

サンプルでみるSabaphyの概要

Sabaphy 0.5.3 向け 佐藤英人 (東京国際大学)

1. はじめに

SabaphyはWebアプリケーションの学習用ライブラリです。言語はPHP5を使用しています。本稿では、このSabaphyのイメージをつかんでいただくために小さなサンプルを紹介します。このサンプルは「アルバイトの登録と料金計算」のサンプルです。これは拙著の「Webアプリケーション入門」という教材の例題がベースになっています。詳しくはそちらを参照してください。サンプルプログラムの実行方法は、プログラムに同梱されているReadMe.txtをご覧ください。

このサンプルは、Webアプリケーションについて一通りの知識がある人を対象に、Sabaphyでの扱いを解説します。本稿の記述が難しいと思う人は、上に書いた教材を先にご覧ください。

【備考】Sabaphyライブラリ

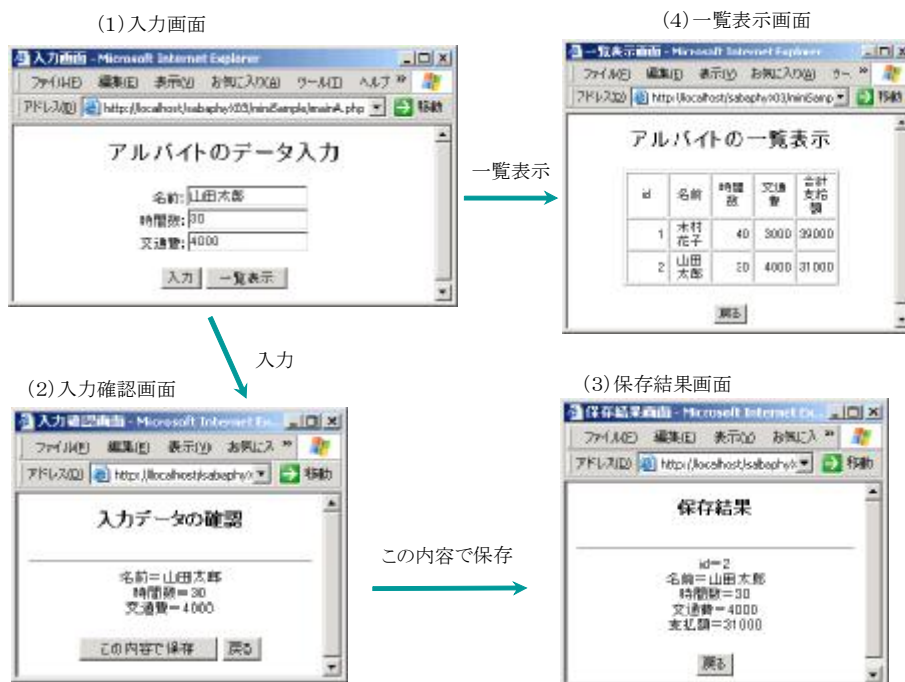
ここで使用するライブラリSabaphyは2つのクラス、Dispatcher (フレームワーク) と EntityManager (O/Rマッピングツール) が主たる構成要素です。分かりやすさのために、敢えて細かくコンポーネントに分けることをしていません。Dispatcher、EntityManagerはプログラムサイズが小さく、1つ1つのファイルで完結しています。ある程度プログラム知識がある人であれば、ライブラリ自身も読み解くことができると思います。トライしてみてください。これらの機能を絞り込んだDispatcher_Light (2KB, 70行程度)、EntityManager_Light (6KB, 170行程度) もannex0に同梱しています。教材サンプルのほとんど (Lect8-2までと本編のmainA.phpとmainB.php) は、これらでも動きます。先にこちらをご覧くださいになる方が分かりやすいでしょう。

2. サンプルの概要

(1) サンプルのイメージ

このサンプルは入力画面、入力確認画面、保存結果画面、一覧表示画面をもっています。

図1 サンプルの画面遷移



入力画面でデータを入力し「入力」ボタンを押すと、入力データはセッションデータとし

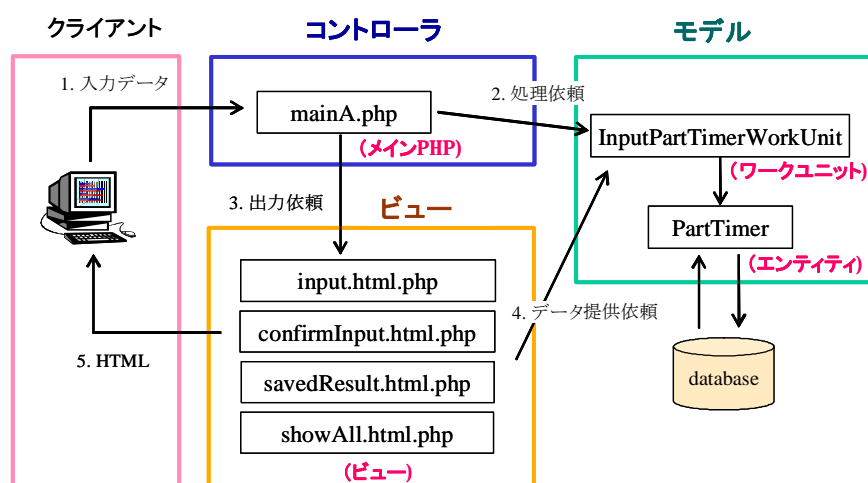
でキャッシュされ、入力確認画面が表示されます。ここで「この内容で保存」ボタンを押すとキャッシュされたデータを使ってアルバイトオブジェクトが生成され（DBに保存され）、保存結果画面が表示されます。

入力画面で「一覧表示」ボタンを押すとDBに蓄積された全アルバイトのデータが表示されます。

(2) アプリケーションの動作

図2は上のサンプルを実行するアプリケーションの構成図です。ここではクライアントからの入力を受け取るコントローラを1つにまとめるフロントコントローラ型を採用しています。このサンプルでは、フレームワークを利用しないコントローラがmainA.php、フレームワークを利用するものがmainB.php、DIコンテナとフレームワークを利用するものがmainC.phpです。どれも同じ動作のアプリケーションを記述しています。

図2 Sabaphyアプリケーションの構成



最初に以下のURLをブラウザのアドレス欄に入力し、mainA.phpを起動します。するとinput.html.phpというHTML生成プログラムが呼び出され、入力画面が表示されます。

URL: <http://localhost/miniSample/mainA.php>

入力画面でデータを入力して「入力」ボタンを押すと、再びmainA.phpが呼び出されます。mainA.phpは押されたボタンの名称（コマンド名）によって処理の振り分けを行います。「入力」のときは、ワークユニットInputArticleWorkUnitを新規に作り、そのプロパティ値として入力データをキャッシュします。ここでワークユニットとは、セッションの継続中存続する作業オブジェクトです。EJBのStateful SessionBeanに相当します。次いでmainA.phpは、confirmInput.html.phpというHTML生成プログラムを呼び出し、確認画面HTMLをクライアントに送り返します。

確認画面で「この内容で保存」ボタンが押されると、mainA.phpはワークユニットInputArticleWorkUnitを再現し、キャッシュされていたデータを使って、エンティティPartTimerのインスタンス（アルバイトオブジェクト）を生成します。エンティティはDBに永続化されるオブジェクトで、EJBのEntityBeanに相当します。次いでmainA.phpは、savedResult.html.phpというHTML生成プログラムを呼び出し、保存結果画面HTMLをクライアントに送り返します。

入力画面で「一覧表示」ボタンが押されると、mainA.phpは、showAll.html.phpというHTML生成プログラムを呼び出し、一覧表示画面HTMLをクライアントに送り返します。

3. アプリケーションの構成部品

(1) エンティティ

SabaphyはJavaの世界のHibernateあるいはEJB3のEntityManagerと同様のインタフェースでオブジェクトの永続化を行います。そこではエンティティも普通のオブジェクト（Java

例えばPOJO)として定義します。DBへの保存はEntityManagerというサービスオブジェクトに依頼して行います。更新、削除、検索も同様です。

<リスト1> エンティティ (miniSample/model/PartTimer.class.php)

```

1) <?php
2) // アルバイトクラス
3) class PartTimer {
4)     // 定数
5)     protected $hourlyWage = 1000;
6)     protected $taxRate = 10;
7)     // プロパティ
8)     public $id;
9)     public $name;
10)    public $hours;
11)    public $carfare;
12)    // 構造定義メソッド
13)    function keyName() { return "id"; } //主キー属性名
14)    // メソッド
15)    function init($name, $hours, $carfare) {
16)        $this->name = $name;
17)        $this->hours = $hours;
18)        $this->carfare = $carfare;
19)        getEntityManager()->save($this); //DBに保存を依頼
20)    }
21)    function changeHours($hours) {
22)        $this->hours = $hours;
23)        getEntityManager()->update($this); //DBに更新を依頼
24)    }
25)    function getTotalPay() {
26)        return $this->hours * $this->hourlyWage *
27)            (100 - $this->taxRate) / 100 + $this->carfare;
28)    }
29)    function delete() {
30)        getEntityManager()->delete($this); //DBに削除を依頼
31)    }
32) }
33) ?>

```

上のリスト1は、エンティティPartTimer (アルバイト) のクラス定義です。ここでは、5～11行で\$hourlyWage (時給)、\$taxRate (税率)、\$id (ID番号)、\$name (名前)、\$hours (勤務時間数)、\$carfare (交通費) という6つのプロパティを定義しています。

後の4つのプロパティはpublic (外部に公開) と宣言しています。Sabaphyの場合、public宣言されたプロパティだけが永続化の対象になります。

オブジェクトの永続化には、「O/Rマッピング情報」すなわち、オブジェクトをリレーショナルDBの行に対応付ける情報が必要です。Sabaphyでは13行目のようにkeyName()という主キー属性名を返すメソッドをクラスに定義することでマッピング情報を与えます。テーブル名にはクラス名 (今の場合、PartTimer)、列名にはプロパティ名 (今の場合、id, name, hours, carfare) がそのまま使われます。

15～20行のinit()は、オブジェクトの生成直後に呼び出される初期化のメソッドです。ここでは、引数として与えられたプロパティ値を自身のプロパティにセットし、DBに保存します。ここではプロパティidの値をセットしていません。SQLiteは、主キー値が与えられないとき、自動で連番の主キー値を与えてくれるので、それを利用しています。

19行目がDBへの保存の指示で、上で述べたEntityManagerのサービスを利用していま

す。ここの`getEntityManager()`は`EntityManager`オブジェクトを取得するグローバル関数です。取得した`EntityManager`に依頼する形で、エンティティのDBへの保存を行います。

21～24行の`changeHours()`は、エンティティのプロパティ値の変更を行うメソッドの例です。勤務時間を引数で与えられたものに変更します。変更後、上と同様に`EntityManager`に依頼して、変更をDBに通知します。

25～28行の`getTotalPay()`は、支払金額を計算して返すメソッドです。このようにエンティティが参照できるものを使って実行できるビジネスルールは、そのエンティティのメソッドとして定義します。

29～32行の`delete()`は、エンティティの削除メソッドです。新規生成や更新の場合と同様に`EntityManager`に依頼して、削除をDBに通知します。このサンプルでは、`changeHours()`と`delete()`を使っていません。これらは`EntityManager`の使い方の説明用に追加したものです。

(2) ワークユニット

ワークユニットは先に述べたようにセッションの継続中存続する作業オブジェクトです。ここにはコントローラやビューからの依頼に応えるアクションメソッドを置きます。必要に応じてセッションデータとして残すべきデータやオブジェクトをプロパティ値にセットします。

<リスト2> ワークユニット (`miniSample/InputPartTimerWorkUnit.class.php`)

```

1) <?php
2) class InputPartTimerWorkUnit {
3)     public $name;
4)     public $hours;
5)     public $carfare;
6)     public $parttimer;
7)
8)     // アクションメソッド
9)     function receiveInput() {
10)         //入力データをセッションデータとしてキャッシュ
11)         $this->name = $_REQUEST['name'];
12)         $this->hours = $_REQUEST['hours'];
13)         $this->carfare = $_REQUEST['carfare'];
14)         $this->parttimer = null; //念のため初期化
15)     }
16)     function fixInput() {
17)         //キャッシュデータを使ってオブジェクトを生成
18)         if (!$this->name) {
19)             die("データが与えられていません。多分ボタンの2度押しです");
20)         }
21)         $parttimer = new PartTimer();
22)         $parttimer->init($this->name, $this->hours, $this->carfare);
23)         $this->parttimer = $parttimer; //ビューからの参照用にキャッシュ
24)         $this->name = null; //念のため初期化 (ボタンの2度押し対策)
25)     }
26) }
27) ?>

```

9～15行の`receiveInput()`は、「入力」ボタンが押されたときにコントローラから呼び出されるメソッドです。ここでは次回のアクセス時のために入力データをワークユニットのプロパティに格納します。ここで利用している`$_REQUEST`は、GETまたはPOSTでブラウザから送られてきたデータを格納しているPHP標準のグローバル連想配列です。

16～25行の`fixInput()`は、「この内容で保存」ボタンが押されたときに起動されるメソッドです。`PartTimer`オブジェクトを生成し、保存されていたプロパティ値を使って初

期化します。23行目で、他からの問い合わせに答えるために、最後に生成された `PartTimer` オブジェクトを `parttimer` というプロパティに格納します。

次節で説明する、フレームワークを使うとき、ワークユニットはセッションに登録されていればそれを再現し、登録されていないければ新規に生成します。このため、メソッドが呼ばれたとき、前のデータが残っている可能性があります。14行目と24行目のプロパティの初期化は、そのための対策で、ワークユニットの状態管理のためのものです。

(3) ビュー

SabaphyではHTML出力に普通のPHPファイルを使用します。下のリスト3は、上のワークユニットの `fixInput()` でエンティティをDBに保存した後に、呼び出されるもので、保存結果画面を表示するビュープログラムです。

<リスト3> HTML生成PHP (`miniSample/view/savedResult.html.php`)

```

1) <?
2)     $parttimer = $workunit->parttimer;
3) ?>
4) <html>
5) <head><title>保存結果画面</title></head>
6) <body>
7) <center>
8) <h3>保存結果</h3>
9) <hr>
10) id=<?=html special chars($parttimer->id) ?><br>
11) 名前=<?=html special chars($parttimer->name) ?><br>
12) 時間数=<?=html special chars($parttimer->hours) ?><br>
13) 交通費=<?=html special chars($parttimer->carfare) ?><br>
14) 支払額=<?=html special chars($parttimer->getTotalPay()) ?><br>
15) <form method="POST">
16)     <input type="submit" name="_COMMAND" value="戻る">
17) </form>
18) </center>
19) </body>
20) </html>

```

Sabaphyのビューでは、直前に使用したワークユニットオブジェクトを `$workunit` という変数で参照できます。2行目がその利用で、ワークユニットのプロパティ `parttimer` の値を取得して、`$parttimer` という変数に代入しています。これは前項で述べた最後に生成した `PartTimer` オブジェクトです。これを利用して、10～14行でアルバイトの属性や導出属性（計算により求まる属性）の値を表示しています。

このようにSabaphyのビュープログラムは、ワークユニットから必要な情報を取得し、出力するだけのものを原則としています。

15～17行は「戻る」ボタンの実装です。Sabaphyでは、次に行う処理を指定するボタンの名前は `"_COMMAND"` という名前に固定しています。16行目のボタンでは、その値（ボタンの表示ラベル）を「戻る」にしています。これが次に実行すべきコマンドの名前になります。15行目の `form` タグでは `action` プロパティを指定していません。この結果、16行目のボタンが押されると、このビューを呼び出したプログラム（次節で述べるコントローラプログラム）に `['_COMMAND = 戻る]` というデータが送られます。

4. コントローラプログラム

(1) コントローラ

前節で説明した構成部品をまとめ、役割に応じた仕事を配分するのがコントローラプログラムです。フレームワークを使わないとき、コントローラは普通のPHPプログラムとして作成します。次のリスト4の `mainA.php` はその例です。

<リスト4> フレームワークを使用しないコントローラ (miniSample/mainA.php)

```

1) <?php
2) // (1) 使用するプログラムファイルの指定
3) require_once '../sabaphy/ExceptionHandler.class.php';
4) require_once '../sabaphy/EntityManager.class.php';
5) require_once 'model/InputPartTimerWorkUnit.class.php';
6) require_once 'model/PartTimer.class.php';
7)
8) // (2) セッションの開始と初期設定
9) session_start();
10) $eh = new ExceptionHandler();
11) $em = new EntityManager('db/mydb');
12) $em->beginTransaction();
13)
14) // (3) 処理の振り分け
15) $cmd = null;
16) if (isset($_REQUEST['_COMMAND'])) $cmd = $_REQUEST['_COMMAND'];
17) if (($cmd == null) || ($cmd == '戻る')) {
18)     require 'view/input.html.php';
19) } elseif ($cmd == '入力') {
20)     $workunit = new InputPartTimerWorkUnit();
21)     $workunit->receiveInput();
22)     $_SESSION["InputPartTimerWorkUnit"] = $workunit;
23)     require 'view/confirm.html.php';
24) } elseif ($cmd == 'この内容で保存') {
25)     $workunit = $_SESSION["InputPartTimerWorkUnit"];
26)     $workunit->fixInput();
27)     require 'view/savedResult.html.php';
28) } elseif ($cmd == '一覧表示') {
29)     require 'view/showAll.html.php';
30) }
31)
32) // (4) DB変更の確定
33) $em->endTransaction();
34) $em->close();
35) ?>

```

3～6行は使用するプログラムファイルの指定です。3行目がSabaphyのExceptionHandlerのクラスファイル、4行目がSabaphyのEntityManagerのクラスファイル、5～6行が上で紹介したワークユニットとエンティティのクラスファイルです。

(2) 初期設定

9行目でセッションを開始または再現させます。これはPHPの機能です。10行目でExceptionHandlerのインスタンスを作り、\$ehという変数に代入しています。\$ehはここでは使われていません。この行は、単に「new ExceptionHandler();」でも構いません。ExceptionHandlerは、エラーが発生したとき、エラー画面の生成を行うコンポーネントです。ExceptionHandlerを生成すると、その中でPHPの標準のエラーハンドラをSabaphyのエラーハンドラに置き換えます。これでエラーが発生したとき、スタックトレースなどが表示できるようになります。

11行目でEntityManagerのインスタンスを作り、\$emという変数に代入しています。引数で与えている'db/mydb'はDBファイルへのパスです。Sabaphyでは、SQLiteというPHP5に標準で組み込まれている簡易DBMSを利用します。MySQLを利用することもできます。12行目でトランザクションを開始しています。もっときめ細かくトランザクション管理をすることもできますが、ここでは簡単化のためにアプリケーションの開始時点でトランザク

ションを開始し、エラーなく終わりまで来たときにトランザクションを終了させる方式を採用しています。

これで永続化のための設定は終わりです。以降は`getEntityManager()`というグローバル関数を呼び出すことで、Sabaphyの永続化サービスを利用することができます。先のリスト1に登場した`getEntityManager()`がそれです。`getEntityManager()`で取得される`EntityManager`は、11行目で生成された`$em`と同じものです。

(3) 処理の振り分け

ビューのところで述べたように、Sabaphyでは、"`_COMMAND`"という名前の入力値(`$_REQUEST['_COMMAND']`)に次に実行すべきコマンドの名前(入力コマンド名)が入っています。15~30行は、この入力コマンドに応じた処理の振り分けです。

`$_REQUEST['_COMMAND']`に値がセットされていないケースは、最初にこの`mainA.php`が呼び出されたときです。このとき18行目の「`require 'input.html.php'`」を実行します。これで入力画面が表示されます。「戻る」ボタンが押されたときも同様です。

(4) ワークユニットの新規利用

20~23行はコマンド「入力」に対応するアクションの実行です。20行目で先のワークユニット`InputPartTimerWorkUnit`のインスタンスを生成し、21行目でそのアクションメソッド`receiveInput()`を実行します。ここで入力データがワークユニットのプロパティにキャッシュされます。

22行目で、このワークユニットをセッション継続中いつでも再現できるように、セッションに登録します。`$_SESSION`はセッションデータを保持するPHP標準の連想配列です。ここにワークユニットのクラス名をキーとして、ワークユニットオブジェクトを登録します。

23行目で`confirm.html.php`にHTMLの出力を依頼します。これは確認画面のHTMLを生成するビュープログラムです。これで確認画面がクライアントのブラウザ上に表示されます。

(5) ワークユニットの再現

25~27行はコマンド「この内容で保存」に対応するアクションの実行です。25行目で`$_SESSION`からワークユニットクラス名'`InputPartTimerWorkUnit`'で登録したワークユニットを取り出し、`$workunit`に代入しています。これで前回22行目でセッションに登録したワークユニットが再現されます。

26行目でこのワークユニットのアクションメソッド`fixInput()`を実行します。ここでキャッシュしておいた入力データを使って、アルバイトオブジェクトを生成し、DBに保存します。次いで27行目で保存結果画面を表示します。

(6) ビューの表示

29行目はコマンド「一覧表示」に対応する処理です。ここでは`showAll.html.php`を呼び出し、一覧表示画面を生成しています。

(7) DB変更の確定

33行目でトランザクションを終了します。これでDBに与えた変更が確定します。途中でエラーがあったときは、ここに到達しません。DBは変更前の状態のままということになります。34行目で`EntityManager`を閉じます。これでDBへの接続が切断されます。

5. フレームワークの利用

(1) コマンド情報一覧

前節のリスト4の「処理の振り分け」は、一見複雑ですが、やっていることは単純で簡単に定型化できます。ここで行っている作業を整理すると、下図のようにまとめることができます。このテーブルは送られてきたコマンドに応じて、ワークユニットで何をし、どのビューを使って出力するかをまとめたものです。このようなテーブルをSabaphyではコマンド情報一覧と呼んでいます。

図3 コマンド情報一覧

コマンド名	ワークユニット		ビュー
	クラス名	アクション	出力生成PHP
null	-	-	input.html.php
戻る	-	-	input.html.php
入力	InputPartTimerWorkUnit	receiveInput()	confirm.html.php
この内容で保存	InputPartTimerWorkUnit	fixInput()	savedResult.html.php
一覧表示	-	-	showAll.html.php

この情報が得られれば、処理の振り分けをプログラムとして書くのではなく、汎用的に行うコントローラを作ることができます。それがSabaphyのアプリケーションフレームワークです。そこでは処理の振り分け作業をDispatcherと呼ばれるコンポーネントに依頼する形で行います。

(2) ディスパッチャの利用

下のリスト5は、Dispatcherを使用したコントローラmainB.phpです。やっていることは先のmainA.phpと同じです。

<リスト5> ディスパッチャを利用したコントローラ (miniSample/mainB.php)

```

1) <?php
2) // (1) 使用するプログラムファイルの指定
3) require_once '../sabaphy/ExceptionHandler.class.php';
4) require_once '../sabaphy/EntityManager.class.php';
5) require_once '../sabaphy/Dispatcher.class.php';
6) require_once 'model/InputPartTimerWorkUnit.class.php';
7) require_once 'model/PartTimer.class.php';
8)
9) // (2) セッションの開始と初期設定
10) session_start();
11) $ex = new ExceptionHandler();
12) $em = new EntityManager('db/mydb');
13)
14) // (3) ディスパッチャの生成
15) $d = new Dispatcher($ex, $em);
16)
17) // (4) コマンド情報の設定
18) $d->addCommand(' null', array('view'=>'view/input.html.php'));
19) $d->addCommand(' 戻る', array('forward'=>' null'));
20) $d->addCommand(' 入力', array(
21)     'action'=>'InputPartTimerWorkUnit::receiveInput',
22)     'view'=>'view/confirm.html.php'));
23) $d->addCommand(' この内容で保存', array(
24)     'action'=>'InputPartTimerWorkUnit::fixInput',
25)     'view'=>'view/savedResult.html.php'));
26) $d->addCommand(' 一覧表示', array('view'=>'view/showAll.html.php'));
27)
28) // (5) コマンドの実行
29) $d->execute();
30) ?>

```

1 2行目までは、先のリスト4のコントローラとほとんど同じです。5行目に、ここで新たに使用するライブラリファイルDispatcher.phpの指定が付け加わっています。

1 5行目はDispatcherオブジェクトの生成です。上のExceptionHandlerとEntityManagerのインスタンスを引数で注入しています。ここではDIコンテナを使いませんが、これはコンストラクタインジェクションと呼ばれる依存性の注入です。DI手法をここで利用してい

ます。このように生成時に使用する部品を指定することで、簡単に別のExceptionHandlerクラスやEntityManagerクラスのコンポーネントに差し替えることができます。実際、SabaphyのEntityManagerには、SQLiteを使うEntityManagerとMySQLを使うEntityManager_MySQLの2つのものがあります。そのどちらを使うかを、この引数で指定することができます。

DispatcherにExceptionHandlerを組み込むと、プログラムで発生する例外を捕捉できるようになります。例外が発生したとき、デフォルトではSabaphy組み込みのエラー画面が表示されます。カスタムエラーメッセージやカスタムエラー画面をこのExceptionHandlerに登録しておく、例外の種類に応じたメッセージや画面を表示することができます。

DispatcherにEntityManagerを組み込むと、コマンドの実行時にトランザクションを開始し、その実行が正常に終わったときにトランザクションを終了して、DB変更を確定させることができます。

18～26行では、このディスパッチャにデータを与える形で、図3のコマンド情報を与えています。18～26行と図3を見比べてください。addCommand()の引数は、図3のコマンド一覧表の各行のデータを連想配列で書き直したただけのものであることがお分かりいただけると思います。この種のコントロールデータ（ワークフローデータ）はXMLで書くのが今日では一般的ですが、Sabaphyでは新しい概念を持ち出さずすませるために、PHPの連想配列を利用して記述しています。PHPのようなインタプリタ言語では、プログラムもデータもメンテナンスの手間は変わらないからです。

29行目でコマンド情報をセットしたディスパッチャにexecute()を指示しています。ここで入力データから要求コマンドを識別し、対応するコマンド情報を見つけ、その指示に従って、処理を実行します。具体的には、トランザクションの開始、ワークユニットの生成あるいは再現、アクションメソッドの起動、ビューの表示、トランザクションの終了です。

6. DI コンテナとオートローダの利用

(1) DI コンテナとSabaphy

Sabaphyのコンポーネントはそれぞれ独立に動くように作られています。分かりやすさを優先させて、細かくサブコンポーネントに分けることをやっていません。また、PHPの場合、グローバル関数が安全に使えますので、Javaの場合と違って、永続オブジェクトにEntityManagerを注入する必要がありません。

このため、Sabaphyでは必ずしもDIコンテナを使う必要はありません。しかし、以下の2つのメリットを考えて、SbContainerというDIコンテナを用意しています。

- 1) 個々のライブラリファイルの位置を隠せる。
- 2) ライブラリクラスの初期化の実装方式を隠蔽できる。

これらは、Sabaphyのバージョンアップでライブラリの構成が変わったとしても、ユーザープログラムを変更せずに済ませられる可能性を高くするものです。DIコンテナの使用に対する抵抗感がなければ、その使用をお勧めします。

(2) オートローダ

これまでメインプログラムの先頭で使用するプログラムファイルを指定しました。ここではこれを簡略化するために、Sabaphy組み込みのAutoLoaderを使用します。AutoLoaderはクラスが使用されたとき、対応するクラスファイルを予め指定された検索パスから検索して読み込むものです。デフォルトではSabaphyのライブラリとメインプログラムが属するフォルダが検索パスに組み込まれています。

(3) AutoLoaderとSbContainerの利用

下のリスト6は、AutoLoaderとSbContainerを使用したコントローラです。やっていることは先のmainA.phpやmainB.phpと同じです。

<リスト6> DIコンテナを利用したコントローラ (miniSample/mainC.php)

```
1) <?php
2) // (1) オートローダの設定
3) require_once '../sabaphy/AutoLoader.class.php';
4) AutoLoader::setup('model');
5)
6) // (2) コンテナの初期設定
7) $cnt = new SbContainer();
8) $cnt->createEntityManager('EntityManager', 'db/mydb');
9)
10) // (3) ディスパッチャの取得
11) $d = $cnt->getDispatcher();
12)
13) // (4) コマンド情報の設定
14) $d->addCommand('null', array('view'=>'view/input.html.php'));
15) $d->addCommand('戻る', array('forward'=>'null'));
16) $d->addCommand('入力', array(
17)     'action'=>'InputPartTimerWorkUnit::receiveInput',
18)     'view'=>'view/confirm.html.php'));
19) $d->addCommand('この内容で保存', array(
20)     'action'=>'InputPartTimerWorkUnit::fixInput',
21)     'view'=>'view/savedResult.html.php'));
22) $d->addCommand('一覧表示', array(
23)     'view'=>'view/showAll.html.php'));
24)
25) // (5) コマンドの実行
26) $d->execute();
27) ?>
```

3行目はAutoLoaderの読み込みの指定です。6行目で追加する検索パスとして「model」フォルダを指定して、AutoLoaderを初期設定します。これで個々のクラスについてrequire_once文を書かなくても必要なクラスファイルが自動的に読み込まれるようになります。

7行目でSbContainerのインスタンスを生成します。先のリスト5 (mainB.php) の10～11行に相当することがここで実行されます。8行目でコンテナに依頼して、EntityManagerの生成を行っています。これは先のリスト5の12行目に相当します。11行目でコンテナからディスパッチャを取得しています。これは先のリスト5の15行目に相当します。リスト6の14行目以降は、先のリスト5と同じです。

このリスト6をよく見ていただくと分かるように、SbContainer以外の他のコンポーネントとして何があり、それがどこにあるかという情報はどこにも書かれていません。つまり、DIコンテナを使っておけば、ライブラリの内部構成が変わったとしても、利用者プログラムを修正する必要がなくなります。

7. おわりに

本稿では、ミニサンプルを用いてSabaphyの概要を説明しました。ここに登場した概念は、今日のWebアプリケーションで標準的なものがほとんどです。Webアプリケーション経験者の方はこれを見ただけで、Sabaphyが何をどのようにやるのかだいたいお分かりいただけたのではないかと思います。

Sabaphyの特徴は、初心者の学習向けに、ライブラリコンポーネントの機能を必要最小限に絞り込み単純化したことにあります。この単純化が適切なものであるかどうか、今後見極めて行く必要があるでしょう。皆さんのご批判・ご助言が得られれば幸いです。

EJBをはじめ一般のフレームワーク・コンテナの世界でも、これまでの方向が複雑でありすぎたという反省が広がり、EoD (Ease of Development) が業界の標語になってきています。それを目指すひとつのアプローチとして、このSabaphyを発展させていきたいと思っ

ています。