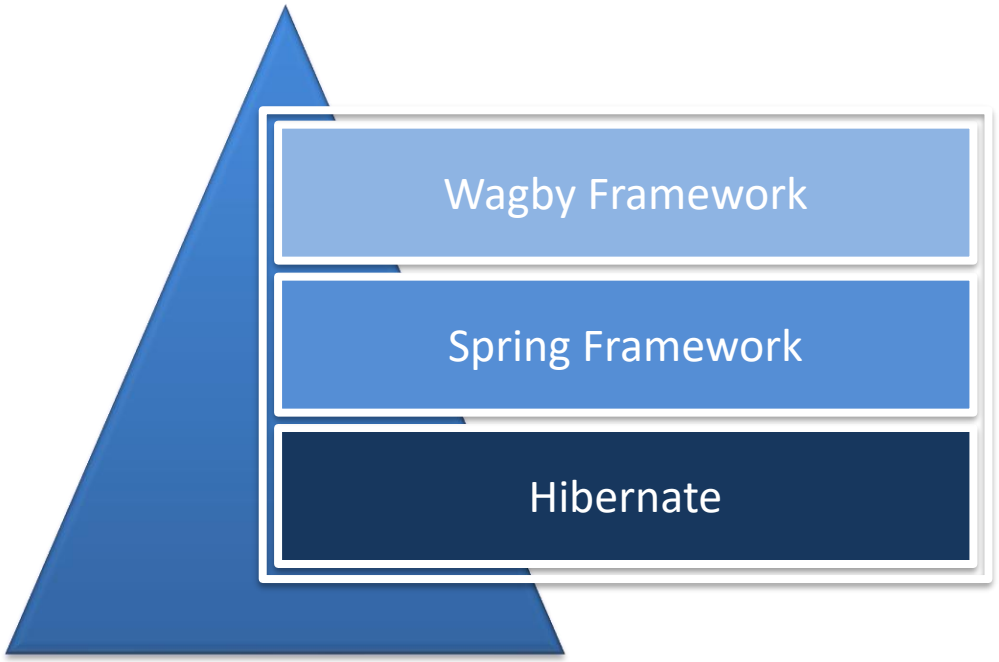
JavaScriptコード例

```
xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) { // DONE
    if (xhr.status == 200) { // OK
      alert(xhr.responseText);
       var ret = eval(xhr.responseText);
       alert(ret.customer.customerid_);
    } else {
      alert("status = " + xhr.status);
    }
  }
}
xhr.open("GET", “/wagby/rest/customer/entry/1000");
xhr.send();
```

データベースアクセス

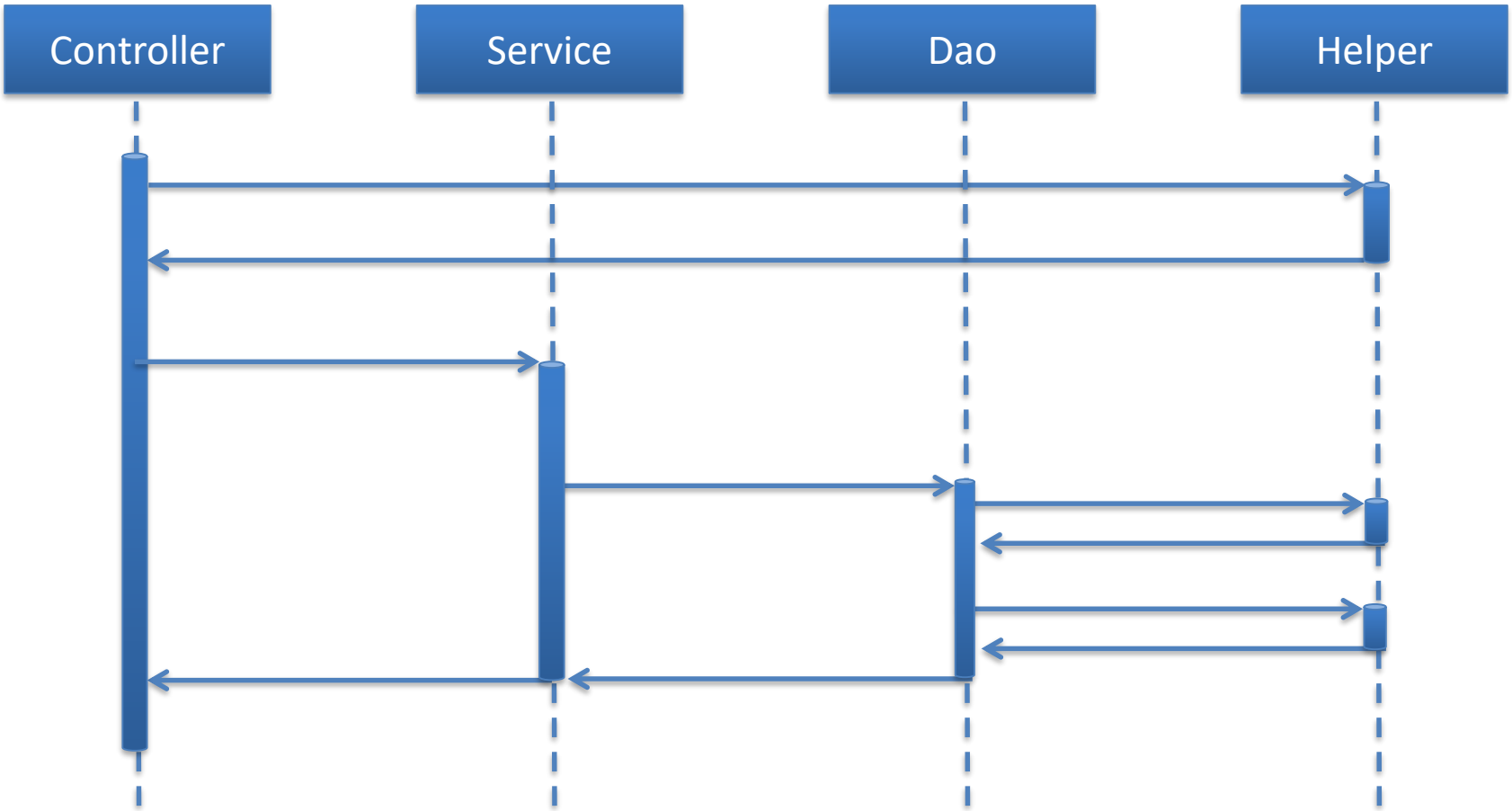
データベースアクセスの基盤

- ORM：Hibernate
- トランザクション管理：Spring Framework

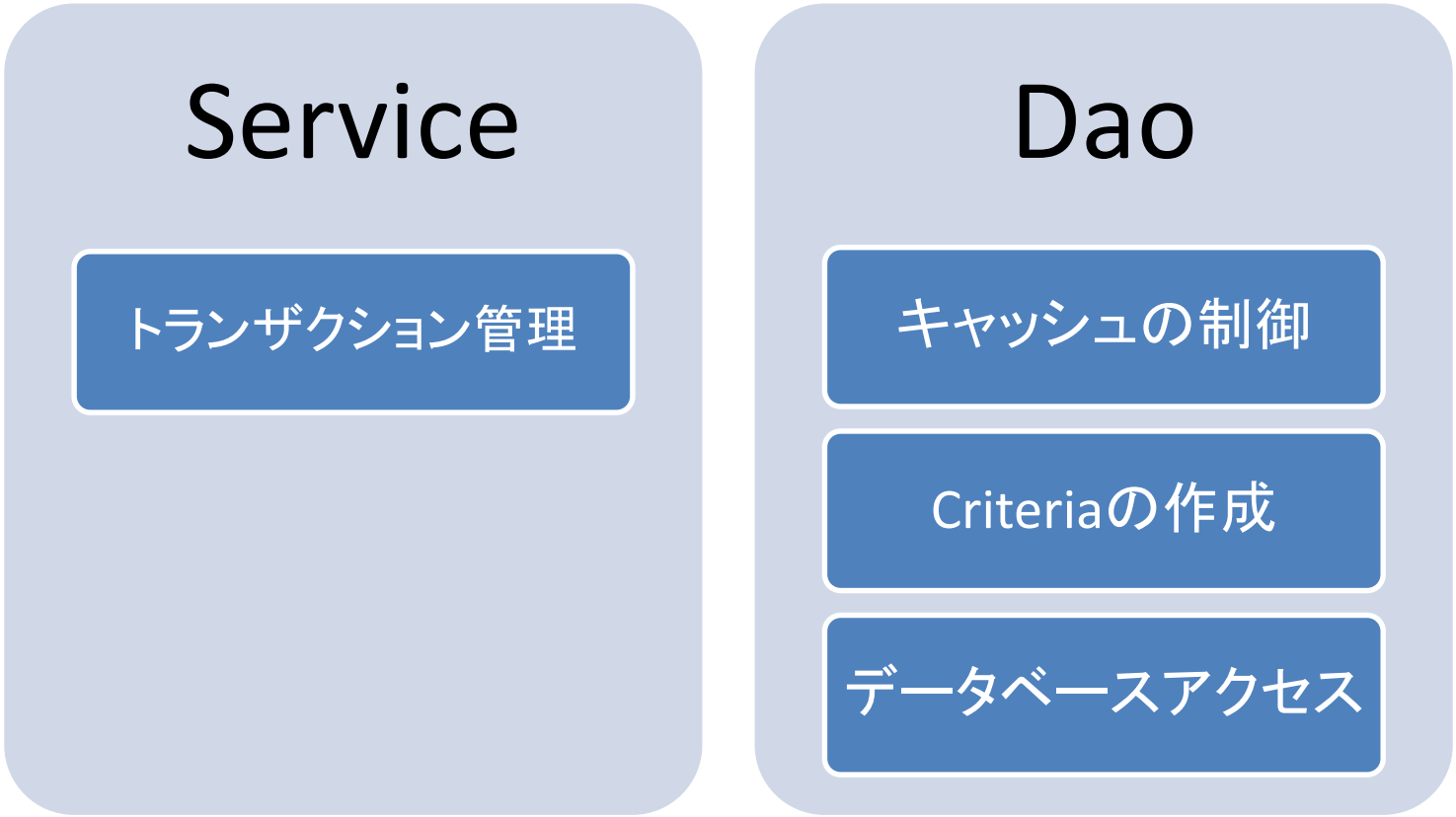


データベースアクセス

- ControllerからServiceを経由してDaoを呼び出す

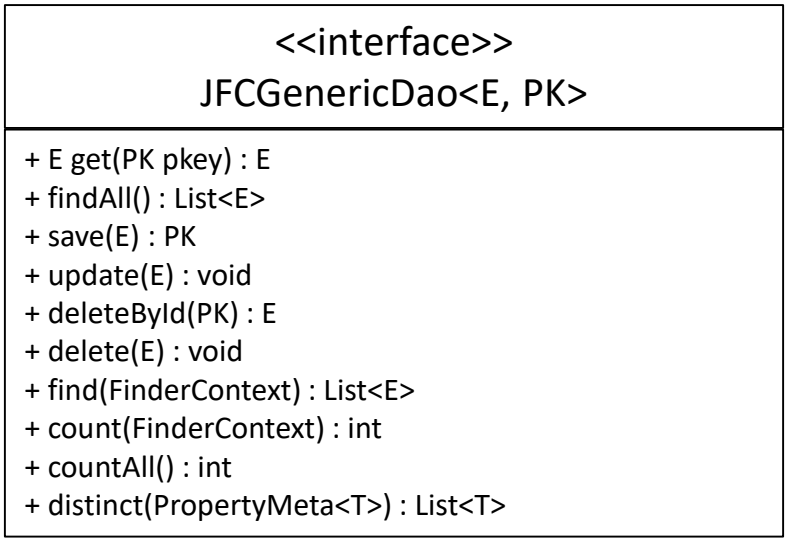


Service/Dao



Dao(Data Access Objects)

- ORMフレームワークHibernateを中心とした設計
  - CRUD処理を一つのDaoクラスで実現
  - オブジェクト指向のCriteriaクエリAPIを採用

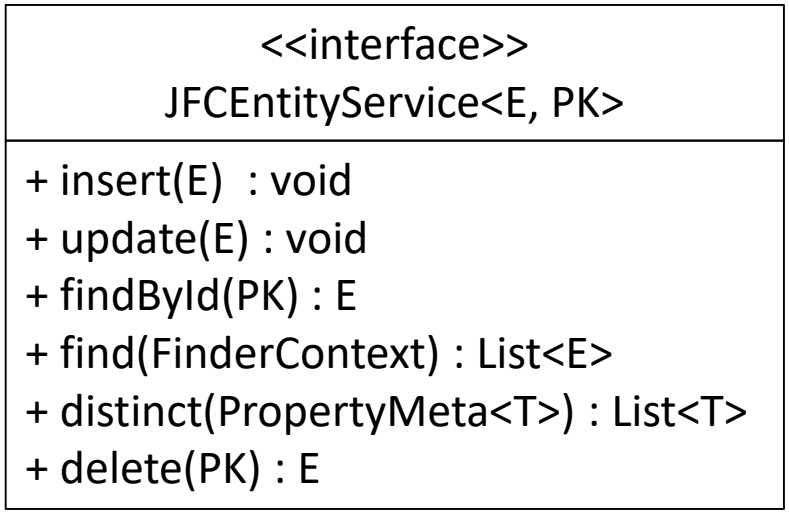


Daoの利用例

```
// 主キー1000のデータを取得(第二引数を true とすることで悲観ロックを同時に行う)
Customer customer = customerDao.get(1000, true);

// データを更新
customer.setName("ジャスミン太郎");
customerDao.update(customer);

// データを登録
Customer customer2 = new Customer(customer);
customer2.setCustomerid(1001);
customerDao.save(customer2);

// 主キー2000のデータを削除(第二引数を true とすることで悲観ロックを同時に行う)
customerDao.deleteById(2000, true);
```

Serviceクラス

- トランザクション境界となるクラス



複合キーへの対応方法

- Daoのget()メソッドは引数がひとつ

JFCHibernateDao

```
public E get(PK pkey, boolean forUpdate) {
    if (pkey == null) {
        return null;
    }
    // キャッシュから取得
    E ret = helper.getCache(pkey);
    if (ret == null) {
        E entity = get(entityClass, pkey);
        ret = entity;
        // LazyInitializationException 対策
        ret = ClassUtils.newInstanceByCopyConstructor(
                entityClass, ret);
        evict(entity);
        // キャッシュに put
        helper.putCache(ret);
    }
    if (forUpdate) {
        LockUtils.lock(ret, helper);
    }
    helper.beforeShow(ret);
    return ret;
}
```

複合キークラス

- 複合キークラスを導入
  - get()メソッドの引数には複合キークラスを指定します。

DailyReport.Key
- staffid : String - reportdate: Date + getStaffid() : String + getReportdate() : Date + setStaffid(String) : void + setReportdate(Date) : void

```
// 複合キーインスタンスの作成
DailyReport.Key key = new DailyReport.Key();
// 主キーをセット
key.setStaffid("M20141107001");
key.setReportDate(DateUtil.getSQLDate("2014-11-07"));
// データの取得
DailyReport dailyReport dao.get(key);
```

コードのスリム化

- Generics, Annotationを中心とした実装へ
  - フレームワークへ処理を移譲
  - 自動生成コードの削減

CustomerDao

```
public class CustomerDao extends JFCHibernateDao<Customer, Integer> {
    /** 処理対象エンティティのメタ情報。 */
    protected CustomerMeta meta;
    /**
     * コンストラクタ。
     */
    public CustomerDao() {
        this(new CustomerMeta());
    }
    /**
     * コンストラクタ。
     * @param meta 処理対象エンティティのメタ情報
     */
    public CustomerDao(CustomerMeta meta) {
        super(meta.entityClass());
        this.meta = meta;
    }
    /** {@inheritDoc} **/
    @Override
    public DetachedCriteria primarykeyCriteria(
            Integer primarykey, DetachedCriteria criteria) {
        criteria.eq(meta.customerid, primarykey);
        return criteria;
    }
}
```

トランザクション

トランザクション管理

- Serviceクラスにアノテーションを指定して制御します。
  - メソッド開始時に自動的にトランザクション開始
  - Exceptionが発生すれば自動的にロールバック
  - メソッドが正常終了すれば自動的にコミット
- コードの記述量が減少します。
  - 記述忘れの抑制
  - 可読性向上

```
public interface JFCEntityService<E, PK  extends Serializable> {
    /**
     * データの登録処理を行います。
     *
     * @param entity 登録対象のデータ
     */
    @Transactional
    void insert(E entity);
```

トランザクション戦略

- Service内のメソッドから他のサービスのメソッドを呼び出します。
  - 例)
    - あるサービス処理内で注文伝票の作成を行いたい
    - OrderEntityService#insert() メソッドを使えば注文伝票の作成及び在庫引当まで行なってくれる
    - 当サービス内のメソッドからOrderEntityService#insert()を呼び出す実装を追加
    - すでにトランザクションは開始されているが、そこからOrderEntityService#insert()を呼び出すとどうなる?

トランザクション戦略

- デフォルトはPROPAGATION_REQUIRED
  - トランザクションが開始されていればそれを継続し、存在しなければ新しいトランザクションを生成します。
- トランザクション属性を変更したい場合はアノテーションに属性を追加します。

```
public interface JFCEntityService<E, PK  extends Serializable> {
    /**
     * データの登録処理を行います。
     *
     * @param entity 登録対象のデータ
     */
    @Transactional(propagation = Propagation.REQUIRED)
    void insert(E entity);
```

トランザクション処理

出庫データの追加に伴う在庫引落処理

```
public class MySyukkoEntityService extends SyukkoEntityService {
 ...
    /** {@inheritDoc} **/
    @Override
    public void insert(Syukko syukko) {
        super.insert(syukko); // 既存処理の呼び出し
         // 在庫データの取得(同時に悲観ロックも行う)
        Zaiko zaiko = zaikoEntityService.findById(syukko.getShohinId(), true);
         // 業務処理 在庫チェック
        int suryou = zaiko.getSuryou();
        int syukkoNum = syukko.getSyukkoNum();
        if (suryou - syukkoNum < 0) {
            throw new BusinessLogicException("在庫 " + suryou + " に対して "
                    + syukkoNum + " を出庫しようとしました。");
        }
         // 業務処理 在庫数の調整処理
        zaiko.setSuryou(suryou - syukkoNum);
        zaikoEntityService.update(zaiko);
    }
```

トランザクション処理

- EntityService, Dao, Helperインスタンスの取得
  - EntityService

```
/** ZaikoEntityService */
@Autowired
@Qualifier("ZaikoEntityService")
protected JFCEntityService<Zaiko, Integer> zaikoEntityService;
```

  - Helper

```
/** ZaikoHelper */
@Autowired
@Qualifier("ZaikoHelper")
protected EntityHelper<Zaiko, Integer> zaikoHelper;
```

  - Dao

```
/** ZaikoDao */
@Autowired
@Qualifier("ZaikoDao")
protected JFCHibernateDao<Zaiko, Integer> zaikoDao;
```

トランザクション処理

- トランザクション処理用メソッド
  - 参照連動項目(参照先保存)、外部キーモデルで利用
  - 登録処理時に関連データの更新を行う
    - updateRelatedModels**Insert**Transaction（E)
  - 更新処理時に関連データの更新を行う
    - updateRelatedModels**Update**Transaction（E)
  - 削除処理時に関連データの更新を行う
    - updateRelatedModels**Delete**Transaction(E)
  - 外部キーモデル削除処理時に関連データの削除を行う
    - cascadeDelete(E)

Hibernateとの連携

Hibernate連携

- Criteriaを使った検索を行ないます。
  - 文字列(HQL)組立処理の除去
  - 可読性の向上
  - オブジェクト指向に沿ったコード

```
CustomerMeta meta = new CustomerMeta();
DetachedCriteria criteria = DetachedCriteria.forClass(Customer.class);

// 顧客名の部分一致検索
criteria.like(meta.customername, condition.getCustomername());

// 部署番号の完全一致検索
criteria.eq(meta.deptcode, condition.getDeptcodeAsInteger());

// 検索結果の取得
@SuppressWarnings("unchecked")
List<Customer> results = executableCriteria.list();
```

拡張DetachedCriteria

- 標準のDetachedCriteriaを拡張
  - 検索条件値のNULLチェック不要

```
if (condition.getCustomername() != null) {
    criteria.like(meta.customername, condition.getCustomername());
}
if (condition.getDeptcodeAsInteger() != null) {
    criteria.eq(meta.deptcode, condition.getDeptcodeAsInteger());
}
```

  - 検索条件値がNULLの場合は絞り込み条件として利用しない

```
criteria.like(meta.customername, condition.getCustomername());
criteria.eq(meta.deptcode, condition.getDeptcodeAsInteger());
```

  - IS NULL検索を行う場合(2種類の方法を提供)

```
criteria.isNull(meta.customername);
criteria.eq(meta.customername, NULLVALUE);
```

拡張DetachedCriteria(2)

- between検索

```
criteria.between(meta.createdAt,
        condition.startDate(), condition.endDate());
```

  - startDate(), endDate() いずれもNULLの場合は絞り込み条件として利用しない
  - startDate() のみNULLの場合次のコードに内部変換

```
criteria.le(meta.createdAt, condition.endDate());
```

  - endDate() のみNULLの場合次のコードに内部変換

```
criteria.ge(meta.createdAt, condition.startDate());
```

拡張DetachedCriteria(3)

- 繰り返しコンテナ内の項目の検索
  - 通常の criteria プログラミング

```
// 繰り返しコンテナの検索用サブクエリ
DetachedCriteria subquery
        = DetachedCriteria.forClass(Report.class, "report_");
// 副問い合わせの結合条件(主キーを結合条件とする)
subquery.setProjection(Projections.id());
subquery.add(Restrictions.eqProperty(
        criteria.getAlias() + ".customerid_", "customerid_"));
// 繰り返しコンテナ内項目への部分一致検索
subquery.add(Restrictions.between("rdate_",
        condition.getRdate1jshparam(), condition.getRdate2jshparam()));
criteria.add(Subqueries.exists(subquery));
```

  - 拡張DetachedCriteria

```
criteria.between(meta.rdate,
        condition.getRdate1jshparam(), condition.getRdate2jshparam());
```

タイプセーフなプログラミング

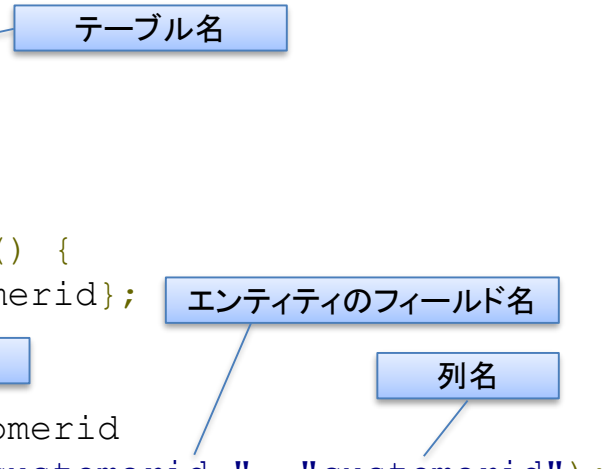
- Metaクラスを利用し、項目名の文字列入力を避ける
  - タイプミスはコンパイルエラーで検知
  - IDEを使ったコード補完

```
CustomerMeta meta = new CustomerMeta();
DetachedCriteria criteria = DetachedCriteria.forClass(Customer.class);
// customerid を検索するための Criteria を組み立てる(数値の範囲検索)。
criteria.between(meta.customerid,
        condition.getCustomerid1jshparamAsInteger(),
        condition.getCustomerid2jshparamAsInteger());
// deptname を検索するための Criteria を組み立てる(部分一致検索)。
criteria.like(meta.deptname, condition.getDeptname());
// 繰返し項目 email を検索するための Criteria を組み立てる。
criteria.like(meta.email, condition.getEmail());
// 繰り返しコンテナ内の項目 rdate を検索するための Criteria を組み立てる。
criteria.between(meta.rdate, condition.getRdate1jshparam(),
        condition.getRdate2jshparam());
// ソート順を設定する。
criteria.addOrder(orderBy(sortKey));
Criteria executableCriteria
        = criteria.getExecutableCriteria(getCurrentSession());
@SuppressWarnings
("unchecked")List<Customer> results = executableCriteria.list();
```

Metaクラス

- 項目名やテーブル名、列名の文字列を一括管理するクラス

```
public class CustomerMeta extends EntityMeta<Customer> {
    public CustomerMeta() {
        super(Customer.class, "customer");
    }
    /** {@inheritDoc} */
    @Override
    public PropertyMeta<?>[] primarykeyMeta() {
        return new PropertyMeta<?>[] {customerid};
    }
    /** customer/customerid */
    public final PropertyMeta<Integer> customerid
        = new PropertyMeta<Integer>(this, "customerid_", "customerid");
    /** customer/name */
    public final PropertyMeta<String> name
        = new PropertyMeta<String>(this, "name_", "name");
```



項目のメタ情報(2)

- 項目のメタ情報を持つMetaクラス
  - 通常項目：PropertyMeta
  - 他モデルの参照項目：RelationMeta
    - 参照先モデル項目の情報を持つ
  - 他モデルの参照連動項目：ReferenceMeta
    - 参照先モデル項目及び連動元項目の情報を持つ
  - 繰返し項目：MultiPropertyMeta
    - 繰返し項目用別テーブルのテーブル名情報を持つ
  - 繰り返しコンテナ項目：ContainerMeta
    - 繰り返しコンテナ項目用別テーブルのテーブル名情報を持つ
  - 繰り返しコンテナ内項目：ContainerPropertyMeta
    - 繰り返しコンテナ項目のメタ情報を持つ

Metaクラス

- Wagbyが提供するフレームワークでは文字列を扱わないようにします。
  - Metaクラスを中心に扱うことで項目名、列名、フィールド名の区別を意識させません。
  - フレームワーク内部で適切な変換を自動的に行います。
    - Meta→項目名
    - Meta→フィールド名
    - Meta→列名
- それでも文字列を扱う必要がある部分
  - request.getParameter(“XXXX”) など

CriteriaConverter

- ConditionをCriteriaへ変換
  - 設計情報から自動生成されます。

```
// CriteriaConverter インスタンスを作成
CustomerCriteriaConverter converter
        = (CustomerCriteriaConverter) p.appctx.getBean(
                "CustomerCriteriaConverter");

// Condition を Criteria に変換
DetachedCriteria criteria = converter.convert(condition, sortKey);

// Criteria を使って検索を実行
List<Customer> results = dao.findByCriteria(criteria);
```

ページング処理

10件ずつデータを取得する例:

- dao.findByCriteria(criteria, 開始インデックス, 取得件数)

```
CustomerCriteriaConverter converter
        = (CustomerCriteriaConverter) p.appctx.getBean(
                "CustomerCriteriaConverter");
DetachedCriteria criteria = converter.convert(condition, sortKey);
// 先頭の 10 件を取得
List<Customer> results = dao.findByCriteria(criteria, 0, 10);
// ...
// 次の 10 件を取得
results = dao.findByCriteria(criteria, 10, 10);
```

FinderContext

FinderContextを使ったページング処理の例：

```
// FinderContext インスタンスを作成
FinderContext<CustomerC> finderContext = new FinderContext<CustomerC>();
// Condition をセット
finderContext.setCondition(condition);
// ページサイズをセット( 0 をセットすると無制限となります)
finderContext.setPageSize(10);
// CriteriaConverter をセット
finderContext.setCriteriaConverter(
        (CustomerCriteriaConverter) p.appctx.getBean(
                "CustomerCriteriaConverter"));

// 先頭の 10 件を取得
List<Customer> results = dao.find(finderContext);

// 次の 10 件を取得
finderContext.next();
results = dao.find(finderContext);

// 最後のページのデータを取得
finderContext.last();
results = dao.find(finderContext);
```

Native SQL

SQLを使ったカスタマイズも可能です。

```
// SQL の組立
String query = "SELECT * FROM customer WHERE …";

// Session(DBコネクション)を取得
Session session = getCurrentSession();

// SQL の実行、及び結果を List で取得。
@SuppressWarnings("unchecked")
List<Customer> list = session.createSQLQuery(query)
        .addEntity(Customer.class).list();
```

Native SQL

(Hibernateを利用しない) JDBCプログラミングの例:

```
public class MyCustomerDao extends CustomerDao {

    /** {@inheritDoc} */
    @Override
    public Customer get(Integer pkey) {
        getCurrentSession().doWork(new Work() {
            public void execute(Connection connection) throws SQLException {
                CallableStatement st = null;
                ResultSet rs = null;
                try {
                    // ストアドプロシジャ呼び出し
                    st = connection.prepareCall("{?= CALL POWER(2, 3)}");
                    st.execute();
                    rs = st.getResultSet();
                    if (rs.next())  {
                      System.out.println(rs.getInt(1)); // 結果の出力。
                    }
                } finally {
                    DbUtils.closeQuietly(st);
                    DbUtils.closeQuietly(rs);
                }
            }
        });

        return get(pkey, true);
    }
}
```

ロック機構

悲観ロックと楽観ロック

- 悲観ロック
  - 更新のための参照をした時点でロックをかけます。
    - 更新画面を開いた時点でロックをかけます。
    - あるユーザーが更新画面を開いている間、別のユーザーは同一データの更新画面を開くことはできません。(閲覧は可能)
    - 楽観ロックよりロック時間が長いです。
- 楽観ロック
  - 更新データの保存時に「バージョン番号」をチェックします。
    - バージョン番号が変更されていなければ他のユーザからの編集はないと判断して保存します。
    - バージョン番号が変更されていれば他のユーザが編集したと判断され楽観ロックエラーとします。

一般的な悲観ロックの実装

- バッチ処理時の場合
  1. トランザクション開始
  2. データの取得

     SELECT * FROM customer WHERE customerid = 1000 **FOR UPDATE**
  3. データの更新
  4. Commit または Rollback
  5. トランザクションの終了(ロックの解除)

※手順2からトランザクションの終了まで customerid が1000のデータはロックされている。

Webアプリケーションでのトランザクション制御

- One transaction per request
  1. トランザクションを開始
  2. データの取得(SELECT * FROM customer WHERE customerid = 1000)
  3. トランザクションを終了
  4. 更新画面が表示される
  5. ユーザーが更新画面でデータを編集後、保存。
  6. トランザクションを開始
  7. データの更新 (UPDATE customer … WHERE customerid = 1000)
  8. Commit または Rollback
  9. トランザクションを終了

一般的な悲観ロックの実装

- Webアプリケーションの場合
  - Request 毎に一つのトランザクション
  - 更新処理を実現するのに複数のリクエストを実行
    - 更新画面表示 → データの編集 →データの保存
  - SELECT … FOR UPDATE は複数のトランザクションにまたがる更新処理には利用できない
- SELECT … FOR UPDATE以外の実装方法
  - 独自にメモリ上で管理する
  - データベースにロック情報を格納し、管理する

※いずれもロックの自動解除機構が必要

Wagbyの悲観ロック

- メモリ上での管理
- Jfclockobject テーブルでの管理
  - 「環境 > カスタマイズ > ロック情報をデータベースのテーブルに格納する」設定を有効にする
  - Jfclockobjectテーブルの構造

項目名	主キー	説明
modelname	○	ロック対象のモデル名（英語）
pkey	○	ロック対象データの主キーの値。複合キーの場合は"$"を区切り文字として値を連結する。
lockForAll	○	数値型。"1" が個別データのロック。"2" がモデル全体のロックを意味する。
userid		ロックを取得した(juserアカウントの)userid値。
usrename		ロックを取得した(juserアカウントの)username値。
sessionid		ロックを取得したユーザのセッションID値。

カスタマイズコードでの悲観ロック処理

データ取得時の悲観ロック

```
// 主キー1000のデータを取得(第二引数を true とすることで悲観ロックを同時に行う)
Customer customer = customerDao.get(1000, true);
```

その後データの保存と同時にロック解除

データ更新時の悲観ロック

- データ取得時は悲観ロックを行わず、更新時にのみロックを行う場合

```
customerDao.update(customer, true);
```

カスタマイズコードでの悲観ロック処理

- Service・Daoを利用しないパターン
  - LockUtilsクラスの利用
  - LockUtils#lock(E, DaoHelper<E, PK>)
    - 引数に指定した Entity をロックする
    - ロックに失敗した場合はExceptionが発生
  - LockUtils#release(E, DaoHelper<E, PK>)
    - 引数に指定した Entity のロックを解除
  - LockUtils#isLocked(E, DaoHelper<E, PK>)
    - 引数に指定したEntityがロックされているかを確認
    - ロックされていない場合はExceptionが発生

悲観ロックの自動解除

- 悲観ロック解除のタイミング
  - 保存時
  - キャンセル時(カスタマイズでのロック時は動作しない)
  - メニュー画面遷移時
  - ログオフ時
  - セッションタイムアウト時

Wagbyの悲観ロックとバッチ処理

- Wagbyアプリケーション外でのバッチ処理
  - jfclockobject テーブルでロック情報を管理するよう定義
  - バッチプログラム側での悲観ロックの配慮
    - データ更新・削除時はjfclockobjectテーブルをチェックし、ロックされていないことを確認の上、ロック情報の追加後、更新・削除を行う。
      - これを怠るとロストアップデート(更新内容の消失)が発生する可能性
    - 更新・削除処理の終了時はロック情報を削除する。
      - これを怠ると他のユーザーがデータを更新できなくなる。

楽観ロック

- Hibernateの楽観ロック機構を利用
  - Version管理用のカラムを用意
    - 型は long
    - 初回登録時は 0 が格納される。更新する毎に1増加。
    - 9223372036854775807 回まで更新可能(実質無制限)

```
<hibernate-mapping ...>
    <import class="Customer"/>
    <class name="Customer" table="customer">
        <id name="customerid_">
            <column name="customerid"/>
        </id>

        <!-- 楽観ロック用 version 管理項目 -->
        <version name="version_" column="version"
                 type="long" unsaved-value="negative"/>
```

楽観ロックとバッチ処理

- 悲観ロックと異なり、ロックの確認が楽になります。
  - 更新対象データの version 管理カラムをチェックするのみ
    - UPDATE customer SET …, version = version + 1
      WHERE customerid = '1000' and version = **X**;
    - (他のユーザがすでに編集し、) version が変更されていればデータの更新は行われない。更新結果件数が0件となる。
    - その場合は、更新をやり直すなど適切な処理を行う。

楽観ロックの注意点

- 他のユーザーと同じデータの編集画面を開いていても気がつかない。
- 他人が更新画面を開いている状態でも、データの削除が行えてしまう。
  - 悲観ロックの場合は悲観ロックエラーとなり削除できない。

Wagby R8 アーキテクチャ
2018年5月　第1.1版