An Introduction to the Framework WebMVC

Quick Reference Guide

CONTENTS

1. The PHP WebMVC framework

1.1 Software framework

A *software framework* is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software.
A software framework provides a standard way to build and deploy applications. It is a universal, reusable software environment that provides particular functionality as part of a larger software platform to facilitate development of software applications, products and solutions. Software frameworks may include support programs, compilers, code libraries, tool sets, and application programming interfaces (APIs) that bring together all the different components to enable development of a project or system.

You can find this definition of software and some distinguishing features of software frameworks here: https://en.wikipedia.org/wiki/Software_framework

1.2 What is WebMVC?

WebMVC is a powerful PHP framework for web application developers and designers. The main goals of WebMVC are:

- Simplify your code and provide facilities that are useful for the rapid prototyping of a WEB application;
- Organize an application in subsystems using the Model View Controller (MVC) architecture to cope with the complexity of big projects;
- Improve the collaboration process among people having different skills (e.g. GUI, PHP and Database designers and developers);
- Use standard web technologies without introducing new ad hoc syntax.
- Provide generalized software components to implements some recurring problems in web development.
- Provide an ORM to interact with MySQL database.

The PHP framework WebMVC is a software an open source project made by Rosario Carvello. You can find the code here:

https://github.com/rcarvello/webmvcframework/wiki

and several examples ready to use here

https://www.webmvcframework.com

1.3 Top level architecture for the execution of WebMVC controllers

Preconditions

In order to develop using WebMVC framework you need:

- Operating System: Linux, Mac or Windows
- Server: Apache web server with mod_rewrite enabled
- Database: MySql (from 5.0 to the latest version)
- Programming language: PHP (5.3+) with DOM and mysqli extensions

Installation

To install the framework:

1. download it from Github
2. create a project in the root folder of your web server
3. import all the directories downloaded from Github into the project folder
4. modify the following lines of config/application.config.php" according to your db and web server

```
/**
 *  MySQL User
 */
define("DBUSER","PUT_YOUR_USERNAME");

/**
 * MySQL Password
 */
define("DBPASSWORD","PUT_YOUR_PASSWORD");

/**
 *  MySQL Database
 */
define("DBNAME","PUT_YOUR_DB_NAME");

/**
 *  MySQL Port
 */
define('DBPORT', '3306');

/**
 * Defines a constant for site URL
 * @note without the ending slash
 * @example: http://localhost/myproject
 */
define("SITEURL","http://PUT_YOUR_HOST/PUT_YOUR_WEB_FOLDER");
```

The fig. 1.1 shows a deployment diagram with the main software components necessary to run WebMVC applications. The diagram also shows the control flow triggered by an http request toward a server that will render a response. With this software architecture, two possible execution scenarios are possible. The first control flow arises from a usual http request aiming at the execution of a php file:

1. The client browser forwards an http request.
1.a. The web server receives the request and retrieves the php file from the web folder.
2. The web server invokes the execution of a PHP interpreter passing it the retrieved file to run. Eventually, the php file interacts with MySQL to populate the HTML page that must be sent back to the client; otherwise, the flow continues to the point 4.
3. MySQL retrieves data from one or more DB tables.
4. The HTML page is created.
5. The web server returns the response to the client.

The second scenario comes into play when the request aims at the execution of a php controller stored in the area of the host web folder where the WebMVC framework resides. When the web server do not match the file name appearing in the URL with a file name in the host web folder, it calls the rewrite engine that applies the rewriting rules established by WebMVC (cfr. Section 2 and section 5) to transform an URL string in a file name that represents a php controller. The flow of control is similar to that reported above where you have to substitute the points 1.a with the point 1.b:

1.b. The web server receives the request and retrieves a php controller in the portion of the host web folder devoted to WebMVC.

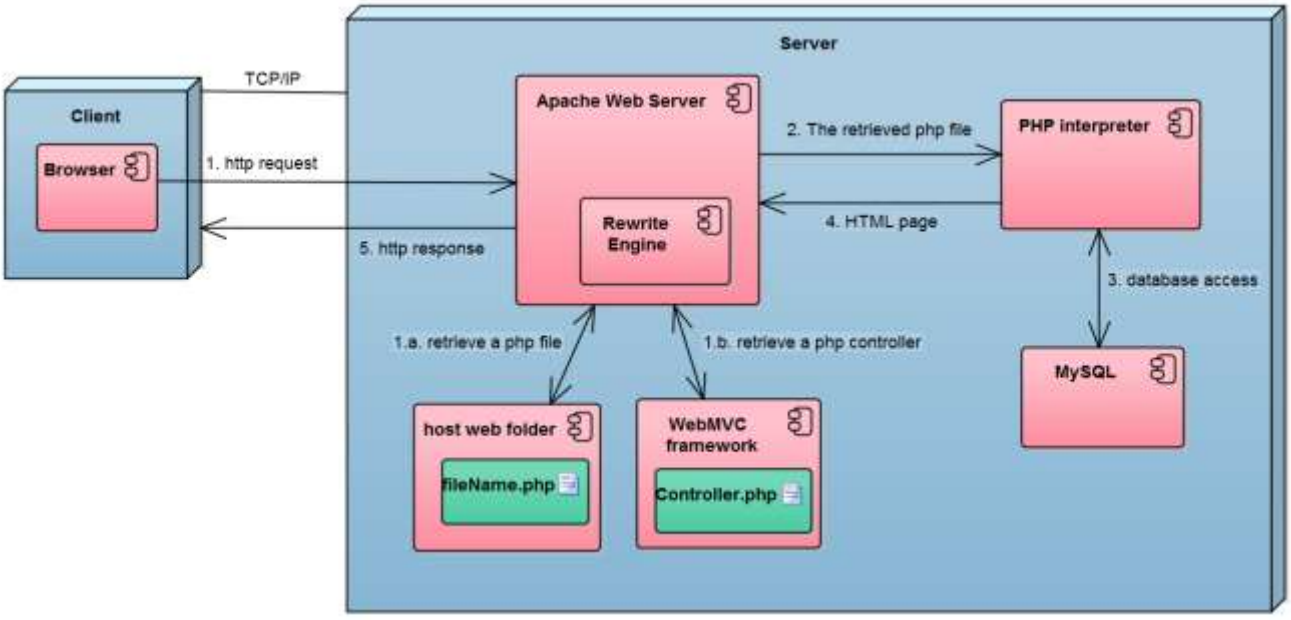


Fig. 1.1. The architecture of software components necessary for the execution of WebMVC applications.

2. Running a controller

Coding and running your first example is simple. To create a custom controller `HelloWorld` create a `HelloWorld.php` file in the controllers directory.

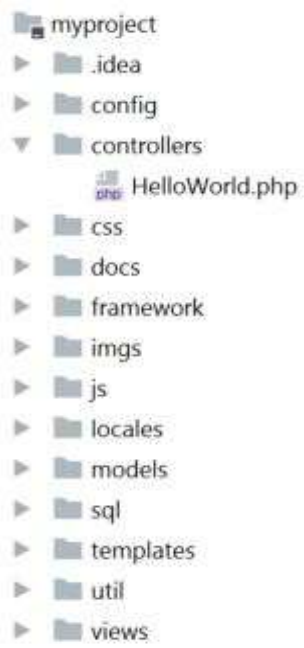


Fig. 2.1. The directory structure of WebMVC.

Then write a class called `HelloWorld` that extends the framework\Controller class.

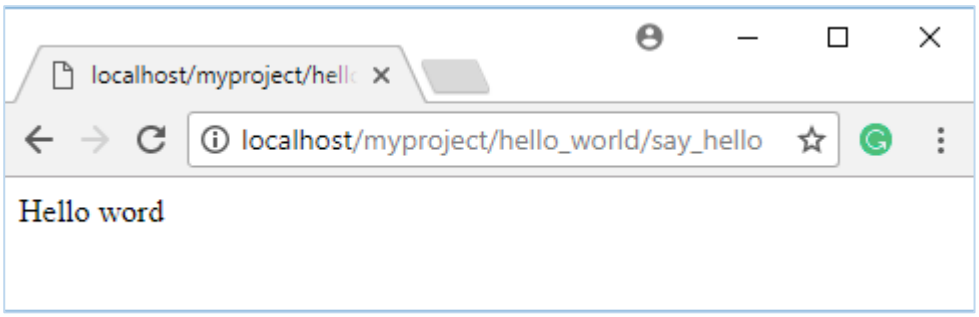
```
// Example 2.1. Your first controller

<?php

namespace controllers;

use framework\Controller;

class HelloWorld extends Controller
{

public function sayHello()
    {
        echo "Hello World";
    }
}
```

As you can see from the code, to create a controller we need to:

- use the class `framework\Controller`
- extend it with by writing the subclass `HelloWorld`

In `HelloWorld` it is defined a method `sayHello` that displays a message. In general, a controller has the responsibility to handle the logic and the control flow of a software application. Within WebMVC, frequently invoked methods are defined in the class `Controller` located into the directory framework, and that's why HelloWorld extends `Controller`. Even if in our first example any `Controller` method is invoked we shall see, in the following, how a subclass of `Controller` can take advantage of its methods.
At this point, the only thing we have to understand is how to instantiate a controller and execute the method. To do this, open your favorite web browser and type the following address:

```
http://localhost/myproject/hello_world/say_hello
```



You should see:

Hello World

Congratulations, you have executed your first example using WebMVC!

But how exactly it is translated the URL into a controller call? The URL you wrote calls the `sayHello` method of a `HelloWorld` class in the `controllers` directory. First, WebMVC automatically changes the syntax from lower case and underscores as follows:

```
hello_world -> HelloWorld          // the class name is transformed in PascalCase notation

say_hello -> sayHello              // the method name is transformed in camelCase notation
```

Then, after this transformation, the `sayHello` method of the `HelloWorld` controller is called. In general, to run a controller method you should use the following syntax:

```
<your server domain>/<controller name>/<method name>
```

This schema can be used when you have a single project to manage. In the case of two or more project existing within your web server root, you should use the following syntax:

```
<your server domain>/<project name>/<controller name>/<method name>
```

WebMVC and OOP programming

WebMVC requires that you create a controller that must extend the `framework\Controller` class. Then, just by adding public methods inside it, you will be able to implement the necessary functionalities. The only knowledge you need is about OOP programming.
The next example shows some concept regarding the interaction between WebMVC and OOP programming. Specifically, it is about the parameters and visibility of methods in a Controller. Let's add two new methods to the HelloWorld class:

`sayHelloMessage($message)` that is public and accepts a parameter `$message`, and

`cantSayHello()` that is a protected method.

```
// Example 2.2. Calling controller's methods

<?php

namespace controllers;

use framework\Controller;

class HelloWorld extends Controller
{

    public function sayHello()
    {
        echo "Hello world";
    }

    public function sayHelloMessage($message)
    {
        echo "Hello $message";
    }

    protected function cantSayHello()
    {
        echo "This method cannot be called from Url";
    }
}
```

Then type the following address:

`http://localhost/hello_world/say_hello_message/Mark`

`http://localhost/hello_world/say_hello_message/John`

Your output will be:

Hello Mark
Hello John

You can note we request the execution of a method `sayHelloMessage` specifying its parameter by typing its value into the URL by using a slash after the requested method

name. This is the rule for passing one or multiple parameters to a method: simply specify the corresponding values into URL and separate them with slashes. Take care of passing the exact numbers of values that a method requires as input parameters.
If you try to type:

`http://localhost/hello_world/say_hello_message/Mark/John` or

`http://localhost/hello_world/cant_say_hello`

in both cases, you will obtain an exception. In the first, you have inserted wrong numbers of parameters for the method `sayHelloMessage`, while in the second you have no access to the method `cantSayHello` because it is not public.

Summary

The examples 2.1.and 2.2 show how simple is to start coding with WebMVC and PHP programming. Just design and implement your application writing controller classes and public methods, and the framework will execute them as common HTTP requests. The URL notation used by WebMVC requires typing the HTTP request writing in lower case the names of controllers and methods, by separating them with a slash. It also requires an underscore for separating the occurrence of composite names. Therefore, you don't need to configure the execution of a particular controller, but you just use the URL notation proposed by WebMVC. This simplicity derives from the convention over configuration (cit.) approach that the framework uses for object instantiation in order to avoid tedious operations of configuration.

3. View

3.1 Static design

A view has the responsibility to organize and show data in graphical structures. With WebMVC, a `framework\View` class manages this responsibility operating on a template containing HTML code. To write an application that uses a static HTML code, you can:

- create a *template file* containing the HTML of the page that you want to show
- use the `framework\View` class provided by WebMVC to manage the template
- define a controller that manages the logic and the control flow

In WebMVC, the controller is the only entity that allows you to instantiate and run code starting from an HTTP request. Therefore, if you need to show an HTML page, you must

create a controller that: a) uses an instance of `framework/View` that manages the HTML file, and b) provides the output.

We can write an example that shows an HTML file, for example a Home page, into the browser following the steps described above. To do this, simply create the file `templates\home.html.tpl` (note that the name `home` is in lowercase) containing the HTML of the web page and put it in the directory `templates`:

```
// example 3.1.a: the template home.html.tpl

<!DOCTYPE html>
<html>
  <head>
    <title>Site home page</title>
  </head>
  <body>
    <p>Welcome to the site Home Page</p>
  </body>
</html>
```

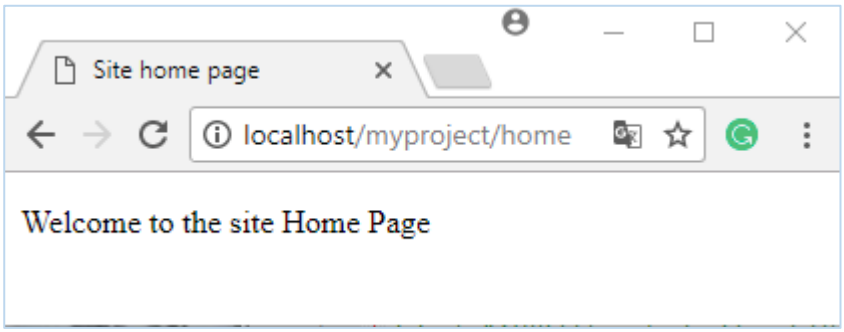
The file must have a `.tpl` extension in order to be accepted by WebMVC. Then, create the Home controller `Home.php` and put it in the directory `controllers` (pay attention to the comments):

```
// example 3.1.b: the controller Home.php

<?php

namespace controllers;

use framework\Controller;
use framework\View;

class Home extends Controller
{
    /**
     * Home constructor.
     * @override parent constructor
     */
    public function __construct()
    {
      /**
       * A reference to the file: templates/home.html.tpl
       * @note Do not to specify the file extension ".html.tpl".
       */
        $tplName = "home";


      /**
       * Set the view variable with a new object of type framework\View.
       * @note: We create the View object passing reference to the template home.
       */
        $this->view = new View($tplName);

      /**
```

```
        * The parent class Controller handle the necessary operations to print the
        * output. First, it uses the created View object to load the file containing
        * the template, then it renders the template in the browser.
        */

         parent::__construct($this->view);
    }

}
```

Finally, run the controller by typing the following address into your web browser:

`http://localhost/myproject/home`



You should see the Site home page with the message

Welcome to the site Home Page

In this example we invoke the controller `Home` without passing any method. The flow proceeds invoking its constructor that: 1) declares the template to use, b) passes the template to the `framework\View` class responsible for the template management, and c) invokes the constructor of the parent class (`framework\Controller`) for the template rendering into your browser.

Summary

The example 3.1 highlights the basic flow of control for the execution of a program within WebMVC where a controller and a view cooperate to do a job. It is analogous to the control flow of a command interpreter that:

1. **Accept a command** (WebMVC accepts an HTTP request and calls the corresponding controller to handle the request)
2. **Execute the command** (the controller executes the command using `framework\View` for the management of a template)

3. **Print the output** (the controller rends the HTML into your browser)

This flow will be reviewed in section 4 after the introduction of the model component.

There are circumstances where we need to introduce a new view defined by the developer because the `framework\View` class alone is no longer enough. This is the case of dynamic views for the dynamic content management that change the initial page content with data coming from sources such as keyboard or database.

3.2 Dynamic content

The code described in the previous example shows in the browser the static data contained in the file `home.html.tpl`. Now, consider the typical situation where you have to manage data dynamically in the view. WebMVC let you design a view class, as an extension of `framework\View`, that will be responsible for managing the corresponding template.

The concept of placehoder

In the following HTML code we augment the previous version of `home.html.tpl`. Now, together with the static content, we want to show `'Welcome {PersonName}'` where `{PersonName}` can assume different values. With `{PersonName}` we declare a *placeholder*, a string delimited by braces and located somewhere in the template, that we want to substitute with another string. Here it is the code of the new `home.html.tpl` in which we wish to replace the placeholder `{PersonName}` with the name of a person.

```
// example 3.2.a. Inserting a placeholder in the template home.html.tpl

<!DOCTYPE html>
<html>
  <head>
    <title>Site home page</title>
  </head>
  <body>
    <p>Welcome to the site Home Page</p>
    <br />
    <p>Welcome {PersonName} </p>
  </body>
</html>
```

We shall use the same class name, that is `Home`, for both the controller and the view that we need to run the example; the same convention will be applied to the model classes in the following sections. The unique identification of a class is possible using the familiar concept of absolute path of a file within a file system. For example, the controller `Home.php` is identified by the path `myproject\controllers\Home.php`, while the view `Home.php` is identified by `myproject\views\Home.php`. This convention allows avoiding the proliferation of

names in complex projects and will be used within php classes also using the notion of php namespaces. See the positioning of the `Home` classes for the controller and the view, as well as the template `home.html.tpm`, in the directory hierarchy of WebMVC. Note that in the case of `home.html.tpm`, the name `home` is in lowercase because it is HTML code instead of a class.

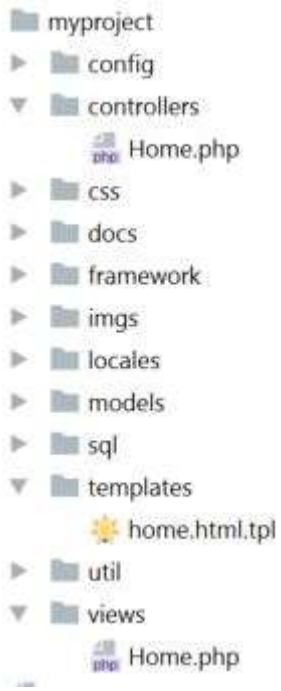


Fig. 3.1. The naming convention for custom defined files.

The setVar method

The code of the `Home` view class, written to handle the dynamic aspects of a page, contains the `setVar()` method inherited from the `framework\View` class. In this example, the method `setvar()` is used to replace the placeholder `{PersonName}` with the name of a person represented by the parameter `$name` of the method `setPersonName($name).`

```
// example 3.2.b: Defining a view that performs a placeholder substitution

<?php

namespace views;
use framework\View;

class Home extends View
{
    public function __construct()
    {
       $tplName = "home";
       parent::__construct($tplName);
    }

    public function setPersonName($name){
```

```
        // setVar is a method inherited from the framework\View class
        $this->setVar("PersonName",$name);
    }
}
```

At runtime, by invoking `setPersonName("Mark")`, the static HTML will be dynamically modified as follows:

```
// example 3.2.c: The placeholder substitution in the file home.html.tpl

<!DOCTYPE html>
<html>
  <head>
    <title>Site home page</title>
  </head>
  <body>
    <p>Welcome to the site Home Page</p>
    <br />
    <p>Welcome Mark</p>
  </body>
</html>
```

Bear in mind that:

- If multiple placeholders `{PersonName}` are in a single HTML file then they will be all replaced in one single `setVar()` code.
- You must call a `setVar()` method after the view has been initialized in the controller.
- You must not call two times the same `setVar()`, because if no placeholder will be found then an exception will be thrown.

Finally, we rewrite our `Home` controller class to call the `setPersonName($name)` method.

The controller provides the method `sayWelcome()` in order to accept the `sayWelcome` command from the user's browser. The management of the template is now done by the user-defined `Home` view.

```
// example 3.2.d: the controller that say welcome to the people provided as parameter

<?php

namespace controllers;

use framework\Controller;
use views\Home as HomeView;

class Home extends Controller
{
    public function __construct() {
        // set the view variable to an instance of the HomeView class
        $this->view = new HomeView();
        parent::__construct($this->view);
    }
```

```
    public function sayWelcome($name) {
        $this->view->setPersonName($name);
        $this->render();
    }
}
```

You can run the method `sayWelcome()` with a parameter to say Welcome to a person. Simply type from your browser the following address:

```
http://localhost/myproject/home/say_welcome/Mark
http://localhost/myproject/home/say_welcome/John
```

Your output will be:

Welcome Mark
Welcome John

3.3 Dynamic block

We can extend the idea of dynamic substitution of a placeholder with a single name to the case where we have to show a list of names. With the framework, you can use the concept of block. A *block* is nothing more than a piece of HTML code delimited by two comments that mark its endpoints. A block is usually subject to a transformation in order to render dynamic content in the browser. The management of blocks is a useful feature, for example, when we have to render a list of records coming from a database.

How can you create a block? Suppose that we have a list of people whose names must be shown in a browser. The file `user_list.html.tpl` provides the static structure of a table in which there exist two placeholders to change: `{FirstName}` and `{LastName}`.

```
// example 3.3.a: the template user_list.html.tpl for the substitution of the block
“User”

<html>
<head>
    <title>User List</title>
</head>

<body>
    <h3>This example shows Blocks usage for data repetition inside a static GUI</h3>
        <table>
            <tr>
                <th>FIRST NAME</th>
                <th>LAST NAME</th>
            </tr>

            <!-- BEGIN User -->
```

```
            <tr>
                <td>{FirstName}</td>
                <td>{LastName}</td>
            </tr>
            <!-- END User -->
        </table>
    </body>
</html>
```

In the HTML code, two comments are present; they together wrap the block of code that will be dynamically replaced many times in the HTML file. These comments mark the BEGIN and the END of a block of name `Users`. This means that WebMVC will be able to recognize the block. Now, we can write the code of `views/UserList` and `controllers/UserList` in order to show a list of people names.

```
// Example 3.3.b: the view UserList that substitutes
// an array of people names to the block “Users”

<?php
namespace views;

use framework\View;

class UserList extends View
{
    public function __construct($tplName = null)
    {
        if (empty($tplName)) {
            $tplName = "/user_list";
                }
        parent::__construct($tplName);
    }

    public function setUserList($userList)
    {
        $this->openBlock("User");
        foreach ($userList as $user){
            $this->setVar("FirstName",$user["FirstName"]);
            $this->setVar("LastName",$user["LastName"]);
            $this->parseCurrentBlock();
        }
        $this->setBlock();
    }
}
```

Note that the constructor of the class `UserList` receives the parameter `$tplName;` this allows to generalize the class behaviour because it can be used in connection with whatever template has the necessity to show a user list. However, the class declares the default template `user_list` to use when the constructor does not receive the name of a template. In this example we use the template `user_list`. We have created a method in the view that uses the block `User` and processes it with all the values contained into the `$userList` received as a parameter. Specifically, we:

- Consider the block `Users` by invoking the `openBlock` method inherited from `framework\View`. This means that, from now, each future action of the `setVar()` method will be restricted to the HTML contained inside the block `User`.
- Then, loop inside `$userList` and for each element we:
    i. Call the `setVar()` method to replace the placeholders `{FirstName}` and `{LastName}` with the value represented by the current element referenced by `$userList;`
    ii. Call the `parseCurrentBlock()` to instruct the View to process the opened block (that contains `{FirstName}` and `{LastName}`). Then, if further people remain in `$userList,` the method arranges the things for the next block. This means that the placeholders contained in the block will be valorized, by invoking once again setVar, as long as there is a couple of name and surname in $userList.
- By calling the `setBlock` method, we close the active opened block; as a result, its content will result dynamically valorized and ready to be shown.

```
// Example 3.3.c: the controller UserList that accept
// a user command to show a list of people names

<?php
namespace controllers;

use framework\Controller;
use views\UserList as UserListView;

class UserList extends Controller
{
    public function __construct() {
      $this->view = new UserListView();
      parent::__construct($this->view);
    }

    public function showUserList() {
      $users = array (
            array("FirstName" => "John", "LastName" => "Red"),
            array("FirstName" => "Mark", "LastName" => "White"),
            array("FirstName" => "Diana", "LastName" => "Brown"),
        );
      $this->view->setUserList($users);
      $this->render();
    }
}
```

Now you can run the controller's method `showUserList` to get a list of people:

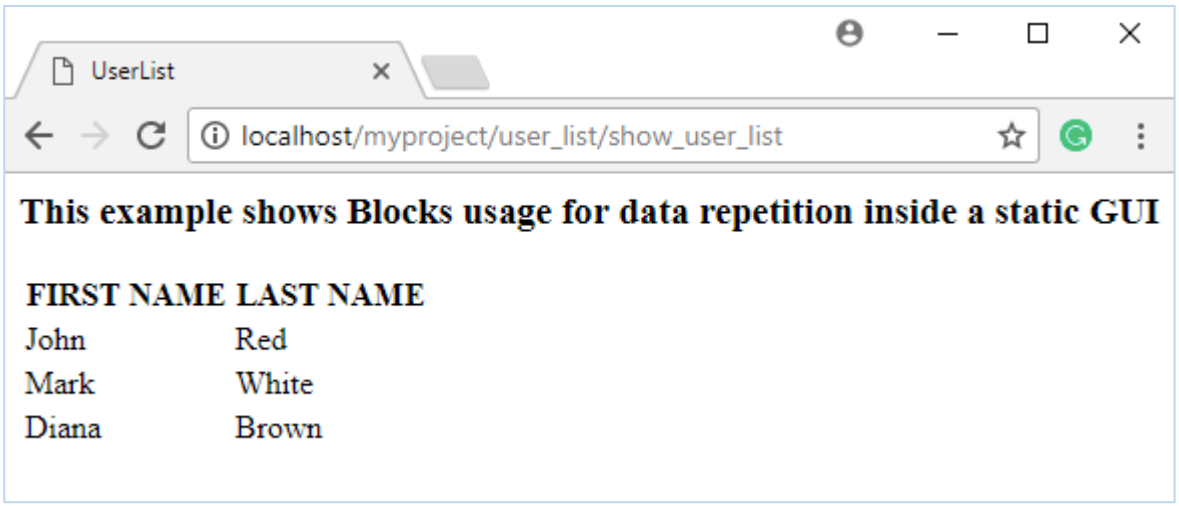


Fig. 3.2. Showing a user list using a template block.

3.4 Block hiding

A recurrent problem of GUI design and implementation is the need to hide parts of an HTML page. This occurs, for example, in a multiuser software application where a high-level role could have full access to the software functionalities presented in a page, while a low-level role could only have restricted access. The *block hiding* capability of WebMVC provides a simple way to hide blocks inside HTML pages.

The next example starts from Example 3.3. We use here a richer template to manage the user interaction so that a list of people can be visualized or hidden. To get this behaviour, we write the template

`toggle_user_list.html.tpl`

and the controller

`ToggleUserList`

Invoking the controller `ToggleUserList` you get the output:

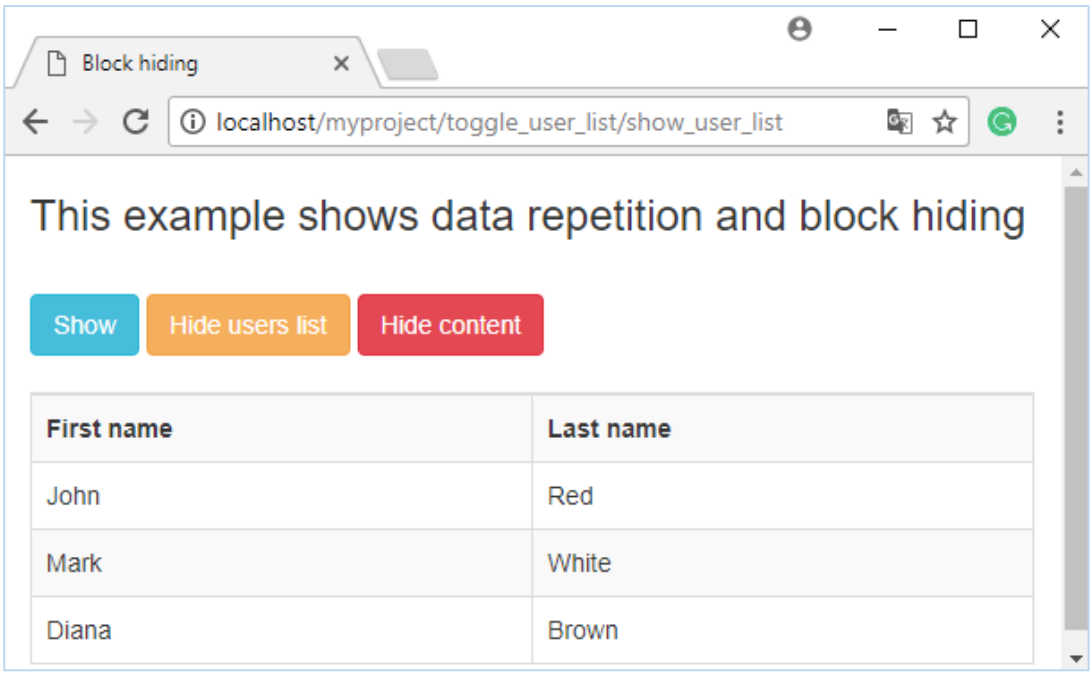


Fig. 3.3. Showing and hiding a user list.

During the interaction, the user can hide either the user list, clicking on the button "Hide user list", or the whole content clicking on the button "Hide content". Since the HTML template `toggle_user_list.html.tpl` contains Bootstrap code to improve the graphical aspect of the presentation that we shall use often in the following, we first describe the HTML that will be reused:

```
<!-- Shared code  3.4.1: The Bootstrap core CSS declarations used in the examples -->

    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <!-- Bootstrap core CSS -->
    <link href="http://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/3.3.4/css/bootstrap.min.css" rel="stylesheet" media="screen">

    <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and media queries -->
    <!--[if lt IE 9]>
    <script
src="http://cdnjs.cloudflare.com/ajax/libs/html5shiv/3.7.2/html5shiv.js"></script>
    <script
src="http://cdnjs.cloudflare.com/ajax/libs/respond.js/1.4.2/respond.js"></script>
    <![endif]-->
```

```
<!-- Shared code 3.4.2: jQuery (necessary for Bootstrap's JavaScript plugins) -->

<script
src="http://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>

<!-- Include all compiled plugins (below), or include individual files as needed -->
```

```
<script src="http://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/3.3.4/js/bootstrap.min.js">
</script>
```

There are four blocks in the HTML body whose content is recalled by their names: `ExampleDescription`, `Warning`, `UserList`, and `User` that, in its turn, is an inner block of `UserList`.

```
// Example 3.4.a. The template toggle_user_list.html.tpl to show and hide a list of
people

<!DOCTYPE html>
<html>
<head>
<title>Block hiding</title>
    <!—- PUT THE SHARED CODE 3.4.1 HERE -->
</head>

<body>
<div class="container">

 <!-- BEGIN ExampleDescription -->
        <h3> This example shows data repetition and block hiding</h3>
 <!-- END ExampleDescription --> <br />

    <a href="/myproject/toggle_user_list/showUserList" class="btn btn-info">Show</a>
    <a href="/myproject/toggle_user_list/hideUserList" class="btn btn-warning">Hide
             users list</a>
    <a href="/myproject/toggle_user_list/hideContent" class="btn btn-danger">Hide
content</a>
    <br /><br />

 <!-- BEGIN Warning -->
          <div class="well h3">
             You are not allowed to view users list
          </div>
 <!-- END Warning -->

 <!-- BEGIN UserList -->
    <div class="table-responsive">
        <table class="table table-bordered table-hover table-striped">
            <tr>
                <th>First name</th>
                <th>Last name</th>
            </tr>

            <!-- BEGIN User -->
            <tr>
                <td>{FirstName}</td>
                <td>{LastName}</td>
            </tr>
            <!-- END User -->
        </table>
    </div>
 <!-- END UserList -->
```

```
</div>

<!-- PUT THE SHARED CODE 3.4.2 HERE -->

</body>
</html>
```

Now the controller code. First of all, the controller reuse the view `UserList` of example 3.3.b passing it the template name `toggle_user_list`. Remember that the events “click on Show”, “click on Hide user list”, and “click on Hide content” that happen when the user interact with the page shown in fig. 3.3, are defined in the template together with a link that specify the URL of the corresponding method for the event handling. There is a method for each event: `showUserList()`, `hideUserList()` and `hideContent()`. Each method perform the appropriate processing when the corresponding event happens. For example, the controller acts on the blocks of `toggle_user_list.html.tpl`, calling the method `hide()` provided by the framework class `View` with the name of the block to hide.

```
Example 3.4.b: The controller ToggleUserList. It accept commands to show or hide a user
list

<?php

namespace controllers;

use framework\Controller;
use views\UserList as UserListView;

class ToggleUserList extends Controller {

  public function __construct() {
     $this->view = new UserListView("/toggle_user_list");
     parent::__construct($this->view);
  }

  public function showUserList() {
      $userList = array (
         array("FirstName" => "John", "LastName" => "Red"),
         array("FirstName" => "Mark", "LastName" => "White"),
         array("FirstName" => "Diana", "LastName" => "Brown") );
      $this->view->setUserList($userList);
      $this->view->hide("Warning");
      $this->render();
  }

  public function hideUserList(){
      $this->view->hide("UserList");
      $this->render();
  }

  public function hideContent() {
      $this->view->hide("ExampleDescription");
      $this->view->hide("Warning");
      $this->view->hide("UserList");
      $this->render();
  }
}
```

Summary

The section 3 shows the main concepts and techniques useful to handle with the presentation layer of Web applications. A static page is managed by WebMVC assigning an HTML file to the predefined `View` class provided by the framework that will take care of the visualization. In WebMVC an HTML files is called template because it usually contains some additional information that instruct the WebMVC engine to do the required job on the HTML code.
The notion of placeholder allows changing the perspective from a static page to a dynamic one. By means of a placeholder, we can do a (dynamic) substitution of a string in a template with another string deriving from data input or processing. In the case of dynamic visualization, it is necessary to introduce new view classes whose structure depends on the kind of user interaction to implement.
When a substantial portion of text in a template is interested by a substitution, we can use a block. A frequent application of this concept is the rendering of a list of records coming from a database. Blocks can be shown or hidden according to the user interaction requirements.

4. Model

What is a model?

In an MVC software architecture, a model is a component that has the responsibility for data management. In other words, the model maintains a repository of data and provides the methods for data recording and retrieval. It is worthwhile to observe that the decomposition into the three components of an MVC architecture reflects the approach of *divide et impera* in which the controller assumes the role of coordinator that assigns the tasks of data management and data presentation to the model and view components respectively. In WebMVC, the definition of a model is similar to that of a controller or a view; in fact, it is sufficient to extend the `framework\Model` class. As an example, we can further discuss the problem of showing a list of people in a browser. In the previous section, the list was taken from the controller; while this could be convenient when the problem to solve is of small dimension (we could do without the model), it is more frequent the case where the data are managed by the model.

```php
//Example 4.1.a: The model UserList that returns an array of people names

<?php

namespace models;

use framework\Model;

class UserList extends Model
{
    public function getUsers()
    {
        $users = array (
            array("FirstName" => "John", "LastName" => "Red"),
            array("FirstName" => "Mark", "LastName" => "White"),
            array("FirstName" => "Diana", "LastName" => "Brown"),
        );
        return $users;
    }
}
```

The controller must take into account the coordination of view and model:

- Linking the variable `$this->view` and `$this->model` to the corresponding class instances passing them to the constructor of the `framework\Controller` class
- Using the instantiated model and view to retrieve and visualize the array of people through the `getUsers()` method

The controller retrieves the data from the model; then, calling the view, it arranges the presentation. Note that the codes of `UserList` view and `user_list_html.tpl` are unchanged.

```php
// example 4.1.b: The controller UserList now coordinates model and view.
// It accepts the invocation of showUserList(), takes data from the model passing them
// to the view. Here we reuse the view of example 3.3.b.

<?php
namespace controllers;

use framework\Controller;
use models\UserList as UserListModel;
use views\UserList as UserListView;

class UserList extends Controller
{
    public function __construct() {
        $this->view =  new UserListView("user_list");
        $this->model = new UserListModel();
        parent::__construct($this->view,$this->model);
    }

    public function showUserList() {
       $userList = $this->getModel()->getUsers();
       $this->view->setUserList($userList);
       $this->render();
    }
}
```

Database interaction

Having in mind the role of a model and how to use it in an MVC architecture, we can modify the previous example and retrieve the array of people from a database. Before we do so, it must be taken into account that:

- WebMVC uses **MySqli** to interact with the database
- The variable `$this->sql`, and the methods `updateResultSet()`, and `getResultSet()` are inherited from the `framework\Model` class
- `updateResultSet()` executes a query previously stored in the variable `$this->sql`
- `getResultSet()` returns the result set of the executed query
- You must configure the file config\application.config.php. Specifically, you must modify the constants DBHOST, DBUSER, DBPASSWORD, DBNAME and DBPORT

In this example, we modify the `UserList` classes by taking a set of people from a database. We assume the availability of a table called "people" containing the same data of the array `$users` in the model definition of Example 4.1.a.

- people -> table name
- name -> the first attribute with the person name
- surname -> the second attribute with the family name

For the sake of simplicity, we change only the methods of the previous classes:

```
// Example 4.2: Changing the classes of Example 4.1 to enable database interaction

// getUsers of model\UserList

public function getUsers() {
    $this->sql = "SELECT * FROM people";
    $this->updateResultSet();
    return $this->getResultSet();
}

// showUserList of controllers\UserList

public function showUserList() {
   $userResultSet = $this->model->getUsers();
   $this->view->setUserList($userResultSet);
   $this->render();
}

// setUserList of views\UserList

 public function setUserList(\mysqli_result $userResultSet) {
    $this->openBlock("User");
    while ($people = $userResultSet->fetch_object()) {
       $this->setVar("FirstName", $people->name);
           $this->setVar("LastName", $people->surname);
       $this->parseCurrentBlock();
    }
    $this->setBlock();
```

```
}
```

To run the code, type the URL:

```
localhost/myproject/user_list/show_user_list
```

What if you want to execute a SQL query that is different from a select? You can use the `query()` method of the model. For example, to insert a person called `George` in the people table you should use:

```
$this->query("insert into people(name) VALUES('George')").
```

The `query()` method can execute every type of SQL operation (e.g. insert, update, select etc). This method can return:

- false, if an error occurred
- true, if no error occurred and the query is not a select
- a result set if no error occurred and the query is a select

Summary

To run an MVC instance of an application we have seen how to:

- create and run a controller and its methods calling them from the URL
- create a views class linking it to a template file
- substitute a dynamic variable to a placeholder inside a template
- declare a block that must be transformed, for example in a list of values, within a template
- create the model class taking the result set from a database of people.

In WebMVC, the flow of control of an MVC application that comprises model, view and controller is the following:

1. The URL calls a controller specifying one among the alternatives:
   1.1 the name of the controller
   1.2 the name of the controller and a method without parameters
   1.3 the name of the controller and a method with parameters
2. The controller runs retrieving data from the model
3. The data retrieved from the model are sent to the view by the controller
4. The view organizes the data for the presentation.
5. Finally, WebMVC sends the output of the execution to the user.

In the next page, we shall discuss how to organize your project into subsystems.

5. System decomposition

One of the purposes of WebMVC is to provide tools that allow software developers to take advantage of sound principles when they design and implement complex web applications. An important principle of software engineering is *system decomposition* that can be used to split out a software system into smaller interacting parts, called subsystems, in order to dominate the system complexity.

In WebMVC, the decomposition of a software can be pursued considering several perspectives. We have already seen how the splitting into model, view, and controller can be done. MVC is a canonical architectural pattern that can be applied in a great variety of software applications, and in a certain sense, we can say that the MVC decomposition pattern can be used for many application domains regardless of their structure.

Another decomposition perspective concerns how to split a software with respect to an application domain. Consider, for example, a software system that has the purpose to manage some fundamental functions of an enterprise such as `manufacturing` and `crm` (client relationship management); we call it `minierp`. After the system design phase made by the software engineer, the subsystems structure can be represented in WebMVC by means of two fundamental concepts strictly related to each other. They are:

- a hierarchy of directories;
- the namespace.

For each subsystem, WebMVC uses a directory to physically store all its classes and uses a namespace to refer each class when it needs to be instantiated and executed; directories and namespaces must have identical names that are conventionally written in lowercase. For example, if we decide to implement the subsystem `manufacturing` we must define:

1. `controllers\manufacturing` - the *directory path* where to store classes for the manufacturing subsystem;
2. `controllers\manufacturing` - the *namespace* that we must use when coding each PHP class of the manufacturing subsystem.

An excerpt of directory structure for the `minierp` web application is shown in the figure 6.1. There are now two decomposition levels: the first is the MVC decomposition (directories: controllers, models, and views/templates) and the second is provided by the structure of the application domain. Note that we have a replica of the subsystem application names within the models, views, and controllers directories as well as for the templates directory. In the `minierp` software, the initial application domain decomposition is made by the subsystems `crm`, and `manufacturing`; the latter comprises the class `Inventory`. The directory `controllers` is the root directory from which all application controllers for the defined subsystems can be invoked. This is because in WebMVC the directory controllers is the entry point to access application software functionalities.

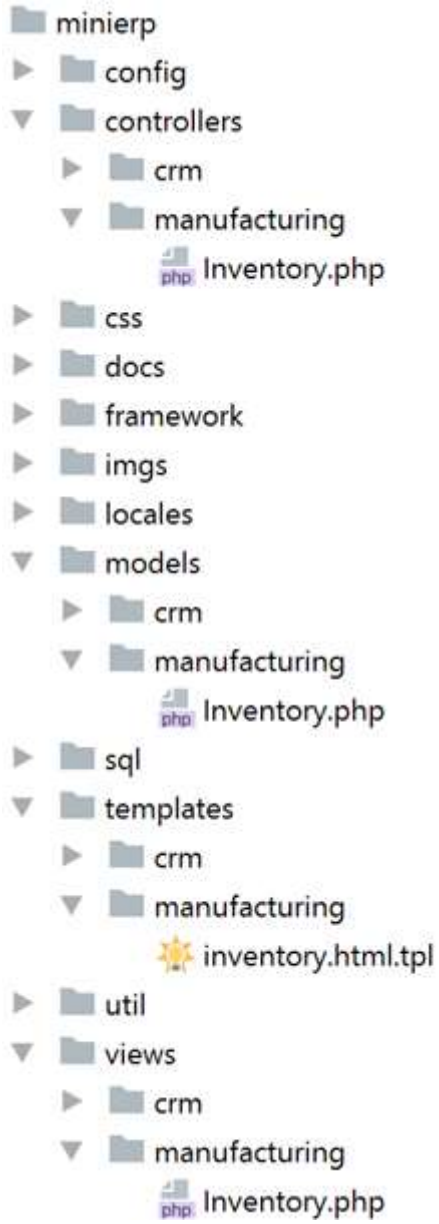


Fig. 5.1. The structure of directory for system decomposition.

How to invoke a subsystem controller class

We know that in order to call a controller, we have to write an URL according to the following formats:

- `http://<site>/<controller name>`
- `http://<site>/<controller name>/<method name>`
- `http://<site>/<controller name>/<method name>/parName`$_1$`/parName`$_2$`/.../parName`$_n$

where the automatic conversion from the URL to the class name, method and parameters works, for example, as follows:

`http://site/user_manager/get_user/1 => UserManager->getUser(1).`

We extend this convention in order to call a controller class located within a subsystem using formats like:

- `http://site/subsystem/<controller name>/<method name>/parName1/.../parNamen`
- `http://site/subsystem/.../subsystem/<controller name>/<method name>/parName1/.../parNamen`

Managing the inventory record

An example taken from the `minierp` web application concerns the presentation of an *inventory app* whose controller is located within the class `manufacturing/Inventory`. It is a simplified version of a software application that aims at the inventory management of a manufacturing industry. The inventory table is taken from the database named `minierp` and is made of the following attributes:

- `code` -> the record key
- `description` -> the description of the good maintained in the inventory
- `stock` -> quantity in stock

The task to retrieve the inventory records is in charge of `models\manufacturing\Inventory`:

```
// Example 5.a: The model Inventory takes data from a database table

namespace models\manufacturing;

use framework\Model;

class Inventory extends Model
{
    public function getInventory()
    {
        $this->sql = "SELECT * FROM inventory";
        $this->updateResultSet();
        return $this->getResultSet();
    }
}
```

Next, the `views\manufacturing\Inventory` class receives the records retrieved by the model and proceeds with the substitution of the template placeholders contained in `templates\manufacturing\inventory` with the values taken from the records. The Inventory view takes advantage of the concept of block.

```
// Example 5.b: The view Inventory manipulates the HTML block named “Part”
// substituting the values retrieved by the model to the placeholders in the block
```

```php
namespace views\manufacturing;

use framework\View;

class Inventory extends View
{
    public function __construct($tplName = null)
    {
        if (empty($tplName))
            $tplName = "/manufacturing/inventory";
        parent::__construct($tplName);
    }

    public function setInventoryBlock($inventoryResultSet) {
        $this->openBlock("Part");
        while ($part = $inventoryResultSet->fetch_object()) {
            $this->setVar("code",$part->code);
            $this->setVar("description",$part->description);
            $this->setVar("stock",$part->stock);
            $this->parseCurrentBlock();
        }
        $this->setBlock();
    }
}
```

The file `inventory.html.tpl` simply arranges the output in the form of a table. The block named `Parts` states how the record retrieved by the inventory table will be rendered in output one row at a time by the Inventory view.

```
// Example 5.c: The template inventory.html.tpl containing the block “Parts”

<!DOCTYPE html>
<html>
<head>
    <title>Inventory</title>
</head>
<body>
    <h1>Inventory</h1>
        <table>
            <thead>
                    <th>code</th>
                    <th>description</th>
                    <th>stock</th>
            </thead>
            <tbody>
                <!-- BEGIN Part -->
                <tr>
                    <td>{code}</td>
                    <td>{description}</td>
                    <td>{stock}</td>
                </tr>
                <!-- END Part -->
            </tbody>
        </table>
</body>
</html>
```

Finally, the code of the `Inventory controller` coordinates, as usual, the work made by the model and the view. We remark that the function `showInventory()` has to be public; it first invokes the method `getInventory()` from the model, then it passes the result set to the method `setInventoryBlock()` of the view that arranges for the placeholder substitutions with the values contained in the retrieved records.

```
// Example 5.d: The Inventory controller.
// The Inventory controller accept a request to show the inventory
// and coordinates the job of model and view to get the result

namespace controllers\manufacturing;

use framework\Controller;
use framework\Model;
use framework\View;

use models\manufacturing\Inventory as InventoryModel;
use views\manufacturing\Inventory as InventoryView;


class Inventory extends Controller
{
    public function __construct()
    {
        $this->view = new InventoryView("/manufacturing/inventory");
        $this->model = new InventoryModel();
        parent::__construct($this->view,$this->model);
    }

   public function showInventory() {
                $inventoryResultSet = $this->model->getInventory();
                $this->view->setInventoryBlock($inventoryResultSet);
                $this->render();
   }
}
```

Assuming that in the table inventory there are data of components necessary to build a digital mouse, we can get the output typing:

`http://localhost/minierp/manufacturing/inventory/show_inventory`

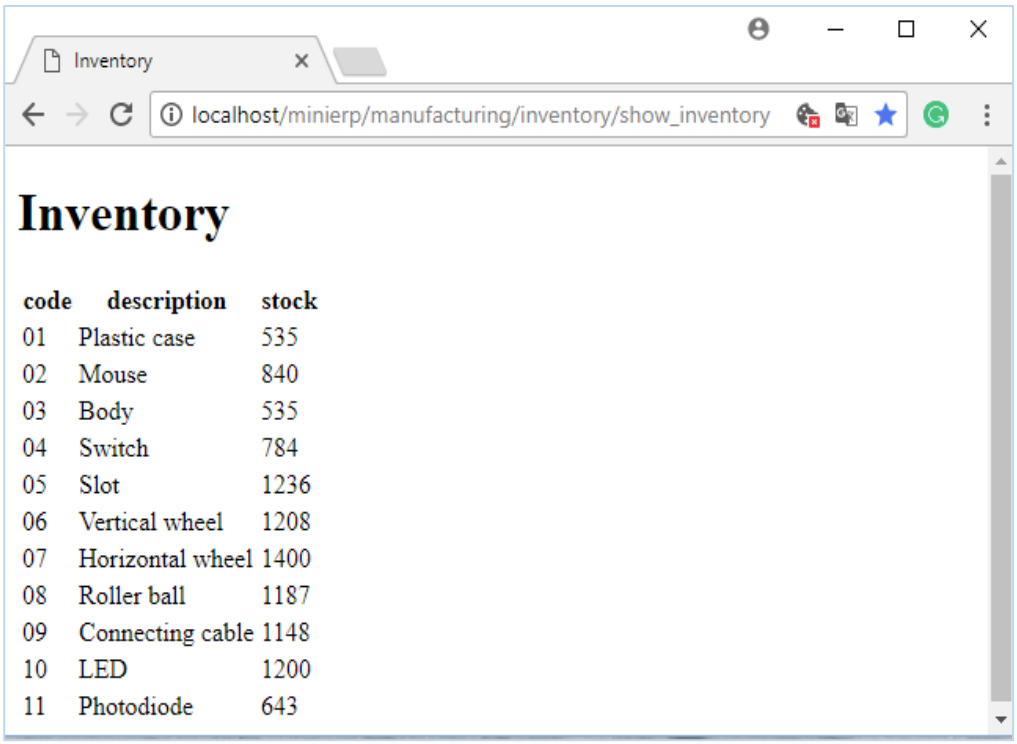


Fig. 5.2. An inventory of materials for the construction of a mouse.

6. Structuring MVC units

The figures 3.1 and 5.1 outline how the convention that uses the same name for models, views, and controllers directories and classes can help to reduce the proliferation of names in complex projects.

There is another reason to give the same name to model, view, controller classes, and the template relied to the view. All these parts (as well as other code logically related to these parts such as CSS, JavaScript and other resources) can be perceived as a conceptual unit, identified by a name, devoted to the solution of a problem. In other words, we can abstract the aggregate of different cooperating software parts into a single logical unit that receives a name that evokes the problem to solve.

In WebMVC, an *MVC unit* is a logical aggregate of software parts that cooperate to solve a problem. For example, we can indicate the set of classes named `Inventory` in examples 5.a (the model), 5.b (the view), 5.d (the controller), and the template `inventory` shown in 5.c as a whole that we identify as the Inventory MVC unit:

```
Inventory = { controllers\Inventory,
              models\Inventory
              views\Inventory,
              templates\inventory }
```

This abstraction step is very useful when we need to construct complex software systems using the techniques discussed in next sections. Indeed, it is not necessary that all the software parts of an MVC unit share the same name. Frequently, the name of one or more parts differs from the name chosen for the MVC unit; this is the case when a software part already existing is reused to simplify the development of an MVC unit. For example, the MVC unit `ToggleUserList` of section 3.4 is represented by:

```
ToggleUserList = { controllers\ToggleUserList,
                   views\UserList,
                   templates\toggle_user_list }
```

Web applications can be complex to design and realize. In addition to the two-level decomposition discussed so far (MVC decomposition and application subsystems), WebMVC provides some techniques for structuring MVC units that can be used to easy the implementation of the control flow of complex web applications. We discuss these techniques in the next sections:

- Coding guidelines
- Hierarchical MVC
- Controller inheritance

6.1. Coding guidelines

Coding guidelines refers to the way we can structure the parts of an MVC unit in order to get:

- generality through a coding pattern that can be taken into consideration for reuse, inheritance and automatic generation of code as well;

- an additional decomposition technique that split the responsibility of a controller constructor into two parts: the constructor and the method `autorun()`.

A simple expedient to get a reusable view class is to pass it a template as parameter. This technique has been used for the class `views\UserList` defined in section 3.3 and reused in example 3.4. If we consider the definition of a controller, we have the opportunity to pass two references: `$view` and `$model`; in this case, the controller uses these particular instances. However, when we do not pass these references, the controller has a default model and a default view, retrieved by means of the methods `getView()` and `getModel()`, that qualify its standard behaviour.

The code of example 6.1.1 generalizes that described in example 4.1.b. It also shows an additional feature of WebMVC: the division of responsibilities between the constructor and the protected method `autorun()`. This method is automatically invoked after the constructor execution. A typical way to exploit this separation of responsibilities is:

- the constructor create/ retrieves objects, establishing relations among them, and predisposing the things to start the computation;
- the `autorun()` method perform the computation.

In other words, the costructor is focused on *static properties* of a system (the parts and the relations among them) whereas the `autorun()` is focused on the *dynamic properties* of a system expressed in terms of *relation activations* (running a controller, invoking a method, performing a loop, etc.).

```
// Example 6.1.1: The generalized version of the controllers\UserList

<?php

namespace controllers;

use framework\Controller;
use models\UserList as UserListModel;
use views\UserList as UserListView;

class UserList extends Controller
{
     public function __construct(View $view=null, Model $model=null) {
        $this->view = empty($view) ? $this->getView() : $view;
        $this->model = empty($model) ? $this->getModel() : $model;
        parent::__construct($this->view,$this->model);
     }

    protected function autorun ($parameters=null) {
      $userList = $this->model->getUsers();
      $this->view->setUserList($userList);
    }

    public function getView() {
        $view = new UserListView("/user_list");
        return $view;
    }

    public function getModel() {
        $model = new UserListModel();
        return $model;
    }
}
```

The coding pattern exemplified in the example 6.1.1, is also used by a skeleton class generator provided by WebMVC. A natural consequence deriving from the automatic invocation of autorun() is that we can assign to it the main (dynamic) responsibility of a class. In this way, we can invoke the execution of the main behaviour of a class without

explicitly calling a method. For example, the autorun() method of example 6.1.1 plays the role of the method showUserList() of example 4.1.b (or of the modified version of example 4.2), but is automatically invoked. Now you can try the execution of this version of the controller UserList typing:

```
localhost/myproject/user_list
```

6.2. Hierarchical MVC

Until now, we have executed examples involving a single controller. In the web, it is usual to divide an HTML page into several sections each one with a different purpose (e.g. a gallery of images, a navigation bar etc...) and different interaction modalities. Usually, some of these sections are reused in different pages. A design principle that can be used to easy the management of different interactions that a user can have with a page is the hierarchical decomposition of MVC units (HMVC). From the point of view of the flow of control, the focus is on the hierarchy of controllers; therefore, the composition of MVC units can also be regarded as controller composition. In WebMVC, a controller has the capability to use other controllers; this allows building a hierarchy that brings the benefits of complexity reduction and software reuse. Controller composition is useful, for example, when an HTML page presents two or more parts that require different kinds of user interaction, and an ad hoc controller is necessary to manage each of these parts. Usually, a given controller acts as a supervisor, and the other controllers perform the remaining control flow each on the assigned part. When a supervisor controller invokes a subordinate one, it implicitly invokes the execution of an MVC unit typically made of controller, model, view, and template, and eventually other related code.

Suppose that we have the problem to manage the user interaction deriving from the presentation of this page:

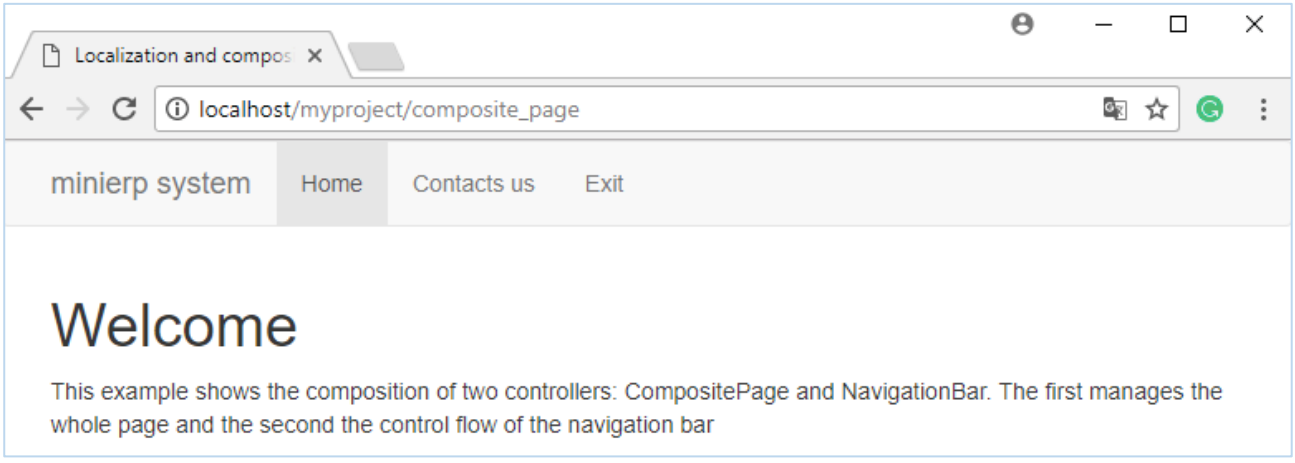


Fig. 6.1. A page with two logical parts: a welcome message and a navigation bar.

The page is made of two parts; the first output a welcome message and the second exposes a navigation bar that requires some input interaction. We can use two MVC units to manage the GUI. We call the first `CompositePage`, assigning it the responsibilities to show a welcome message and to supervise the work of the second MVC unit, called `NavigationBar`, that manages instead the user input interaction. `NavigationBar` and `CompositePage` are MVC units that aggregate the following code respectively:

```
NavigationBar = { controllers\NavigationBar,            // ex. 6.2.a
                  views\NavigationBar,                  // ex. 6.2.b
                  templates\navigation_bar }            // ex. 6.2.c

CompositePage = { controllers\CompositePage,            // ex. 6.2.d
                  models\CompositePage                  // ex. 6.2.e
                  views\CompositePage,                  // ex. 6.2.f
                  templates\composite_page,             // ex. 6.2.g
                  {NavigationBar} }                     // subordinate MVC unit
```

The MVC unit `NavigationBar` placed within the aggregate that comprises `CompositePage`, highligths that `NavigatonBar` is a subordinate MVC unit. First, we present its code:

```php
// example 6.2.a: The NavigationBar controller that manages the user interaction.
// This class is used by the controller CompositePage.

<?php

namespace controllers;

use framework\Controller;
use views\NavigationBar as NavigationBarView;

class NavigationBar extends Controller
{

    protected function autorun($parameters = null)
    {
        $this->view = new NavigationBarView();
    }
}
```

```php
// Example 6.2.b: The view class NavitationBar. This view
// loads the template navigation_bar.

<?php

namespace views;

use framework\View;

class NavigationBar extends View
{
    public function __construct($tplName = null)
    {
        if (empty($tplName))
            $tplName = "/navigation_bar";
```

```
        parent::__construct($tplName);
    }


}
```

```
<!-- Example 6.2.c: The template navigation_bar.html.tpl. It exposes the GUI to
                    navigate the application -->


<nav class="navbar navbar-default">
   <div class="container">
     <div class="navbar-header">
           <button type="button" class="navbar-toggle collapsed" data-toggle="collapse"
                  data-target="#navbar" aria-expanded="false" aria-controls="navbar">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="">minierp system</a>
        </div>
        <div id="navbar" class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li class="active"><a href="">Home</a></li>
                <li><a href="">Contacts us</a></li>
                <li><a href="">Exit</a></li>
            </ul>
        </div>
    </div>
</nav>
```

The controller `CompositePage` supervises the whole job. The flow of control starts in the constructor that creates the view and the model instances necessary to `CompositePage` to solve the presentation problem of fig. 6.1. Then, the flow proceeds in the method `autorun()` that performs the following steps:

1. the model returns the welcome message passed to the view;
2. an instance of the controller `NavigationBar` is created with the purpose to manage the user interaction by means of a navigation bar;
3. the method `bindController()`, of the framework class `Controller`, is invoked. This method instruct the framework to use the MVC unit `NavigationBar` for the GUI necessary to navigate the page of fig. 6.1.

```
// Example 6.2.d: The CompositePage controller.
// this class comprises the controller NavigationBar.

<?php

namespace controllers;

use framework\Controller;
use framework\Model;
use framework\View;
```

```php
use models\CompositePage as CompositePageModel;
use views\CompositePage as CompositePageView;
use controllers\NavigationBar;

class CompositePage extends Controller
{
    protected $view;
    protected $model;

    public function __construct()
    {
        $this->view = new CompositePageView("/composite_page");
        $this->model = new CompositePageModel();
        parent::__construct($this->view,$this->model);
    }

    protected function autorun($parameters = null)
    {
        $message = $this->model->getMessage();
        $this->view->setVarMessage($message);
        $navigation = new NavigationBar();
        $this->bindController($navigation);
    }
}
```

Exercise: write the generalized version of CompositePage according to the code pattern of example 6.1.1.

The model simply returns a message:

```php
// Example 6.2.e: The Model class Composite Page; it simply returns a message.

<?php

namespace models;

use framework\Model;

class CompositePage extends Model
{
 public function getMessage()
   {
    return "This example shows the composition of two controllers:
            CompositePage and NavigationBar. The first manages the whole page
            and the second the control flow of the navigation bar";
   }
}
```

```php
<?php

// Example 6.2.f: The view class CompositePage. This view loads the template
// composite_page and inject in it the welcome message returned by the model.

namespace views;

use framework\View;
```

```
class CompositePage extends View
{
    public function __construct($tplName = null)
    {
        if (empty($tplName))
            $tplName = "/composite_page";
        parent::__construct($tplName);
    }

    public function setVarMessage($value)
    {
        $this->setVar("Message",$value);
    }

}
```

From the point of view of the composition of WebMVC controllers, the relevant aspect of the template `composite_page.html.tpl` is the placeholder contained in the body section:

`{Controller:NavigationBar}.`

When the supervisor controller `CompositePage` described in example 6.2.d invokes the method `bindController($navigation)`, it instruct the framework to use the controller class `NavigationBar` in the point where appears the placeholder `{Controller:NavigationBar}`. The effect of `bindController()` is that:

a) the placeholder `{Controller:NavigationBar}` is replaced by the template `navigation_bar` of example 6.2.; thus, we obtain a richer template that allows the user to navigate the page;
b) The MVC unit `NavigationBar` manages the GUI for the page navigation.

```
<!—Example 6.2.g. The template composite_page.html.tpl declares the use of the
controller NavigationBar to manage the section devoted to user interaction -->

<!DOCTYPE html>
<html>
<head>
<title>Composite Page</title>
    <!—- PUT THE SHARED CODE 3.4.1 HERE -->
</head>
<body>

{Controller:NavigationBar}

<div class="container">
    <h1>Welcome</h1>
    <p>{Message}</p>
</div>

<!—- PUT THE SHARED CODE 3.4.2 HERE -->
</body>
```

```
</html>
```

6.3. MVC inheritance

The second technique that we can use to structure a controller is *MVC inheritance*. It is just a particular case of the standard class inheritance of OOP where a child controller inherits from a father controller that implicitly makes available a complete MVC unit to the child. Again, since the focus is on the controller, we can use the term *controller inheritance* as well. This technique can be useful when we already have an MVC solution of a problem that can be used "as is" or with little modifications to solve a wider problem. To illustrate how controller inheritance works, consider again the problem of showing/hiding a list of people, and suppose that we already have a solution for the problem "show a list of people". We can take as reference the solution proposed in section 4 for the presentation of a list of users with a complete MVC unit made of:

```
UserList= { controllers\UserList,            // ex. 6.1.1,
            views\UserList,                  // ex. 3.3.b, as modified by ex. 4.2
            model\UserList                   // ex. 4.1.a, as modified by ex. 4.2
            templates\user_list }            // ex. 3.3.a
```

Having the problem of showing/hiding a list of people, we can write a new version of ToggleUserList that now use controller inheritance. The controller class ToggleUserList extends the controller UserList, and therefore it reuse the MVC unit UserList. Because the GUI of fig. 3.3 is managed by the template toggle_user_list, the only thing that we have to do is to replace `user_list` with toggle_user_list. The MVC unit `ToggleUserList` represents MVC inheritance as follows:

```
ToggleUserList= { controllers\ToggleUserList,
                  UserList:controllers\UserList    // ToggleUserList inherits
                  UserList:views\UserList,         // the MVC unit UserList
                  UserList:model\UserList
                  templates\toggle_user_list }
```

The code of the controller `ToggleUserList` is shown below. The template code `toggle_user_list` is loaded by the method `loadCustomTemplate` provided by the `framework\View` class; the code of `toggle_user_list` is that reported in ex. 3.4.a.
The inheritance is used when the controller invokes the `parent::autorun()` method. By convention, `autorun()` implements the main responsibility of a class so that the class `ToggleUserList` ask to the controller `UserList` to do its job, that is, to show a list of users using the received template `toggle_user_list`.

```
// Example 6.2. The variant of ToggleUserList with hiding/showing capabilities.
// Now we extend the already existing class UserList.
```

```php
<?php

namespace controllers;

use framework\Controller;
use models\UserList as UserListModel;
use views\UserList as UserListView;

class ToggleUserList extends UserList

{
  public function showUserList() {
      $this->view->loadCustomTemplate("templates/toggle_user_list");
      parent::autorun();
      $this->view->hide("Warning");
      $this->render();
  }

  public function hideUserList(){
      $this->view = new UserListView("/toggle_user_list");
      $this->view->hide("UserList");
      $this->render();
  }

  public function hideContent() {
      $this->view = new UserListView("/toggle_user_list");
      $this->view->hide("ExampleDescription");
      $this->view->hide("Warning");
      $this->view->hide("UserList");
      $this->render();
  }
}
```

7. Software Components

Reusing architectures, design patterns, and off-the-shelf components can significantly reduce the development effort required to deliver a system (Brugge, Dutoit 2014). People responsible for the decision about the better strategies to choose during the software development, are concerned with the problem of how much reuse must be taken into consideration. In general, there are several possibilities:

- *Architecture reuse*. It is the case where a project responsible, for example a project manager or a software architect, decide to adopt an already existing software architecture when it is suitable to develop the software project.

- *Design pattern reuse*. Design pattern provide solutions to partial design problems. Design pattern reuse is similar to architecture reuse but the usability range is typically restricted to a well-established design aspect. [Gamma et. al., 1994]

- *Framework reuse*. Typically, a framework is shared among several projects. Even if the framework impose some constraint, because of its standard way to build and deploy applications, these constraint are usually beneficial in terms of orderly development, knowledge sharing and effort alignment.

- *Component reuse*. Components of an application, ranging in size from subsystems to single objects, may be reused.

In Sommerville, you can find a discussion about the benefits of component-based software engineering. The following definition of software component is due to Szypersky:

"A software component is a unit of composition with contractually-specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

A widely used definition is due to UML:

"A *modular part of a system, that encapsulates its content and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces*".

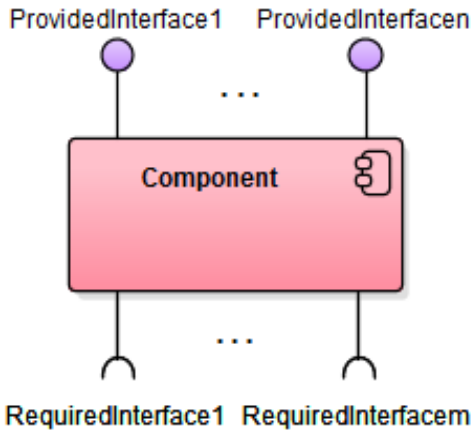


Fig. 7.1. The UML representation of a software component.

A component may be replaced by another if and only if their provided and required interfaces are identical. This idea is the underpinning for the plug-and-play capability of component-based systems and promotes software reuse.

For recurring problems occurring during the implementation of data intensive applications, WebMVC provides software components that can be reused to easy the software development. In WebMVC, a component is just like a ready to use MVC unit, that exhibits a

recurring behavioural pattern, whose reuse reduce the development effort. The following sections show how to use the components provided by the framework.

7.1 Data Repetition

The components provided by the framework are instances of the abstract class `Component`, in its turn an instance of the abstract class `Controller`; therefore, a component is a full-fledged controller. The simpler component made available by WebMVC is `framework\components\DataRepeater` provided by the framework to easy the displaying of a data coming from a given source. Two possible scenarios where the `DataRepeater` can be conveniently used are when: 1) a list of records from a database or
2) data stored in an array, must be provided in output according to a given visualization structure. In the first example of data repetition, we propose a variant of the MVC unit

```
UserList= { controllers\UserList,            // ex. 4.1.b
            views\UserList,                  // ex. 3.3.b
            model\UserList,                  // ex. 4.1.a
            templates\user_list }            // ex. 3.3.a
```

where the source data are retrieved by an array directly maintained by `model\UserList`. Given that `DataRepeater` is a component that populates variables of a template block with data coming from an array or from a database table, we can compose the variant of `UserList` as follows:

```
UserListDataRepeater = { controllers\UserListDataRepeater,    // ex. 7.1.a
                         UserList:controllers\UserList,      // ex. 4.1.b (inheritance)
                         views\UserList,                     // ex. 7.1.b
                         UserList:model\UserList,            // ex. 4.1.a (inheritance)
                         UserList:templates\user_list,       // ex. 3.3.a (inheritance)
                         framework\components\DataRepeater }
```

where the controller `UserListDataRepeater` shown in `example 7.1.a` uses the component `DataRepeater` to populate a block and inheritance to extend the controller `UserList`. The constructor of `DataRepeater` takes as parameters: a view, a model, the name of the template block to manipulate, and a reference to an array. Apparently, the MVC unit `UserListDataRepeater` seems more complicated than `UserList` because it has 6 components instead of 4. However, the inheritance can be avoided, if you wish, completing the definition of `example 7.1.a` with the methods of the controller `UserList`, and then removing it from the MVC unit. Furthermore, the definition of `views\UserList` is somewhat simplified, as shown in the `example 7.1.b`; this simplification becomes more evident when we have to substitute many placeholders in a template.

```
// example 7.1.a: the controller UserListDataRepeater uses the component DataRepeater

<?php
```

```php
namespace controllers;

use framework\components\DataRepeater;

class UserListDataRepeater extends UserList
{
    public function autorun($parameters = null)
    {

        $users = $this->getModel()->getUsers();

       // DataRepeater passes the values retrieved by the model to the view in order
       // to populate the HTML block named “User”. The second parameter is null
       // because we wish to retrieve our data from the associative array $users and
       // not from a DB. The keys in the array must have the same name of placeholders
       // in the block.
       // With the help of the framework class Controller, the method render()
       // populates the the block ”User”.

        $repeater = new DataRepeater($this->view, null ,"User", $users);
        $repeater->render();

    }
```

The version of `views\UserList` is now simplified because of the work aiming at the population of values within a template is now done by the component `DataRepeater`.

```php
// example 7.1.b: compare this version of UserList with that of example 3.3.b.

<?php
namespace views;

use framework\View;

class UserList extends View
{
    public function __construct($tplName = null)
    {
        if (empty($tplName)) {
            $tplName = "/user_list";
         }
        parent::__construct($tplName);
    }
}
```

The second application that uses the component `DataRepeater` implements a web page that shows a list of parts from the inventory table of the `minierp` system. The page contains a navigation bar and a list of parts from the inventory. In addition to the use of `bindController()`, already discussed in section 6.2, the controller `InventoryDataRepeater` uses the `bindComponent()` method provided by the framework class `Controller`. It plays a role that is analogous to that of `bindController()`, described just before example 6.2.g, with the difference that now the placeholder to substitute in the template has the structure

```
{<component name>:<placeholder name>}.
```

From a technical point of view, the use of `bindComponent()` is not strictly necessary for the component `DataRepeater` because it has not a matching placeholder in the corresponding template of `example 7.1.e`. Indeed, in the code of `example 7.1.d` we could replace the instruction

```
$this->bindComponent($parts);
```

with

```
$repeater->render();
```

obtaining the same behaviour. However, `bindComponent()` is usually necessary to bind a component to a template as we will show during the discussion of the `Paginator` component in section 7.2.

The MVC unit `InventoryDataRepeater` is aggregated as follows:

```
InventoryDataRepeater =
 { controllers\manufacturing\InventoryDataRepeater,       // ex. 7.1.d
   views\manufacturing\Inventory,       // ex. 7.1.b (use Inventory instead of UserList
                                        //      and $tplName="/manufacturing/inventory";)
   model\manufacturing\Inventory,              // ex. 7.1.c
   templates\manufacturing\inventory,          // ex. 7.1.e
   {manufacturing\NavigationBar},              // the MVC unit NavigationBar (hierarchy)
   framework\components\DataRepeater }        // the framework component DataRepeater
```

```
// Example 7.1.c: The model Inventory. Compare this code with the code of example 5.a.

namespace models\manufacturing;

use framework\Model;

class Inventory extends Model
{
    public function __construct()
    {
        parent::__construct();
        $this->sql = "SELECT * FROM inventory";
        $this->updateResultSet();
    }
}
```

As usual, the constructor of the controller `InventoryDataRepeater` of example 7.1.d builds the necessary structure for the computation. The method `autorun()` invokes the component `DataRepeater` to show a list of records into your browser. `DataRepeater` passes the records retrieved by the model to the view in order to substitute the HTML block named `Part` with the values of the retrieved record. The fourth parameter is null because we wish to retrieve data from a DB.

```php
// example 7.1.d: the controller InventoryDataRepeater.

<?php

namespace controllers\manufacturing;

use framework\Controller;
use framework\Model;
use framework\View;

use models\manufacturing\Inventory as InventoryModel;
use views\ manufacturing\Inventory as InventoryView;
use controllers\NavigationBar;
use framework\components\DataRepeater;


class InventoryDataRepeater extends Controller
{
    public function __construct(View $view=null, Model $model=null)
    {
        $this->view = empty($view) ? $this->getView() : $view;
        $this->model = empty($model) ? $this->getModel() : $model;
        parent::__construct($this->view,$this->model);
        $navigation = new NavigationBar();
        $this->bindController($navigation);
    }


    protected function autorun($parameters = null) {

        $parts = new DataRepeater($this->view, $this->model, "Part", null);
        $this->bindComponent($parts);
    }

    public function getView()
    {
        $view = new InventoryView("/manufacturing/inventory");
        return $view;
    }

    public function getModel()
    {
        $model = new InventoryModel();
        return $model;
    }
}
```

```
// example 7.1.e: the inventory template. It contains a placeholder to bind the
// controller NavigationBar

<!DOCTYPE html>
<html>
<head>
   <title>Inventory Data Repeater</title>
    <!-- PUT THE SHARED CODE 3.4.1 HERE -->
```

```
</head>


<body>
{Controller:manufacturing\NavigationBar}
 <div class="container">
    <h1>Inventory</h1>
    <div class="table table-responsive">
        <table class="table-bordered">
            <thead>
                    <th>code</th>
                    <th>description</th>
                    <th>stock</th>
            </thead>
            <tbody>
                <!-- BEGIN Part -->
                <tr>
                    <td>{code}</td>
                    <td>{description}</td>
                    <td>{stock}</td>
                </tr>
                <!-- END Part -->
            </tbody>
             <tfoot>
                <tr>
                    <td class = "text-center" colspan="9">AllParts</td>
                </tr>
            </tfoot>
        </table>
  </div>

<!—- PUT THE SHARED CODE 3.4.2 HERE -->
</body>
</html>
```

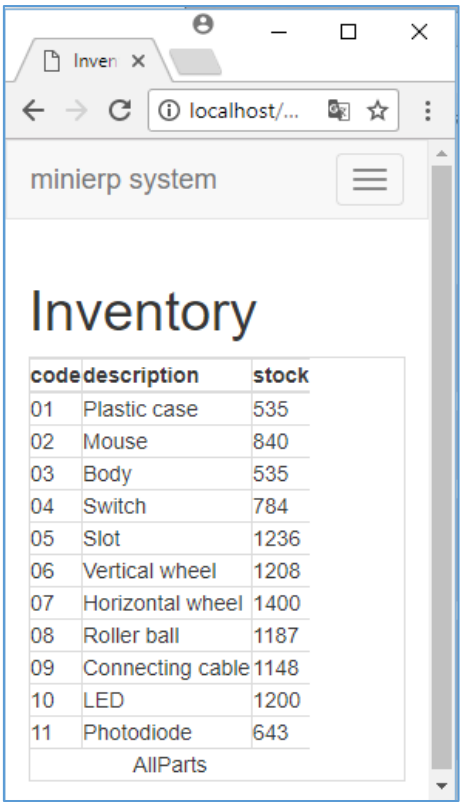
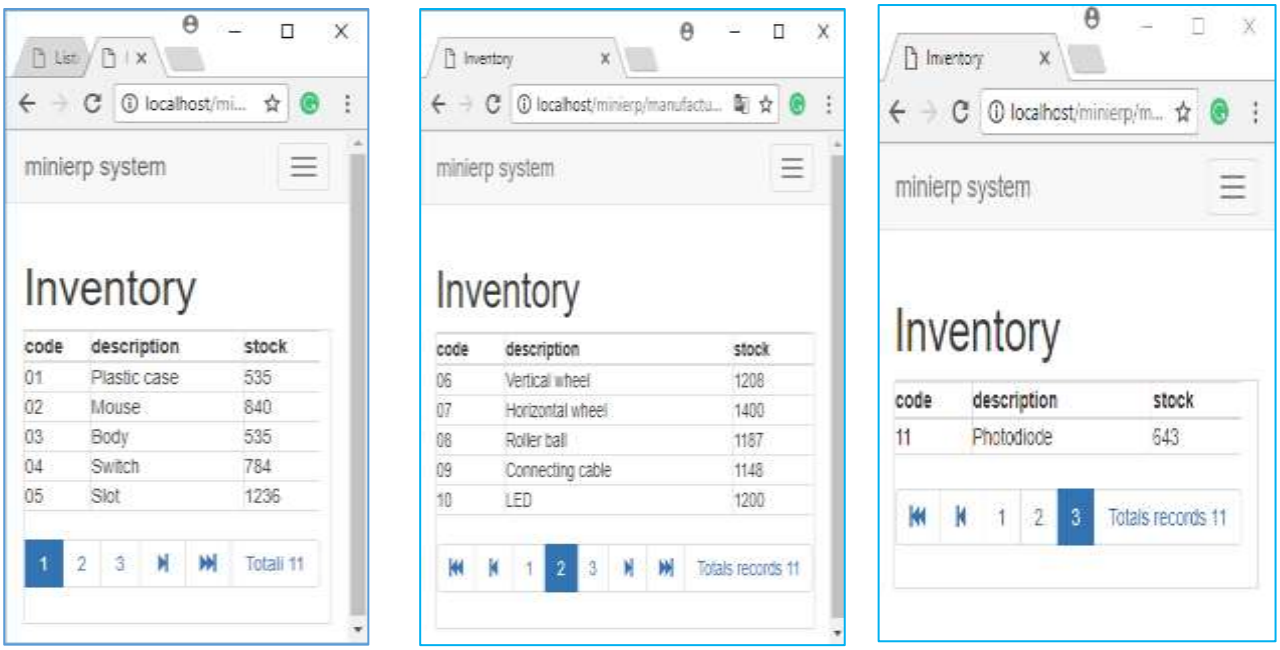


Fig. 7.2. The output of `http://localhost/minierp/manufacturing/inventory_data_repeater`.

7.2 Pagination

The output of fig. 7.2 shows a list of records starting from the inventory table. Because the table stores only eleven records, the output fits the web page. However, it is more frequent the situation where we have much more records to show. In such a case, the programmer should provide a solution to navigate the file showing one page at a time. This not trivial task usually occurs many times even in the context of a single software application. Luckily, WebMVC offers the component `Paginator` that makes easier the building of paginated queries.
The following example introduces the MVC entity `InventoryPaginator` necessary to implement a solution that shows page by page a list of goods from the inventory table of the `minierp` system. Apart the list of goods, the page contains a navigation bar and a paginator designed to manage the user interface. `Paginator` is an already available component designed for reuse; it is implemented, just like the other components provided by the framework, as an MVC unit. In fig. 7.3.a) the initial page retrieved by the paginator is shown. Then, when the scenario changes because of the user interaction, the graphical aspect provided for the pagination changes accordingly as shown in fig. 7.3.b) and 7.3.c).



c) Initial page. b) Intermediate page. c) Last page.

Fig. 7.3. The output of `http://localhost/minierp/manufacturing/inventory_paginator`.

The supervisor MVC unit `InventoryPaginator` coordinates the work of model and view. It also coordinates the MVC unit `NavigationBar` and the components `DataRepeater` and `PaginatorBootstrap`; the class `PaginatorBootstrap` is charged to handle the GUI that enhance the graphical aspect during the user interaction. As we shall see in the example 8.2.d, `PaginatorBootstrap` extends the component `Paginator` that has the responsibility to build the paginated query. In summary, the MVC unit `InventoryPaginator` aggregates the following parts:

```
InventoryPaginator =
 { controllers\manufacturing\InventoryPaginator,          // ex. 7.2.a
   model\manufacturing\InventoryPaginator,                // ex. 7.2.b
   views\manufacturing\InventoryPaginator,  //ex. 7.1.b (use InventoryPaginator instead of
                          // UserList and $tplName=”/manufacturing/inventory_paginator”;)
   templates\manufacturing\inventory_paginator,           // ex. 7.2.c
   {manufacturing\NavigationBar},              // the MVC unit NavigationBar (hierarchy)
   framework\components\DataRepeater,
   framework\component\bootstrap\PaginatorBootstrap  // ex 7.2.d graphics for the pagination
 }
```

Before going into the details of the code that implements the solution for the pagination problem, it is convenient to show in fig. 7.4 the structure of the involved classes. The class diagram emphasizes both the role of the controller `InventoryPaginator` that acts as a coordinator, and the hierarchy of required components. For the sake of simplicity, we show only used methods in the component hierarchy; you can find the details of the solution in the examples from 7.2.a to 7.2.e.

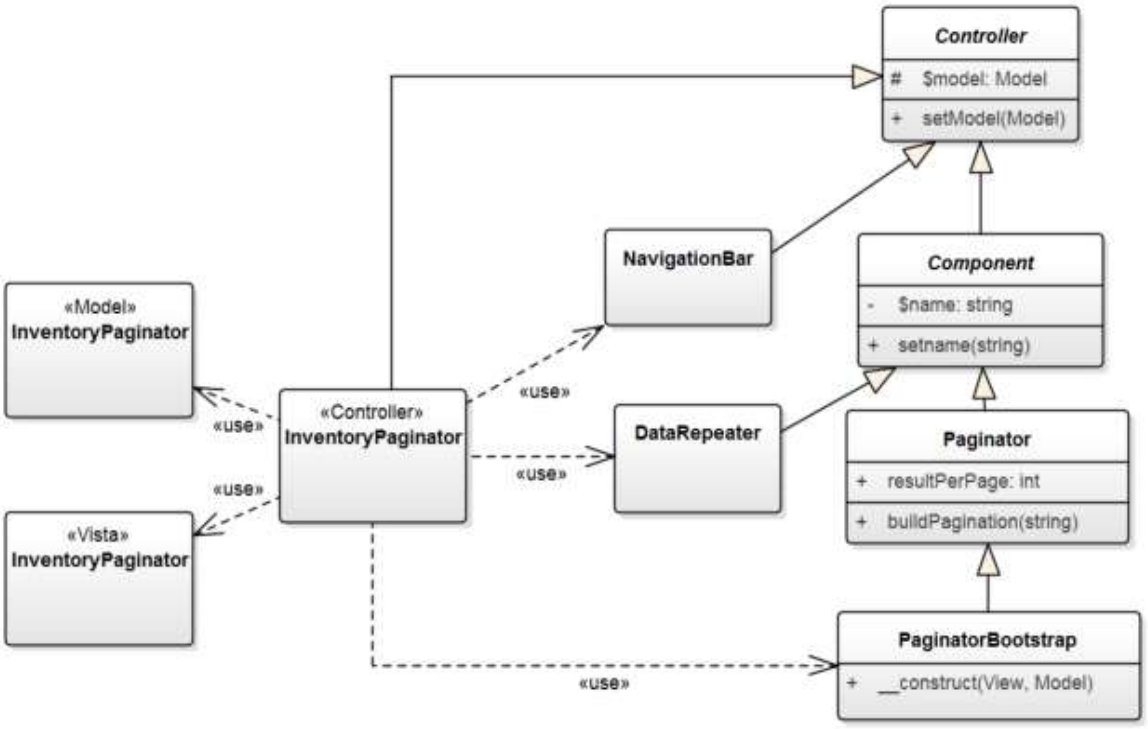


Fig. 8.4. The structure of the InventoryPaginator MVC unit: UML class diagram.

The constructor of `InventoryPaginator` does the usual arrangements in order to predispose the structure for computation and user interaction. In the `autorun()` method, the following computational steps are performed:

1. the instance `$paginator` of `PaginatorBootstrap` is created; the constructor loads the corresponding template;

2. the method `setName()`, inherited by the abstract class `Component`, sets the variable `$name` to the value `Bottom`. This value also appears in the template `8.2.c` in order to bind the component;

3. the inherited attribute `resultPerPage` of `$paginator` is initialized to 5 to indicate the number of records to show in a page; this value can be changed to adapt the visualization to your needs;

4. the attribute `$model` of `$paginator` , inherited by the class `Controller`, is set to the model to use (cfr ex. 8.2.b) by `setModel();`

5. the method `buildPagination()`, inherited by the component `Paginator`, paginates the query 5 records at a time;

6. the `DataRepeater` component is created for the displaying of a data coming from the model in the template `manufacturing\inventory_paginator;`

7. the method `bindComponent()` is called twice:

   - to populate the template with data coming from the execution of the query;

- to render the graphic that enable the pagination through user interaction.

```
// example 7.2.a: The InventoryPaginator controller. It uses: 1) a NavigationBar to
// navigate the minierp application; 2) the components DataRepeater and
// PaginatorBootstrap to show the DB inventory table one page at a time.

<?php

namespace controllers\manufacturing;

use framework\Controller;
use framework\Model;
use framework\View;

use models\manufacturing\InventoryPaginator as InventoryPaginatorModel;
use views\manufacturing\Inventory as InventoryView;
use controllers\manufacturing\NavigationBar;
use framework\components\DataRepeater;
use framework\components\bootstrap\PaginatorBootstrap;

class InventoryPaginator extends Controller
{

    public function __construct()
    {
        $this->model = $this->getModel();
        $this->view = $this->getView();
        parent::__construct($this->view,$this->model);
        $navigation = new NavigationBar();
        $this->bindController($navigation);
    }

    protected function autorun($parameters = null)
    {
        $paginator = new PaginatorBootstrap();

        $paginator->setName("Bottom");
        $paginator->resultPerPage = 5;
        $paginator->setModel($this->model);
        $paginator->buildPagination();

        $parts = new DataRepeater($this->view, $this->model, "Part", null);

       $this->bindComponent($parts);
       $this->bindComponent($paginator);

    }

    public function getView()
    {
        $view = new InventoryView("/manufacturing/inventory_paginator");
        return $view;
    }

    public function getModel()
    {
//        $model = new InventoryModel();
    $model = new InventoryPaginatorModel();
```

```
        return $model;
    }
}
```

```
// example 7.2.b: The InventoryPaginator model. It declares the SQL query to
// retrieve data from the inventory table.

<?php

namespace models\manufacturing;

use framework\Model;

class InventoryPaginator extends Model
{
    public function __construct()
    {
        parent::__construct();
        $this->sql =
<<<SQL
                SELECT
                   code,
                   description,
                   stock
                FROM
                inventory
SQL;
        $this->updateResultSet();
    }
}
```

```
// example 7.2.c: The template inventory_paginator for DB data rendering and GUI.

<!DOCTYPE html>
<html>
<head>
    <title>Inventory Pagination</title>
    <!-- PUT THE SHARED CODE 3.4.1 HERE -->
</head>

<body>
{Controller:manufacturing\NavigationBar}
<div class="container">
    <h1>Inventory</h1>
    <div class="table table-responsive">
        <table class="table-bordered">
            <thead>
                    <th>code</th>
                    <th>description</th>
                    <th>stock</th>
            </thead>
            <tbody>
                <!-- BEGIN Part -->
                <tr>
                    <td>{code}</td>
                    <td>{description}</td>
                    <td>{stock}</td>
```

```
                </tr>
                <!-- END Part -->
            </tbody>
            <tfoot>
            <tr>
                <td class = "text-center" colspan="9">{PaginatorBootstrap:Bottom}</td>
            </tr>
            </tfoot>
        </table>
    </div>

</div>

<!-- PUT THE SHARED CODE 3.4.2 HERE -->
</body>
</html>
```

The controller `InventoryPaginator` indirectly uses the component `Paginator` through the component `PaginatorBootstrap`, that is, in its turn, an MVC unit. To get an idea of how the graphical aspect for the pagination are managed, we report the code of `PaginatorBootstrap` and of the corresponding template that you can find in the following directory:

`framework/resources/components/bootstrap/paginator`

```
// example 7.2.d: The class PaginatorBootstrap extends the component Paginator. It
// introduces the graphical aspects to improve the GUI. The work for the query
// pagination is done by the Paginator component.

<?php

namespace framework\components\bootstrap;
use framework\components\Paginator;
use framework\View;

class PaginatorBootstrap extends Paginator
{
    public function __construct(View $view = null, Model $model = null)
    {

        if ($view == null) {
            $tpl = "framework/resources/components/bootstrap/paginator";
            $view = new View();
            $view->loadCustomTemplate($tpl);
        }

        parent::__construct($view,$model);

        // Bootstrap customizations
        $this->previous =   "glyphicon glyphicon-step-backward";
        $this->next =       "glyphicon glyphicon-step-forward";
        $this->first =      "glyphicon glyphicon-fast-backward";
        $this->last =       "glyphicon glyphicon-fast-forward";
        $this->offModeHidden = true;
        $this->offValue = "nav-item hidden";
        $this->activeFlag = "nav-item active";
```

```
    }

}
```

```
// example 7.2.e: The template paginator used by the component PaginatorBootstrap. The
blocks are shown/hidden accordingly to the user interaction (cfr. Fig. 8.3).

<!-- BEGIN Pagination -->
<nav>
    <ul class="pagination">

        <!-- BEGIN First -->
        <li class="{First_Off}">
            <a href="{First_URL}">
                <span aria-hidden="true" class="{First_On}"></span>
            </a>
        </li>
        <!-- END First -->

        <!-- BEGIN Prev -->
        <li class="{Prev_Off}">
            <a href="{Prev_URL}" aria-label="Previous">
                 <span aria-hidden="true" class="{Prev_On}"></span>
            </a>
        </li>
        <!-- END Prev -->

        <!-- BEGIN Pages -->
            <li class="{is_active}">
                <a href="{Page_URL}">
                    <span aria-hidden="true">{Page_Number}</span>
                </a>
            </li>
        <!-- END Pages -->

        <!-- BEGIN Next -->
        <li class="{Next_Off}">
            <a href="{Next_URL}" aria-label="Next">
                <span aria-hidden="true" class="{Next_On}"> </span>
            </a>

        </li>
        <!-- END Next -->

        <!-- BEGIN Last -->
        <li class="{Last_Off}">
            <a href="{Last_URL}">
                <span aria-hidden="true" class="{Last_On}"></span>
            </a>
        </li>
        <!-- END Last -->

        <!-- BEGIN Total -->
        <li>
            <span aria-hidden="true">{RES:TotalRecords} {TotalRecords}</span>
        </li>
        <!-- END Total -->
```

```
    </ul>
</nav>
<!-- END Pagination -->


</html>
```

8. Internationalization & Localization

The following definitions of internationalization and localizations are taken by the business dictionary [www.businessdictionary.com]:

Internationalization:

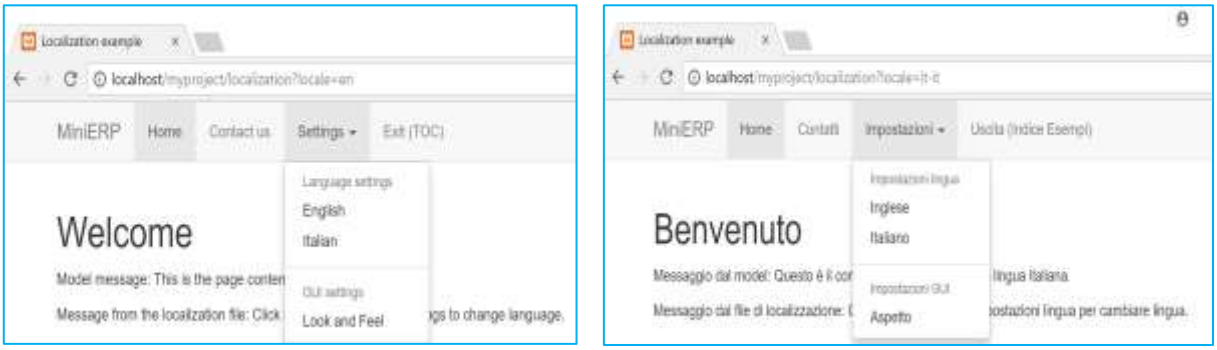
1) Commerce: The growing tendency of corporations to operate across national boundaries.
2) Marketing and Computing: An approach to designing products and services that are easily adaptable to different cultures and languages.

Localization:

The practice of adjusting a product's functional properties and characteristics to accommodate the language, cultural, political and legal differences of a foreign market or country.

Like other frameworks, WebMVC provides support to write software applications aiming at reaching a larger audience by means of internationalization and localization. The definition 2) of internationalization is interpreted in WebMVC how the capability to build, with little effort, the GUI of a software in different natural languages. This capability allows the presentation of your application to people of different nations or with specific visualization requirements.
The term localization is the counterpart of internationalization. Having a product or service that is ready for the international market, with the term localization we refer to the process that adapt a product or service to meet the needs of a language, culture or desired population's "look-and-feel". From one side, we can say that the internationalization focuses on the structure of your application because it builds several static contents ready to serve people of different nations or culture. On the other side, the localization process places a user within a context that is near, familiar and easy to use. It can be regarded as a dynamic aspect that restrict the wide context of a multilinguistic application to the smaller context suitable for the user. It is trivially to observe that the most important part of a localization process is the translation of a word or a text from one language to another, but localization is a bit more. For example, a message could be written in a completely different manner when we write it for a nation rather than another one, even if the message conveys the same semantics. These aspects are considered by WebMVC by means of a standard way to build

multilanguage applications. In example 8.1, we show an MVC Unit, named `Localization`, that manages a GUI capable to shift language from a page written in English to a page written in Italian:



a) Welcome page. c) Pagina di benvenuto

Fig. 8.1. Welcome page for the application `minierp`: a) english version b) italian version.

WebMVC manages the technicalities of internationalization/localization providing a folder `locales` where resource files containing the presentation content can be placed in different subfolders, one for each natural language. Again, the folder `locales` has to reflect the structure of system decomposition that we made for our project `myproject/controllers`. Because the folder `myproject` is simple (with no subsystems) the structure of `myproject/locales` does not contains application subsystems. In fig. 8.2, the directories `en` and `it-it` contain the resource files for the translation of the content shown in fig. 8.1. In particular, for the welcome page of fig. 8.1.a), the file

```
locales/en/application.txt
```

contains the translation of the `Welcome` word, whereas the file

```
locales/en/controllers/Localization.txt
```

contains a list of the *resource identifiers* that will be used to translate the page content. For example, the resource identifier `InfoMessage` is linked to the value `"Message from the localization file: Click Settings->Language settings to change language"` that will replace the placeholder `{RES:InfoMessage}` appearing in the `templates/localization` file. Note that both pages of fig. 8.1 also convey a message coming from the model according to the current language setting for the browser.

Fig. 8.2. The structure of the folder `myproject/locales`.

The MVC unit `Localization` aggregates the following parts:

```
Localization =
 { controllers\Localization,                                  // ex. 8.h
   model\Localization,                                        // ex. 8.e
   views\Localization,                                        // ex. 8.f
   templates\localization,                                    // ex. 8.g

   locales\en\application.txt,                    // ex. 8.a the “Welcome” string
   locales\en\controllers\Localization.txt,       // ex. 8.b English translation for the
                                                         controller Localization
   locales\it-it\application.txt,                // ex. 8.c the “Benvenuto” string
   locales\en\controllers\Localization.txt,      // ex. 8.d Italian translation for the
                                                         controller Localization
   framework\classes\Locale            // The WebMVC class that Manages localization files.
 }
```

The code of examples 8.a and 8.b shows the resource files for the localization to the English language:

```
// example 8.a: The file “locales\en\application.txt” contains the text:

Welcome=Welcome

// example 8.b: The file “locales\en\controllers\Localization.txt” contains the text:

#Comment:Tranlactions for the controller localization
ProjectName=MiniERP
Contacts=Contacts
Setting=Settings
LanguageSettings=Language settings
English=Englilsh
Italian=Italian
GuiSettings=GUI settings
LookAndFeel=Look and Feel
Exit=Exit
InfoMessage=Message from the localization file:
            Click Settings->Language settings to change language.
```

The resource files for the localization to the Italian language:

```
// example 8.c: The file locales\it-it\application.txt contains the text:

Welcome=Benvenuto

// example 8.d: The file “locales\it-it\controllers\Localization.txt” contains:

#Comment:Traduzioni per la localizzazione del controller
ProjectName=MiniERP
Contacts=Contatti
Setting=Impostazioni
LanguageSettings=Impostazioni lingua
English=Inglese
Italian=Italiano
GuiSettings=Impostazioni GUI
LookAndFeel=Aspetto
Exit=Uscita
InfoMessage=Messaggio dal file di localizzazione:
            Click Impostazioni->Impostazioni Lingua per cambiare lingua.
```

The remaining parts of the MVC Unit `Localization` necessary to the rendering of fig. 8.1 are described below. Assuming that we have an application that retrieves data from a multilanguage database, the model of example 8.e simulates the multilanguage database using the associative array `$bodiesDb`; in the example it contains two messages in English and Italian respectively.

```
// example 8.e: The Localization model. It simulates a multilanguage database and
returns a message in the language chosen for the deployment of a software application

<?php
namespace models;
use framework\Model;

class Localization extends Model
{
  private $pageBodies;

  public function __construct()
  {                              // Simulate a multi language database
    $bodiesDb = array(
      "it-it" => "Messaggio dal model: Contenuto della pagina per la lingua italiana.",
       "en"  =>  "Model message: This is the page content for english language");
    $this->pageBodies = $bodiesDb;
  }

    public function getBody($locale)
    {
      if (@$_REQUEST[LOCALE_REQUEST_PARAMETER])
            $locale = $_REQUEST[LOCALE_REQUEST_PARAMETER];
      return $this->pageBodies[$locale];
```

```
    }
}
```

The file `wiew\Localization` file:

```
// example 8.f: The Localization view. It loads the template “localization” and sets the message retrieved by the model.

<?php

namespace views;

use framework\View;

class Localization extends View
{
    public function __construct($tplName = null)
    {
        if (empty($tplName))
            $tplName = "localization";
        parent::__construct($tplName);
    }

    public function setVarBodyMessage($value)
    {
        $this->setVar("BodyMessage",$value);
    }

}
```

Each line of a localization file has the form (*resource identifier*=*value*). During the execution of a controller (for example the controller `Localization`), WebMVC will take the resource identifiers of its localization file one at a time and will proceed to the text substitution for the matching placeholders in the template. Each placeholder has the form {`RES`:*variable*}; therefore, for the *resource identifier* that matches a v*ariable*, the *value* string will replace the whole placeholder. For example, the file of example 8.d

`locales\it-it\controllers\Localization.txt`

contains the line `Contacts=Contatti` that matches the placeholder `{RES:Contacts}` in the file `templates\localization` of example 8.g. When the controller `Localization` runs, WebMVC modifies the template writing the string `Contatti` in place of `{RES:Contacts}` according to the current setting about the natural language to use.

```
<-- ex. 8.g the template “localization”. It contains the placeholders with general
    format {RES:variable} where variable will be replaced by a string coming from the
    locale file containing the translation -->
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Localization example</title>
    <!-- PUT THE SHARED CODE 3.4.1 HERE -->
</head>
<body>

<nav class="navbar navbar-default">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed" data-
toggle="collapse" data-target="#navbar" aria-expanded="false" aria-controls="navbar">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="#">{RES:ProjectName}</a>
        </div>
        <div id="navbar" class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li class="active"><a href="">Home</a></li>
                <li><a href="#about">{RES:Contacts}</a></li>
                <li class="dropdown">
                  <a href="#" class="dropdown-toggle" data-toggle="dropdown"
                   role="button" aria-haspopup="true" aria-expanded="false">{RES:Setting}

 <span class="caret"></span></a>
                    <ul class="dropdown-menu">
                        <li class="dropdown-header">{RES:LanguageSettings}</li>
                        <li><a href="?locale=en">{RES:English}</a></li>
                        <li><a href="?locale=it-it">{RES:Italian}</a></li>
                        <li role="separator" class="divider"></li>
                        <li class="dropdown-header">{RES:GuiSettings}</li>
                        <li><a href="">{RES:LookAndFeel}</a></li>
                    </ul>
                </li>
                <li><a href="..">{RES:Exit}</a></li>
            </ul>
        </div>
    </div>
</nav>

<div class="container">
    <h1>{RES:Welcome}</h1>
    <p>{BodyMessage}</p>
```

```
    <p>{RES:InfoMessage}</p>
</div>

<!-- PUT THE SHARED CODE 3.4.2 HERE -->
</body>
</html>
```

Finally, the code of the `Localization` controller:

```
// example 8.f: The Localization controller. It uses the method getCurrentLocale() of the framework class Locale to initialize the LCID (for example “en” or “it-it”), then coordinates the model and the view. The user localization files are managed behind the scene by the class Locale.

<?php
namespace controllers;
use framework\classes\Locale;
use framework\Controller;
use framework\Model;
use framework\View;
use models\Localization as LocalizationModel;
use views\Localization as LocalizationView;

class Localization extends Controller
{
    protected $view;
    protected $model;

    public function __construct(View $view=null, Model $model=null)
    {
        $this->view = empty($view) ? $this->getView() : $view;
        $this->model = empty($model) ? $this->getModel() : $model;
        parent::__construct($this->view,$this->model);
    }

    protected function autorun($parameters = null)
    {
        $locale = new Locale();
        $currentLocale = $locale->getCurrentLocale();
        $body = $this->model->getBody($_SESSION["CurrentLocale"]);
        $this->view->setVarBodyMessage($body);
    }

    public function getView()
    {
        $view = new LocalizationView("localization");
        return $view;
    }
```

```
    public function getModel()
    {
        $model = new LocalizationModel();
        return $model;
    }
}
```

9 Security

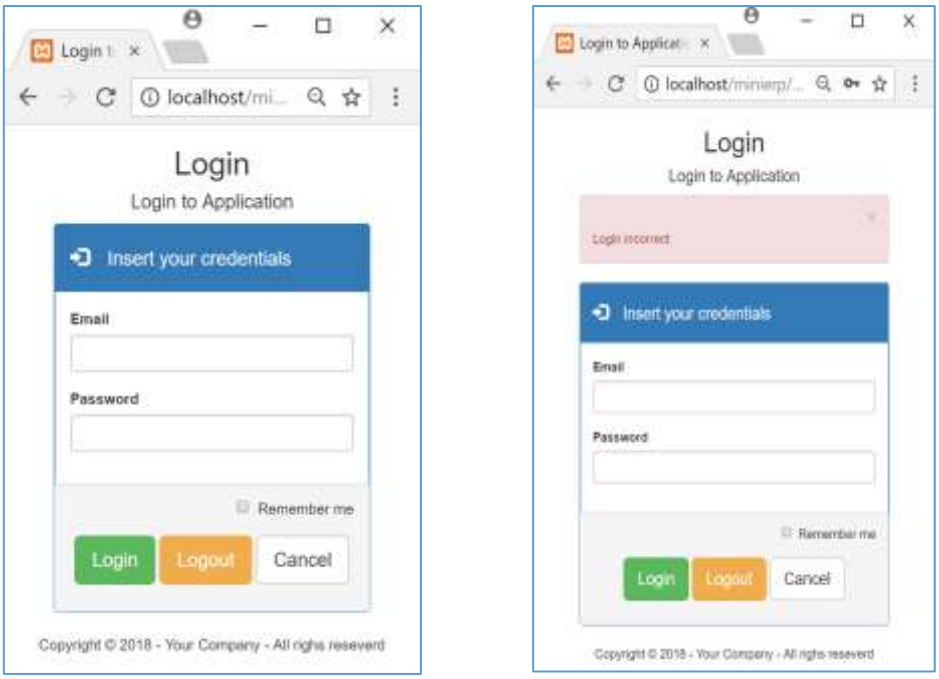
Several functionalities concerning the security of a software application are provided by WebMVC. They concern the user authentication, the Role Based Access Control (RBAC) and … In section 9.1 and 9.2 we shall discuss the framework classes for user authentication and RBAC that a developer can reuse to assign the rights to use functionalities of a web application.

9.1 Authentication

The user authentication can be done reusing the following `Login` MVC unit:

```
Login =
 { controllers\common\Login,                        // the Login controller;
   framework\User,                                // ex. 9.1.b: the model with registered users
   views\common\Login,                              // ex. 9.1.c: the Login view;
   templates\common\login,                         // ex. 9.1.a: the login template;

   locales\en\controllers\common\Login.txt,        // English translation for the
                                                      controller Login;
   locales\it-it\controllers\common\Login.txt,     // Italian translation for the
                                                      controller Login;
   framework\classes\ChipherService    // The WebMVC class that …
```

In fig. 9.1.a) we can see the state of a login form before the input of user credentials while in fig 9.1.b) the form also shows an incorrect login message. The examples of this section use the localization technique discussed in the previous chapter.



a) The login form b)The form after an incorrect input

Fig. 9.1. The login form: a) before the input of user credentials, b) after the input of wrong credentials.

From one side, the visualization of fig. 9.1.a) is possible thanks to the hiding of the block `LoginErrorMessage` appearing in the `templates/common/login.html.tpl` managed by the code of `views/common/Login`. On the other side, when the login is not successful, the `Login` controller shows the "Login incorrect" message retrieving it from the file `locales/en/controllers/common/Login` and substituting the message to the placeholder `{RES:LoginError}` in the code of example 9.1.a.

```
// Example 9.1.a. The template login.html.tpl for the login to an application.

<!DOCTYPE html>
<html>
<head>
    <title>{RES:LoginPageTitle}</title>

    <!-- PUT THE SHARED CODE 3.4.1 HERE -->

    <link href="{GLOBAL:SITEURL}/js/spinner/spinner.css" rel="stylesheet">

</head>

<body>
<div class="col-md-12 text-center">
        <h2>Login</h2>
        <h4>{RES:LoginPageTitle}</h4>
    <h5 class="text-danger">{LoginWarningMessage}</h5>
</div>
```

```
<div class="col-sm-4"></div>

<div role="main" class="col-sm-4 center-block">

  <!-- BEGIN LoginErrorMessage -->
    <div class="alert alert-danger alert-dismissible col-sm-12" role="alert">
      <button type="button" class="close" data-dismiss="alert"><span aria-
      hidden="true">×</span><span class="sr-only">Close</span></button>
      <br>{RES:LoginError}
    </div>
  <!-- END LoginErrorMessage -->

  <form role="form" id="login_form" class="form" method="post" name="login_form">
    <div class="panel panel-primary">
      <div class="panel-heading">
        <h4><span aria-hidden="true" class="glyphicon glyphicon-
                                            login"> </span>{RES:LogiFormTitle}</h4>
      </div>
      <div class="panel-body">

        <div class="form-group">
          <label for="LoginFormemail">Email</label>
          <div>
            <input type="text" id="LoginFormemail" class="form-control" maxlength="100"
                   value="" name="email" required>
          </div>
        </div>

        <div class="form-group">
          <label for="LoginFormpassword">Password</label>
          <div>
            <input type="password" id="LoginFormpassword" class="form-control"
                   maxlength="100" value="" name="password" required>
          </div>
        </div>

      </div>

      <div class="panel-footer text-center">
        <div class="form-group text-right">
          <input type="checkbox" id="remember_me" class="form-cotrol" value="1"
                 name="remember_me" >  {RES:RememberMeText}
        </div>
      <div class="form-group">
        <input class="btn btn-success btn-lg" type="submit" id="login_form_do_login"
               class="Button" alt="{RES:LoginButtonCaption}"
               value="{RES:LoginButtonCaption}" name="login_form_do_login">

        <!-- <input class="btn btn-warning btn-lg" type="submit"
            id="login_form_do_logout"  class="Button" alt="{RES:LogoutButtonCaption}"
            value="{RES:LogoutButtonCaption}"
            name="login_form_do_logout" formnovalidate> -->

         <input class="btn btn-default btn-lg" type="submit" id="login_form_do_cancel"
                class="Button" alt="{RES:CancelButtonCaption}"
             value="{RES:CancelButtonCaption}" name="login_form_do_cancel"
             onclick="history.back()" formnovalidate>
        </div>
      </div>
    </div>
```

```
  </form>

</div>
<div id="divLoading"></div>
<div class="col-sm-4"></div>
<div class="col-md-12 text-center">
        Copyright © {RES:CopyRightInfo}
</div>

<!-- PUT THE SHARED CODE 3.4.2 HERE -->
<script src="{GLOBAL:SITEURL}/js/spinner/spinner.js"></script>
</body>
</html>
```

The class `framework\User` has the responsibility to determine if a user has the right to get access to a software application. It uses the table `user` stored in the database that you declared in the `framework/config` file, and acts as a model in the `Login` MVC unit. The designer choice here has been to reuse the model class `framework/User` already available from WebMVC. However, if the application domain requires a different Login MVC unit, you can write your own version.

```
// ex. 9.1.b: The model for the Login MVC unit.

<?php
namespace framework;

use framework\classes\ChiperService;

class User extends MySqlRecord implements BeanUser
{
    private $userTable;
    private $fieldUserId;
    private $fieldUserEmail;
    private $fieldUserPassword;
    private $fieldUserRole;
    private $id;
    private $email;
    private $password;
    private $role;
    private $useMd5Password;

    public function getId()
    {
        return $this->id;
    }

    public function getEmail()
    {
        return $this->email;
    }

    public function getPassword()
    {
```

```
        return $this->password;
    }

    public function getRole()
    {
        return $this->role;
    }

    public function __construct($email=null, $password=null, $useMd5Password=true)
    {
        parent::__construct();
        $this->userTable = USER_TABLE;
        $this->fieldUserId = USER_ID;
        $this->fieldUserEmail = USER_EMAIL;
        $this->fieldUserPassword = USER_PASSWORD;
        $this->fieldUserRole = USER_ROLE;
        $this->useMd5Password = $useMd5Password;

        // If email and password are null try to set
        // them with cookie values.
        // $this->autoLoginFromCookies();


        if (isset($_SESSION["user"])) {
            $this->unserializeUser();
        } elseif ($email != null && $password != null) {
            $this->login($email, $password);
        }
    }

    public function login($email, $password)
    {
        $email = $this::real_escape_string($email);
        $password = $this::real_escape_string($password);
        // TODO use  PHP 5.4 password() crypt algo
        if ($this->useMd5Password)
            $password = md5($password);
        $sql = "SELECT * FROM {$this->userTable} WHERE {$this->fieldUserEmail}={$this-
>parseValue($email,'string')} AND {$this->fieldUserPassword}={$this-
>parseValue($password,'string')}";
        if (USER_ENABLED != "")
            $sql .=  " AND ". USER_ENABLED . "=1";
        $this->resetLastSqlError();
        $result = $this->query($sql);
        $this->resultSet = $result;
        $this->lastSql = $sql;
        if ($result) {
            $rowObject = $result->fetch_object();;
            $this->id = $rowObject->{$this->fieldUserId};
            $this->email = $rowObject->{$this->fieldUserEmail};
            $this->password = $rowObject->{$this->fieldUserPassword};
            $this->role = $rowObject->{$this->fieldUserRole};
```

```
            $this->serializeUser();
            return true;
        } else {
            $this->lastSqlError = $this->sqlstate . " - " . $this->error;
            return false;
        }

    }

    public function logout()
    {
        if (isset($_SESSION["user"])) {
            unset($_SESSION["user"]);
            $this->id = null;
            $this->email = null;
            $this->password = null;
            $this->role = null;
        }
        $chiper = new ChiperService();
        $secured = isset($_SERVER["HTTPS"]);
        setcookie($chiper::CREDENTIALS_COOKIE_NAME,"",time() - 3600,
"/",null,$secured,true);
        session_destroy();
        return true;
    }

    public function isLogged()
    {
        if (!empty($this->id) && !empty($this->email) && !empty($this->password)) {
            return true;
        } else {
            return false;
        }
    }

    public function checkForLogin($redirect = null, $returnLink= null,
$LoginWarningMessage=null)
    {
        $this->autoLoginFromCookies();
        $returnLink = (!empty($returnLink)) ? "?return_link=$returnLink": "";
        $LoginWarningMessage=(!empty($LoginWarningMessage)) ?
"&login_warning_message=$LoginWarningMessage": "";
        if (empty($redirect))
            $redirect = SITEURL . "/" . DEFAULT_LOGIN_PAGE;
        if (!$this->isLogged()) {
            header('Location: ' . $redirect . $returnLink . $LoginWarningMessage);
        }
    }

    public function autoLoginFromCookies()
    {
        if (!$this->isLogged()){
```

```
            $chiper = new ChiperService();
            $parts = $chiper->parseCredentialsCookie($chiper::CREDENTIALS_COOKIE_NAME);
            if (isset($parts) && (count($parts) > 2))
                list($username, $password, $expirationDate) = $parts;
            if ( !empty($expirationDate) && $expirationDate > time()) {
                if (!empty($username) && !empty($password)) {
                    $this->login($username, $password);
                    if ($this->isLogged())
                        $chiper->refreshCredentialsCookie($expirationDate);
                }
            }
        }
    }

    private function serializeUser()
    {
        $_SESSION["user"] = serialize($this);
        return true;
    }

    private function unserializeUser()
    {
        $user = unserialize($_SESSION["user"]);
        $this->id = $user->getId();
        $this->email = $user->getEmail();
        $this->password = $user->getPassword();
        $this->role = $user->getRole();
        return true;
    }
}
```

The view simply handle the login template and set the Login warning message either to the empty string in the case of successful login, or to the warning message in the case of incorrect login. (previously customized using config files located in config folder)

```
<?php
namespace views\common;
use framework\View;
class Login extends View
{
    public function __construct($tplName = null)
    {
        if (empty($tplName))
            $tplName = "/common/login";
        parent::__construct($tplName);
        $this->setLoginWarningMessage();

    }
    protected function setLoginWarningMessage()
    {
        if (isset($_GET["login_warning_message"])) {
            $warningMessage = $_GET["login_warning_message"];
        } else {
            $warningMessage = "";
```

```
        }
        $this->setVar("LoginWarningMessage", $warningMessage);
    }
}
```

The code of the controller ...

```
<?php

namespace controllers\common;

use framework\Controller;
use framework\Model;
use framework\View;

use framework\User as  LoginModel;
use framework\classes\ChiperService;
use views\common\Login as LoginView;

class Login extends Controller
{
    protected $view;
    protected $model;

    public function __construct(View $view=null, Model $model=null)
    {
        $this->view = empty($view) ? $this->getView() : $view;
        $this->model = empty($model) ? $this->getModel() : $model;
        parent::__construct($this->view,$this->model);
    }

    protected function autorun($parameters = null)
    {
        // Handles login form submission
        if (isset($_POST["login_form_do_cancel"])){
            header("Location: " . SITEURL);
        } else if (isset($_POST["login_form_do_login"])) {
            $this->configCookies();
            $email = $_POST["email"];
            $password = $_POST["password"];
            $this->model->login($email,$password);
            if ($this->model->isLogged()) {
               $this->hide("LoginErrorMessage");
               $returnPage = (isset($_GET["return_link"])) ?
                               SITEURL . "/" . $_GET["return_link"] :
                               SITEURL;
               header("Location:". $returnPage);
            }
        } else if (isset($_POST["login_form_do_logout"])) {
            $this->model->logout();
            header("Location:". SITEURL);
        } else {
            $this->hide("LoginErrorMessage");
        }
    }

    protected function configCookies()
    {
```

```
        if (isset($_POST["remember_me"])){
            $email = $_POST["email"];
            $password = $_POST["password"];
            if (!empty($email) && !empty($password)) {
                $chiper = new ChiperService();
                $chiper->setCredentialsCookie($email, $password);
            }
        }
    }

    public function getView()
    {
        $view = new LoginView("/common/login");
        return $view;
    }

    public function getModel()
    {
        $model = new LoginModel();
        return $model;
    }

}
```

The code of cipher service ...

```
<?php

namespace framework\classes;

class ChiperService
{
  const CREDENTIALS_COOKIE_SALT = CHIPER_CREDENTIALS_COOKIE_SALT;
  const CREDENTIALS_COOKIE_EXPIRATION_DATE = CHIPER_CREDENTIALS_COOKIE_EXPIRATION_DATE;
  const CREDENTIALS_COOKIE_SLIDING_EXPIRATION =
        CHIPER_CREDENTIALS_COOKIE_SLIDING_EXPIRATION;
  const CREDENTIALS_COOKIE_NAME = CHIPER_CREDENTIALS_COOKIE_NAME;

  private function cipherInit($key) {
        global $CipherBox, $CipherKey;
        $temp = '';
        $idx1 = 0;
        $idx2 = 0;
        $keyLength = strlen($key);
        for ($idx1 = 0; $idx1 < 256; $idx1++) {
            $CipherBox[$idx1] = $idx1;
            $CipherKey[$idx1] = ord($key[$idx1 % $keyLength]);
        }
        for ($idx1 = 0; $idx1 < 256; $idx1++) {
            $idx2 = ($idx2 + $CipherBox[$idx1] + $CipherKey[$idx1]) % 256;
            $temp = $CipherBox[$idx1];
            $CipherBox[$idx1] = $CipherBox[$idx2];
            $CipherBox[$idx2] = $temp;
        }
  }

  private function encryptString($inputStr, $key) {
        return strtoupper($this->bytesToHex($this->cipherEnDeCrypt($inputStr, $key)));
  }
```

```
private function decryptString($inputStr, $key) {
      return $this->bytesToString($this->cipherEnDeCrypt($this>hexToBytes($inputStr),
                                  $key));
}

public function encryptDBPassword($password) {
      return md5($password);
}

private function cipherEnDeCrypt($inputStr, $key) {
      global $CipherBox;
      $result = array();
      $i = 0;
      $j = 0;
      $this->cipherInit($key);
      for ($a = 0; $a < strlen($inputStr); $a++) {
          $i = ($i + 1) % 256;
          $j = ($j + $CipherBox[$i]) % 256;
          $temp = $CipherBox[$i];
          $CipherBox[$i] = $CipherBox[$j];
          $CipherBox[$j] = $temp;
          $k = $CipherBox[(($CipherBox[$i] + $CipherBox[$j]) % 256)];
          $crypted = ord($inputStr[$a]) ^ $k;
          $result[$a] = $crypted;
      }
      return $result;
}

private function bytesToString($bytesArray) {
      $result = '';
      foreach ($bytesArray as $byte) {
          $result .= chr($byte);
      }
      return $result;
}

private function bytesToHex($bytesArray) {
      $result = '';
      foreach ($bytesArray as $byte) {
          $tmp = dechex($byte);
          $result .= str_repeat("0", 2 - strlen($tmp)) . $tmp;
      }
      return $result;
}

private function hexToBytes($hexstr) {
      $result = '';
      $num = 0;
      for ($i = 0; $i < strlen($hexstr); $i += 2) {
          $num = hexdec(substr($hexstr, $i, 1)) * 16;
          $num += hexdec(substr($hexstr, $i + 1, 1));
          $result .= chr($num);
      }
      return $result;
}

private function chiperSetCookie($parameter_name, $param_value, $expired = -1,
                     $path = "/", $domain = "", $secured = false, $http_only = true)
  {
```

```
        $secured = isset($_SERVER["HTTPS"]);
        if ($expired == -1)
            $expired = time() + 3600 * 24 * 366;
        elseif ($expired && $expired < time())
            $expired = time() + $expired;
        setcookie ($parameter_name, $param_value, $expired,
                   $path, $domain, $secured, $http_only);
  }

  private function chiperGetCookie($parameter_name)
    {
        return isset($_COOKIE[$parameter_name]) ? $_COOKIE[$parameter_name] : "";
  }

  public function setCredentialsCookie($login, $password) {
     $login    = $this->encryptString($login, $this::CREDENTIALS_COOKIE_SALT);
     $password = $this->encryptString($password, $this::CREDENTIALS_COOKIE_SALT);
     $result   = $this->encryptString($login . ":" . $password . ":" . (time() +
           $this::CREDENTIALS_COOKIE_EXPIRATION_DATE), $this::CREDENTIALS_COOKIE_SALT);
     $this->chiperSetCookie($this:: CREDENTIALS_COOKIE_NAME, $result, time() +
                      $this::CREDENTIALS_COOKIE_EXPIRATION_DATE, "/", "", false,true);
  }

  public function refreshCredentialsCookie($expirationDate) {
   if ($this::CREDENTIALS_COOKIE_SLIDING_EXPIRATION)
    {
     if (($expirationDate - ($this::CREDENTIALS_COOKIE_EXPIRATION_DATE / 2)) > time())
       {
            list($login, $password, $expDate) =
            $this->parseCredentialsCookie($this:: CREDENTIALS_COOKIE_NAME);
            $this->setCredentialsCookie($login, $password);
       }
    }
  }

  public function parseCredentialsCookie($cookieName) {
        $cookieValue = $this->chiperGetCookie($cookieName);
        $decryptedCookieValue = (strlen($cookieValue)) ?
                $this->decryptString($cookieValue, $this::CREDENTIALS_COOKIE_SALT) :
                "";
        $pos = strpos($decryptedCookieValue, ':');
        $parts = array();
        if ($pos) {
           $parts = explode(":", $decryptedCookieValue);
           $parts[0] = $this::decryptString($parts[0],$this::CREDENTIALS_COOKIE_SALT);
           $parts[1] = $this::decryptString($parts[1], $this::CREDENTIALS_COOKIE_SALT);
        }
        return $parts;
  }
}
```

9.2 Role based access control

** to do **

4. The Benefits of WebMVC

Since its introduction, thanks to the work of Trygve Reenskaug on the MVC pattern for GUI software design, the pattern suggested the decoupling of model and view. In this way, it is possible to associate different views to a given model and display the same data in alternative formats or different devices. Since then, the technologies are changing but the MVC pattern remains valid and new considerations can be made when the pattern is used in the Web. In general, there are two kinds of advantages deriving from the use of the MVC pattern for the development of complex web applications: technological and organizational. In fact, the pattern does not impose any constraint on the use of technology and a designer can decide to use languages and systems that she considers most appropriate for software application to develop. A natural consequence of the decision about the use of an MVC pattern, is that the principle of specialization and coordination will be widely used in both: technological and organizational perspectives. In general, organizational benefits usually derive from a better organization of work through specialization and coordination. Specialization concerns the division of a work into smaller parts and their assignment to specialized workers. This is a standard practice in modern software development methodologies where the work can proceed in parallel in relatively little increments assigned to software developers and testing activities can be pursued as soon as possible. However, the specialization that comes from the division of work also requires coordination because what has been broken by specialization (subsystem decomposition) has to be reconducted to unity by coordination (system integration).

The advantages deriving from the division of work are evident also when we decide to use the MVC architectural pattern. The technologies used in each subsystem are homogeneous (e.g. HTML and Javascript for the View part, PHP for the Controller part and PHP / SQL for the Model part). The code reflects the separation of the professional skills necessary to deal with the various aspects of development as shown in table 6.1. The code is easier to design, implement, verify and maintain, and you can use pre-existing code where appropriate. The same information can be presented in several ways (e.g. textual/graphics) and on different devices. Note that the controller assumes the role of coordinator of view and model and this easy the system integration activity necessary to build a software system perceived as a unity.

Table 6.1. *Typical separation of skills and technologies for the development of Web MVC applications*.

Entity	Goals	Technology	Role/skill
View	• Defines GUI and formats for the interaction between users and GUI • Receives data input • Provides data output	HTML - The GUI definition can contain references to • CSS style sheets • Javascript code	*Web designer* with skills in the design and implementation of Web GUI. *Skills*: HTML, CSS, Javascript and related technologies
Controller	• Manages the sequence of interactions with the user • Knows how to divide the work • Coordinates the job of model and view • Coordinate the job of subordinate controllers	PHP	• *Analyst* • *Programmer* *Skill*: PHP programming
Model	• Maintains domain knowledge • Provide interfaces to data sources • Handle business rules and business logic	PHP, SQL	• *Domain expert* • *DB designer* • *Software engineer* *Skills*: Entity/Relationship modelling, PHP and SQL programming

References

[1] B. Brugge, A. H. Dutoit, "Object Oriented Software Engineering Using UML, Patterns, and Java (3rd Ed.), Pearson, 2014.

[2] Ian Sommerville, "Software Engineering (10e), Pearson, 2017.

[3] E. Gamma et al., "Design Patterns, Elements of Reusable Object-Oriented Software", Addison Wesley, 1994.