## Introduction

This post gives an example of a controller that can maintain a database table. The controller is only accessible by an administrator.

## URIs

The URI scheme is as follows:

|  |  |  |
| --- | --- | --- |
| **URI** | **Controller** | **Responsibility** |
| /admin/categories | Categories | Displays a list of categories with action links |
| /admin/category/new | Category | Manages creation of a new category |
| /admin/category/edit/n | Category | Manages editing of an existing category |
| /admin/category/delete/n | Category | Manages deletion of an existing category |
| /admin/category/view/n | Category | Displays an existing category with action links |

(Note: n represents a category ID)

## Model(s)

Models are the heart of any application. They express the application logic and so-called *business rules* in a way that is independent of the user interface. It’s important then that they are robust and well tested. Of course, all our code should be robust and well tested, but if we have to make any trade-offs, models should get the first call on our time.

The categories list is basically just a read-only list of names so I haven’t bothered with a formal model yet; I’ve just used an array as the data model. Later, I’ll probably refactor this to use a formal model. It’s not ideal, but I can live with a few “smells” in the code for the time being. At present, all the real work is happening in the category.

I’ve created a ***CategoryModel*** class that handles all the CRUD (Create/Read/Update/Delete) operations. It also includes validation and persistence in a database. The version of the constructor without an ID creates a new empty model (i.e. nulls in all data). To create a category, use the setters to populate data and then call save. The version of the constructor with an ID populates the model with data from the database. To edit this, use appropriate setters to make changes and call save. To delete it, use the delete method. Data can be accessed by calling the getters. The model has the responsibility of assuring data integrity, so it will throw exceptions if it finds anything amiss; I call this the *ambulance at the bottom of the cliff* approach. Well written code should check data before processing it; I call this the *fence at the top of the cliff* approach. To make this easy for users of the class I’ve written validators for all data field as shared *static* methods; you don’t need to create an instance of the class to use static methods.

I’ve also written a unit test called ***CategoryTest***. This gives the category class a good workout and is run automatically in runtests.php.

## Views

In previous posts, I’ve created a separate view for each controller class, but this is not really a goal. Ideally views should be as generic as possible and shared across multiple controllers. Views are really just helpers for a controller and save the controller dealing with presentation details. Later, we’ll add views that know how to do things like formatting an html table or presenting a drop down list. For now, I’ve created a couple of generis views.

The class ***SimpleView*** just lets us add generic html content to the content section of our master page; I’m using this in the categories controller. The class ***CommonView*** supports the idea of a sub-view with its own template. I’m using this in the category controller with separate templates for new, edit, delete and view. Each template is just html with appropriate substitutable parameters.

## Controllers

The ***CategoriesController*** just queries the database for a list of categories and then generates an html with action links. This is **not** a good example of MVC! The code “smells” referred to earlier are (a) accessing the database directly rather than through a model and (b) creating html directly rather than using a view. I’ll have to improve this later, but for now, I’d like to note that with an agile approach this is quite a common situation. We just have to remember to add the refactoring task to the backlog!

In contrast, the CategoryController demonstrates a good MVC pattern. Because I’m guessing that categories won’t be the only table needing CRUD capability, I’ve jumped right in and created a crudController class for the common logic and a sub-class Category Controller with the category specific stuff.

The crudController handles the basic pattern of new/edit n/delete n/view n. It manages the logic of presenting a blank form, validating data, representing invalid data with error messages and processing but delegates all the detail to abstract methods implemented in sub-classes. These methods are discussed below.

The CategoryController provides overrides for category specific information in the CRUD handling as follows:

|  |  |  |
| --- | --- | --- |
| **Method** | **Process** | **Return** |
| getPagename | None | “Categories” |
| getTemplateForNew | None | “html/forms/adminCategoryNew.html” |
| getTemplateForEdit | None | “html/forms/adminCategoryEdit.html” |
| getTemplateForDelete | None | “html/forms/adminCategoryDelete.html” |
| getTemplateForView | None | “html/forms/adminCategoryView.html” |
| createModel | Creates a new category model | The model |
| getModelData | Calls setField for data | None |
| getFormData | Validates input data, calls setField and setError | None |
| updateModel | Calls setters on model then save. Redirects to admin/categories | None |
| deleteModel | Calls model delete and redirects to admin/categories | None |

## Putting it all together

I’ve added the routes to the front end controller. Start with runtests.php; this will add the categories table to the database. Note that the *list of categories* option will come up with “page not found” if the administrator is not logged in. The login is “[mike.lopez@cpit.ac.nz](mailto:mike.lopez@cpit.ac.nz)”, password “cabbages”.