Lab 4: Controller Classes

In this lab you will refactor your current app to full MVCS architecture. You will ensure that your queries are in Service and Model classes, your HTML output is in a View class, your logic is in a Controller file, and your validation is in a helper class. You will also use URL rewriting to implement Search-Engine-Friendly (RESTful) page URLs.

Objectives & Outcomes

Successful completion of this activity will show that you can:

- Refactor an aplication to use proper MVCS architecture.
- Apply the URL rewriting features of Apache to build SES/SEF URLs.

Level of Effort

This activity should take approximately 4h to complete. It will require:

- 30m Research
- 5m Prep & Delivery
- 3h Work

If you find that this activity takes you significantly less or more time than this estimate, please contact me for guidance.

Reading & Resources

Book

PHP Objects, Patterns, and Practice
Chapters:
11
Pages:
216-220
Author:
Matt Zandstra

Publisher:
Apress
Edition:
3 (2010)
A review of the Command Pattern from your OOP course would probably be very helpful.

Instructions

This lab is similar to the last one: you will develop data access solutions in both PHP and CFML, using the View classes you've already built. This lab will go in a `04_controller` folder.

Development

For this lab you will combine your controller pages into a single Command Pattern Controller for each language. That is, you will have an `index.php` and an `index.cfm` that handle the validation and routing logic for your application.

The easiest way to do this is to use URL rewriting, such as Apache's `mod_rewrite`:

```
RewriteEngine On
RewriteCond  %{REQUEST_FILENAME}  !-f
RewriteCond  %{REQUEST_FILENAME}  !-d
RewriteRule  ^([a-z]+)$  index.php?action=$1  [L,NS,QSA]
```

This `.htaccess` file will allow Apache to accept URLs that look like `whatever/userAdd` and use them as if they were `whatever/index.php?action=userAdd` allowing your URLs to not have to explicitly use the query string.

But once you do this, *all* of the page requests will be sent through your single index file. This means that you will need to include a fair bit of logic to handle every page in your application:

```
If (Action === "userAdd") Then
    FormErrors = FormHelper.ValidateElements "email",
"password", "fullname"
    If (FormErrors.length === 0) Then
        UserGateway.add(FormElements)
        SiteView.show "userAddOkay"
    Else
        SiteView.show "userAdd", "Please fix these errors:" +
FormErrors
    End If
Else If (Action === "userDelete") Then
    # etc, etc, etc
Else
    ShowView "home"
End
```

Reflect on the classes and objects you will need to implement this lab:

- **FormHelper:** Your forms will have common elements that will always need to be validated the same way. An email address should always look like an email address, for example. You can do this sort of validation by hand, over and over and over again, or you can put the common methods into a class.
- **UserGateway:** This is the *Service* part of the MVCS pattern, and gives you easy access to manipulate your data. Alternatively, you could implement a `UserService` that returns Data Access Objects instead of primitive data.

- **SiteView:** All of your HTML should be wrapped into a single View class, including templates from one folder. In larger applications you may implement a Facade pattern to use multiple views (such as: HtmlView, JsonView, XmlView, etc), but for now a single class will suffice.

By the end of the lab your User Management app should work and be a great example of MVCS architecture and the Command Pattern. You shouldn't have any HTML or queries outside of your Views and Services, respectively. Your app should be an Object-Oriented paradise.

Above & Beyond

If you want to get really fancy, you *could* implement a SiteController class that works like your SiteView class, but for Controller logic instead of View presentation. That is, your landing page could call a Controller method:

```
FormErrors = SiteController.before Action
If (FormErrors.length === 0) Then
    ViewName = SiteController.do Action
    SiteView.show ViewName
Else
    SiteView.show Action, FormErrors
End If
```

In this example, the `before` method handles any validation and returns any problems. If there aren't any problems the Controller is given a second chance to perform any actions, such as calling the Service layer. This may mean "redirecting" to a different View. Otherwise, the current View is shown, with any validation errors as necessary.

In this way, you could have files named `controllers/before/whatever.*` and `controllers/do/whatever.*` that get included just like Views.

Git Commits & Video Reflection

The same rules about Git Commits and Reflection Videos apply for this Lab as for previous labs. Review the Lab 1 documentation to ensure you meet all of the required points. Your required questions are mostly the same, with small differences:

1. What didn't go as well as you had hoped? Where did you run into problems with the lab?
2. What went better or was easier than you expected? What went right?
3. **Do you think all of this Controller stuff is worth it just to get pretty URLs? Or is there more to it than that?**
4. How do you think what you did in this lab is going to help you in your future school work, career, or life?

Where should you be?

The breakdown of the data layer, the presentation layer, and the logic should be more familiar now. You should feel very comfortable making new classes and working with objects. URL rewriting may be a little fuzzy, as may the concept of using the Command Pattern to build a single Controller.

Deliverables

Make sure your source code, assets, and reflection video URL are committed to the repository. Push your changes to the server before the start of the next lecture.