

# Views (DRAFT)

## lesson #basic04

**James L. Parry**  
**B.C. Institute of Technology**

---

## Agenda

---

1. [View Construction Strategies](#)
2. [View Assembly Strategies](#)
3. [HTML Form Handling](#)
4. [Data Transfer Buffer](#)
5. [File Uploading](#)
6. [Loose Ends](#)

---

## 1. VIEW CONSTRUCTION STRATEGIES

---

Views are all about presentation, driven by UI design. Even if we take the perspective that "design" is not a programmer's job, we still need to provide the view components with the data to present.

Most developers will use a CSS/Javascript framework to style the presentation nicely - integrating these into a webapp is part of next week's lesson.

---

## Our Target View Excerpt

---

Our views are data-driven, and we will look at several strategies to assemble a view with iteration, such as the table shown to the right. Each of the strategies that follow will end up with the same view in the browser - they differ in their approach server-side.

The view to the right shows a "typical" table, with one row per contact. The styling comes from default Bootstrap CSS.

Example			
#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
3	Larry	the Bird	@twitter

---

## Traditional View Construction

---

The code below is typical of the "traditional" PHP approach.

Controller...	views/contacts.php
<code>\$this-&gt;load-&gt;view('contacts');</code>	<code>&lt;table...&gt;</code>
	<code>&lt;?php foreach (\$contacts-&gt;... as \$record) {</code>
	<code>?&gt;</code>
	<code>&lt;tr&gt;</code>
	<code>&lt;td&gt;&lt;?php echo \$record-&gt;id ?&gt;&lt;/td&gt;</code>
	<code>&lt;td&gt;&lt;?php echo \$record-&gt;firstname ?&gt;&lt;/td&gt;</code>
	<code>&lt;td&gt;&lt;?php echo \$record-&gt;lastname ?&gt;&lt;/td&gt;</code>
	<code>&lt;td&gt;&lt;?php echo \$record-&gt;username ?&gt;&lt;/td&gt;</code>
	<code>&lt;/tr&gt;</code>
	<code>&lt;?php } ?&gt;</code>
	<code>&lt;/table&gt;</code>

---

## View Construction Using the Template Parser

---

This excerpt is typical of the "template parser" approach, which you saw in lesson 2.

Controller...	views/mycontacts.php
<code>\$rows = array();</code>	<code>&lt;table...&gt;</code>
<code>foreach (\$contacts-&gt;... as \$record)</code>	<code>{records}</code>
<code>\$rows[] = (array) \$record;</code>	<code>&lt;tr&gt;</code>
<code>\$parms['records'] = \$rows;</code>	<code>&lt;td&gt;{id}&lt;/td&gt;</code>
<code>\$this-&gt;parser-&gt;parse('mycontacts', \$parms);</code>	<code>&lt;td&gt;{firstname}&lt;/td&gt;</code>
	<code>&lt;td&gt;{lastname}&lt;/td&gt;</code>
	<code>&lt;td&gt;{username}&lt;/td&gt;</code>
	<code>&lt;/tr&gt;</code>
	<code>{/records}</code>
	<code>&lt;/table&gt;</code>

---

## View Construction Using View Fragments

---

The code below is typical of the "view fragment approach", which you also saw in lesson 2.

Controller...	views/mycontacts.php
<code>\$result = '';</code>	<code>&lt;table...&gt;</code>
<code>foreach (\$contacts-&gt;... as \$record)</code>	<code>{inside_stuff}</code>
<code>\$result .= \$this-&gt;parser-&gt;parse('just1',</code>	<code>&lt;/table&gt;</code>
<code>(array) \$record, true);</code>	
<code>\$parms['inside_stuff'] = \$result;</code>	views/just1.php...
<code>\$this-&gt;parser-&gt;parse('mycontacts', \$parms);</code>	<code>&lt;tr&gt;</code>
	<code>&lt;td&gt;{id}&lt;/td&gt;</code>
	<code>&lt;td&gt;{firstname}&lt;/td&gt;</code>
	<code>&lt;td&gt;{lastname}&lt;/td&gt;</code>
	<code>&lt;td&gt;{username}&lt;/td&gt;</code>
	<code>&lt;/tr&gt;</code>

# Comparing View Construction Strategies

The "traditional" approach is marginally more efficient, with one less layer of computation compared to using the template parser, but you end up with PHP scriptlets mixed up with HTML in your view. All iteration is done in the view, and the view accesses the model. This is **bad**.

The "template parser" approach has a much cleaner view, with substitution fields and no PHP. Iteration, however, appears in both the controller and the view. The controller part of the iteration results in building an associative array of associative arrays, which is not comfortable for everyone!

The "view fragment" approach has the simplest views, but at the expense of some complexity in the controller.

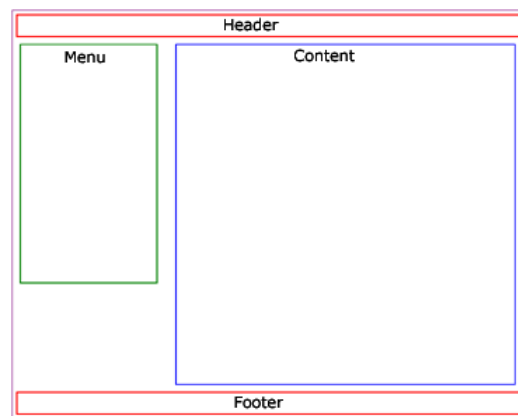
All of these are *\*legal\**, but the first will cost you marks!

## 2. VIEW ASSEMBLY STRATEGIES

View assembly refers to how you pull together your different view fragments for presentation, regardless of how you constructed them.

Most assembly is by using a wireframe layout, naming the panels in it, and then setting corresponding view parameters before rendering.

```
$this->data['header'] = makeheader(...);
$this->data['menu'] = makemenu(...);
$this->data['content'] = makecontent(...);
$this->data['footer'] = makefooter(...);
```

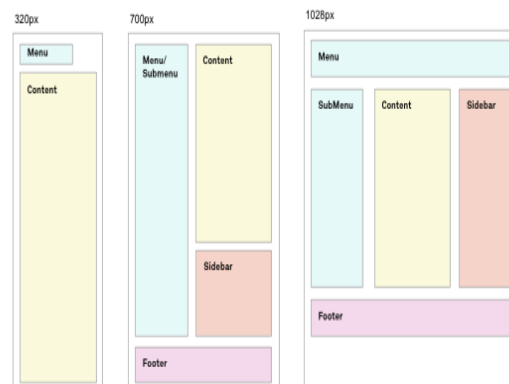


## Responsive Layout

With responsive layout, you would build your view components and assemble them with a desktop display in mind.

You would then use CSS, with media queries, to layout the components differently for different browser window sizes, or even to hide some of the panels at smaller resolution.

```
$this->data['menu'] = makemenu(...);
$this->data['submenu'] = makesubmenu(...);
$this->data['sidebar'] = makesidebar(...);
$this->data['content'] = makecontent(...);
$this->data['footer'] = makefooter(...);
```



## Multiple Layouts

Multiple layouts are often appropriate for a site.

In such a case, you could pass the desired layout as a view parameter, and inside your base controller's `render()` method choose the appropriate layout template to load. The layouts would share similar panel names, as appropriate.

```
$this->data['layout'] = 'secondary';
$this->data['slideshow'] = makecarousel(...);
$this->data['content'] = makecontent(...);
```



## 3. HTML FORM HANDLING

HTML form handling is really two separate issues:

- Handling submitted input, and
- Generating the form to be completed

We will deal with some strategies for each of these issues, in the above order.

## Sample Input to Handle

Refer to the [example-forms webapp](#) for \*some\* actual code (no guarantees that it is complete)!

A simple form, in pure HTML:

```
<form action="/dealwithme" method="post">
  Why don't they play poker in the jungle?<br>
  <input type="radio" name="jungle" value="treefrog"> Too many tree frogs.<br>
  <input type="radio" name="jungle" value="cheetah"> Too many cheetahs.<br>
  <input type="radio" name="jungle" value="river"> Too many rivers.<br>
  Check the box if you want your answer to be graded:
  <input type="checkbox" name="grade" value="yes"><br>
  <input type="submit" name="submit" value="Submit"><br>
</form>
```

The result of the form:

Why don't they play poker in the jungle?

- ☐ Too many tree frogs.  
☐ Too many cheetahs.  
☐ Too many rivers.

Check the box if you want your answer to be graded: ☐

Submit

## Handling Input Traditionally

The traditional PHP approach to handling form input is to use the `$_POST` superglobal, which gives access to the form fields. This would look something like...

```
if (!empty($_POST['grade'])) {
    if (!empty($_POST['jungle'])) {
        if ($_POST['jungle']=="cheetah") { echo "You got the right answer!"; }
        } else { echo "Sorry, wrong answer."; }
    } else { echo "You did not choose an answer."; }
} else { echo "Your answer will not be graded."; }
```

## Handling Input Using CodeIgniter

CodeIgniter provides an [Input class](#) to do the same job. The advantages of using it are input filtering, and easy access to input data whether it came from a form or was part of the URL. This would look something like...

```
if ($this->input->post('grade') !== false) {
    if ($this->input->post('jungle') !== false) {
        if ($this->input->post('jungle']=="cheetah") { echo "You got the right answer!"; }
        } else { echo "Sorry, wrong answer."; }
    } else { echo "You did not choose an answer."; }
} else { echo "Your answer will not be graded."; }
```

## What About Error Handling?

CodeIgniter provides a [Form Validation class](#) to do this. A sample excerpt using this would look something like...

```
$this->form_validation->set_rules('passconf', 'Password Confirmation', 'required');
$this->form_validation->set_rules('email', 'Email', 'required');

if ($this->form_validation->run() == FALSE) {    $this->load->view('myform');
} else {
    $this->load->view('formsuccess');
}
```

## What About Creating the Form?

An input form can be pretty simple, but ugly ... see below.

### Source Code:

```
<!DOCTYPE html>
<html>
<body>

<form action="">
<fieldset>
<legend>Personal information:</legend>
Name: <input type="text" size="30"><br>
E-mail: <input type="text" size="30"><br>
Date of birth: <input type="text" size="10">
</fieldset>
</form>

</body>
</html>
```

---

## Styling the Form?

---

The form can always be styled to look better, but that can get complicated, as shown below. Wait until next week's lesson :)

Your HTML...

```
<label for="name">Name</label>
<input name="name" type="text" id="name" class="textfield" />

<label for="phone">Phone</label>
<input name="phone" type="text" id="phone" class="textfield" />

<label for="email">Email Address</label>
<input name="email" type="text" id="email" class="textfield" />
```

The accompanying CSS...

```
label {
    clear: left; float: left;
    padding: 3px 10px 2px; text-align: right; width: 180px;
}
.textfield, .textarea, .selectlist {
    font-family: Arial,Helvetica,sans-serif;
    font-size: 12px; margin: 0 0 8px; width: 250px;
}
```

---

## Making Forms With CodeIgniter

---

CodeIgniter provides a [Form Helper](#) to make this a bit easier. This would look something like...

```
<?php
echo form_open('#');
echo form_fieldset('Personal information:');
$params = array('name'=>'name','size'=>30);
echo 'Name: ' . form_input($params) . '<br/>';
$params = array('name'=>'email','size'=>30);
echo 'E-mail: ' . form_input($params) . '<br/>';
$params = array('name'=>'born','size'=>10);
echo 'Date of birth: ' . form_input($params) . '<br/>';
echo form_fieldset_close();
echo form_close();
?>
...

```

There are better ways, which we will explore next week, so don't get hung up on this as *\*the way\** to create forms!

---

## 4. DATA TRANSFER BUFFER

---

The "data transfer buffer" design pattern says that you use a session object (associative array, in our case) to hold field values for form handling.

If a form was intended to hold a customer record, for instance, then you might use the "record" session object to hold a record's values during processing. You would update this to record in-progress input, so that the user does not have to re-enter a form from scratch if they make a mistake and you reject their input.

---

## Data Transfer Buffer Pseudo-code

---

Generally speaking, the logic to apply this pattern looks like...

### Form presenting:

- if no session object
  - create an empty one if adding a new record
  - create one with existing data if updating
- get the session object
- forward its values as view parameters

### Form handling:

- retrieve the session object, and merge the posted data into it
- validate the updated record
- if the data is valid, update your database and destroy the session object
- if the data isn't valid, update the session object, add error messages to the session and redisplay the form

---

## Sessions

---

The "data transfer buffer" design pattern relies on sessions, which are handled in CodeIgniter using the [Session library](#), which is a wrapper for PHP's native sessions.

Think of a session as a container for key/value pairs, like a `java.util.Map`

You will want to configure the sessions properly, using a database session driver. The SQL to create this is in the user guide, referenced above.

Autoload your session library!

---

## Getting Session Data

---

Session data can be retrieved several different ways, using the native provisions or using the Session library.

To retrieve a "record" stored in a session,

- `$record = $_SESSION['record'];` // use the PHP superglobal, or
- `$record = $this->session->record;` // use PHP's magic getter, or
- `$record = $this->session->userdata('record');` // use the Session library

The `userdata()` method returns NULL if the requested entry is not in the session, while the other techniques will trigger a PHP error.

---

## Setting Session Data

---

Session data can be set several different ways, using the native provisions or using the Session library.

To store a "record" as a session property,

- `$_SESSION['record'] = $record;` // use the PHP superglobal, or
- `$this->session->record = $record;` // use PHP's magic setter, or
- `$this->session->set_userdata('record',$record);` // use the Session library

---

## Session Flashdata

---

The CodeIgniter Session library also supports the notion of "flashdata", which is a session property that only exists until the next request. This could be a great way to hold error messages, or to hold tokens used to make sure that a form isn't submitted and handled twice.

Retrieve flashdata just like any other session data.

To set a flashdata property in a session,

- `$this->session->set_flashdata('errors', $error_messages);` // use the Session library, or
- `$_SESSION['errors'] = $error_messages;` // save the property normally
- `$this->session->mark_as_flash('errors');` // and flag it as flashdata

---

## 5. FILE UPLOADING

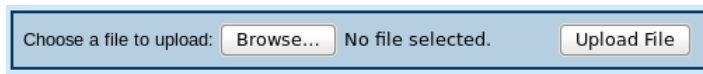
---

File uploading is a standard expectation of a webapp.

Your HTML form will need to use POST for submission, and you will need to specify an appropriate enclosure type, for instance...

```
<form enctype="multipart/form-data" action="/uploader" method="POST">
  Choose a file to upload:
  <input name="uploadedfile" type="file" /><br />
  <input type="submit" value="Upload File" />
</form>
```

This will result in something like




---

## Traditional File Uploading

---

With traditional PHP, uploaded files are accessed through the `$_FILES` superglobal.

- `$_FILES["file"]["name"]` - the name of the uploaded file
- `$_FILES["file"]["type"]` - the type of the uploaded file
- `$_FILES["file"]["size"]` - the size in bytes of the uploaded file
- `$_FILES["file"]["tmp_name"]` - the name of the temporary copy of the file stored on the server
- `$_FILES["file"]["error"]` - the error code resulting from the file upload

You provide error-checking.

Handling multiple file uploads is a topic for another day.

---

## Saving An Uploaded File

---

PHP provides a method to move the uploaded file from the system's temporary folder to a destination you specify:

```
move_uploaded_file($_FILES["file"]["tmp_name"],
  $target_folder . $_FILES["file"]["name"]);
```

You are responsible for proper file naming, for instance replacing spaces or undesirable characters with underscores.



# File Uploading With CodeIgniter

CodeIgniter provides a File Uploading library to make the job easier.


The user guide explanation is a bit clunky, but the gist is...

```
public function do_upload() {
    $config['upload_path'] = './uploads/'; $config['allowed_types'] = 'gif|jpg|png'; $config['max_size'] = 100;
    $config['max_width'] = 1024; $config['max_height'] = 768;
    $this->load->library('upload', $config);
    if ( ! $this->upload->do_upload() ) {
        $error = array('error' => $this->upload->display_errors());
        $this->load->view('upload_form', $error);
    } else {
        $data = array('upload_data' => $this->upload->data());
        $this->load->view('upload_success', $data);
    }
}
```

## File Uploading Widgets

Much better file upload widgets are available for your webpage HTML, eg. the Jasny file uploader (shown below), for Bootstrap.

Example

 boat-names-1.jpg
 

Change Remove

- `<div class="fileupload fileupload-new" data-provides="fileupload">`
- `<div class="input-append">`
- `<div class="uneditable-input span3"><i class="icon-file fileupload-exists"></i> <span class="fileupload-preview"></span></div><span class="btn btn-file"><span class="fileupload-new">Select file</span><span class="fileupload-exists">Change</span><input type="file" /></span><a href="#" class="btn fileupload-exists" data-dismiss="fileupload">Remove</a>`
- `</div>`
- `</div>`
-

## 6. LOOSE ENDS

What about some of the slick forms widgets? (date pickers, smart fields, etc)

Most CSS frameworks use jQuery (or similar), and there are lots of plugins/gadgets/widgets for them.

Be patient, young padawans... this is next week's lesson

---

## Menu Item Highlighting

---

Menu item highlighting is CSS framework dependent. Generally, add "active" class to the "current" item

The general idea is something like:

```
foreach($menuitems as $item) {  
    $active = $item->link == $this->uri->segment(0);  
    $thelist .= make_menu_item($item..., $active);  
}
```

---

## Image Handling

---

If you want to mess with images, for instance creating thumbnails, that is usually done with a library, and not necessarily as part of image uploading (though it could be). See [libraries/simpleimage](#) for an example.

Caution: make sure that any graphics libraries you need are part of your PHP configuration.

---

## Cross Site Scripting (XSS)!

---

Check the [security class](#) in CI ... it is meant to block cross-site scripting attempts, or inappropriate filenames.

Another approach: use the `htmlentities` and the `html_entity_decode` PHP functions to convert applicable characters to HTML entities instead. This would prevent someone from injecting a JS exploit into a text area, for instance.

---

## Congratulations!

---

You have completed lesson #basic04: Views (DRAFT)

If you would take a minute to [provide some feedback](#), we would appreciate it!

The next activity in sequence is: [forms](#) Example webapp - forms

You can use your browser's back button to return to the page you were on before starting this activity, or you can jump directly to the course [homepage](#), [organizer](#), or [reference](#) page.