# Protean Framework v4.0 Overview

**Eric Jennings - document v1.2**
**Monday, March 7, 2011**

**What is this about?** 1

**First of all, why the name "Protean"?** 2

**MVC and the Front Controller in detail** 2

**Domain Objects–not everything talks to the controller** 4

**Supermodel? Just about, when you use an ORM** 5

**A Smarty-er way to handle the view** 6

**Multi-Language support in the templates? Not a problem** 7

**How to test all of this** 7

**A sample module** 7

## What is this about?

This is an overview of the PHP5 Protean framework. It's goal is to give you a 30,000 foot view in order to better understand the way it works, and more importantly, how to save you valuable time while building your next project.

## First of all, why the name "Protean"?

Let's look at Webster's definition:

**pro·te·an** [**proh**-tee-*uh* n, proh-**tee**-] **– adjective**

1. readily assuming different forms or characters; extremely variable.
2. changeable in shape or form, as an amoeba.
3. (of an actor or actress) versatile; able to play many kinds of roles.

So Protean's goal, at its core, is to make it *easy* to make changes to underlying assumptions or goals of the web application built on top of it. If there's one thing that's constant in web development, it's change. And a surefire way to sign your own death warrant programming-wise is to choose a framework that won't adapt with you when you most need it.

Here are some ways Protean achieves this flexibility:

- Protean uses MVC–specifically the Front Controller design pattern–to manage what to do, when to do it, and how to display it.
- Protean uses Smarty, a completely separate front-end template engine
- Protean uses Propel, a completely separate back-end database ORM layer
- The core API of Protean is small (~30 files), so it's easy to follow what's going on even at the very depths of the framework.

Another main goal of Protean was to never re-invent the wheel if possible. Thus it's very easy to add third-party libraries to Protean, whether they're PEAR-based (Log, PHPUnit), PECL-based (apc, memcache), or independent libraries (Propel, Smarty, jQuery). In fact, several of the named libraries are already included in Protean.

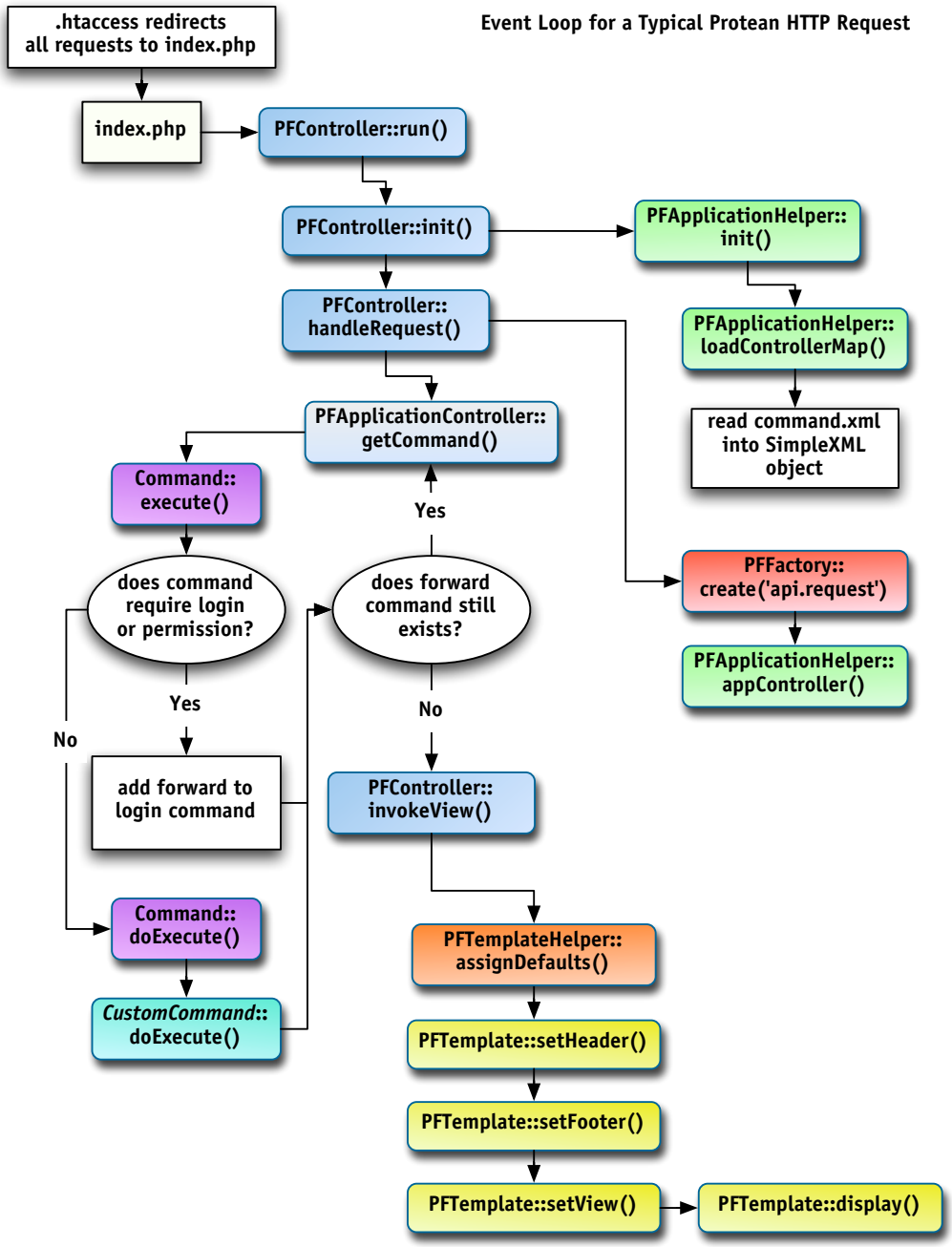
## MVC and the Front Controller in detail

The front controller design pattern comes from Martin Fowler's *Patterns of Enterprise Application Development*. It routes all requests for the entire web application through a single handler object. In Protean's case, it happens in an .htaccess mod_rewrite rule and forwards all requests to the index.php file. This handler object is called, conveniently, *PFController*.

The front controller only knows what to do because it reads its the sitemap from a file called *command.xml*. When a web request is received, this file tells the controller what

command to execute, and what template header, footer, and view to use to display the output of that command.

This gives **enormous** power to the developer, as it now becomes trivial to make page 1 go to page 3 and then page 2 instead of page 1 go to page 2 then to page 3. And, if it's decided that a new header should be shown for a particular page within that sequence (perhaps an admin page), simply swap out the header template for another one within this file--no PHP code is ever touched.



The command.xml file, command objects, HTTP verbs, and status codes all create an environment where routing and chaining events becomes trivial.

Consider this scenario: A page is created to change a password. But the page should not be shown unless the person is logged in. If they are not logged in, show a login page and redirect back to the change password page if successfully logged in. If they are logged in, show the change password page, and if the form was submitted successfully, forward them on to the thank you page.

To do this by hand would be tedious and error-prone. Here's how Protean makes life easy. To set up all of the routing and forwarding described above, we simply put this snippet in our command.xml file.

```
<uri name="/registration/changepassword">
      <command>registration.changepassword</command>
      <viewheader>content.header</viewheader>
      <viewfooter>content.footer</viewfooter>
      <view>registration.changepassword</view>
      <login>true</login>
      <view status="CMD_OK">registration.thankyou</view>
</command>
```

Now we simply create a *registration.changepassword* command file and implement it's *doExecute()* method. Make sure your changepassword view template file looks the way you want, and that the thank you view template looks good too. Good to go!

A small sidenote: *Commands* in Protean are a subset of the *Controller* within the MVC paradigm. They handle one request made by the browser–in this case, the change password request. The Protean controller manages the routing between all of the commands. A subtle but important distinction.

## Domain Objects–not everything talks to the controller

Having a full-fledged MVC framework is nice, but there will be systems and protocols that will want to interact with your code outside of a web browser. Web services? Crontabs? Unit testing? Mobile? In this case, you want to use domain objects.

Domain objects are simple to grasp–they sit between the controller and the model in our MVC paradigm. Consider them, perhaps, as smart objects that sit on top of the

database/model layer. If you're writing code that is business-specific, whether it's a shopping cart object or a user object, make it a domain object. To be more precise, Martin Fowler describes domain objects as part of a *Domain Model*:

> *A [domain] model is a group of objects of the domain that incorporates both behavior and data. At its worst business logic can be very complex. Rules and logic describe many different cases and slants of behavior, and it's this complexity that objects were designed to work with. A Domain Model creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line on an order form.*

If you ever wonder what should go in a domain object, just ask yourself this question: If web browsers died tomorrow, could my business logic still be invoked by an alternative protocol (say, web services?) If it can't, it's not in a domain object. Put it there and you're all set.

## Supermodel? Just about, when you use an ORM

ORMs, or Object-Relational Mappers, have traditionally been a mixed-bag. One one end, they can be a pain to set up, a pain to maintain, and finicky about the tiniest things. On the other hand, once set up, they can save you countless hours of hand-writing SQL queries, converting the results to arrays to pass to your view, auto-managing validation, and so on.

Protean uses Propel for its ORM. If mysql_query is a tricycle, then Propel is a 651 horsepower Ferrari Enzo. What that means is you can pass over vast distances of code and busywork at blinding speed, but you need to have some experience handling a high-performance piece of machinery. Here are some of the best features of Propel:

- Clean support of objects directly out of SQL tables. It's like having an OO database.
- Very easy to set up relationships: one-to-one, one-to-many, many-to-many, and even self-referencing relationships used for adjacency-lists and trees.
- Easy to write both Propel-based queriees and raw SQL.
- Support for migrations to easily alter schema between releases (and rollback too!)
- Built-in support for query caching to memcache, APC, and sqlite
- Event listeners. 'nuff said.

- Support for behaviors. Behaviors allow you to add support simply to tables for functionality like versioning, soft-delete, timestamps, i18n, searchable, geographical, and others. Instead of always adding those *created_at* and *updated_at* fields to every table, just give it a *Timestampable* behavior, and the rest is taken care of for you, from field creation to auto updating the fields.

## A Smarty-er way to handle the view

Contrary to what some people think, we like Smarty. It's reasonably lightweight, supports compilation and caching of parsed templates, is easily extensible with its pre/post filtering, and rarely gets in the way. You know a library is good if it helps you a lot and rarely holds you back.

There is not much that can be said about Smarty here in an short intro to Protean. However, here are some best-practices in using Smarty with Protean:

- Don't be afraid to put some logic in Smarty. {if}{else}{/if} statements are immensely helpful when trying to extract the maximum amount of reusability and functionality out of a single template. However **do not** put business logic in the Smarty template, only display and interface logic. Remember the domain objects in the previous section? If Smarty died tomorrow, would your code still process correctly?

- {include} is your friend. This gives you tons of flexibility in the interface department. Make one HTML menu snippet, call it something like *nav-top.tpl*, and include it all over the site in other templates. Need to add an item to the menu? Modify *nav-top.tpl* and voilà, all pages everywhere that use that menu are updated for you! Think in HTML snippets, not in whole web pages.

- Use modifiers! Don't format your data within your domain or controller objects. Formatting is an interface issue, deal with it in the view. Smarty has a ton built in–here are a couple:

  - {$fullstory|wordwrap:60:"<br />"}
  - {$fullstory|truncate:100}
  - {$fullstory|nl2br}
  - {$fullstory|wordwrap:60:"<br />"|truncate:100|nl2br} (*Neat! Chain them together!)*

- Need a constant directly from PHP? Just call {$smarty.const.CONST_NAME}.

## Multi-Language support in the templates? Not a problem

Protean has built-in support for multiple languages, even down to the error message level. This is achieved with a clever mix of controller magic and Smarty's built-in support for pre-filters.

To add tokens for additional languages, simply add them to a file called global.lng contained in a directory named the ISO 2 character code of the language. A typical file entry looks like this:

YOU_HAVE_BEEN_LOGGED_OUT=You have been logged out.

and in Smarty, you simply put this in your file:

...<h2>##YOU_HAVE_BEEN_LOGGED_OUT##</h2>...

## How to test all of this

So, with a system that was designed from the ground up to be flexible, simple to get results going quickly, and able to support very large web applications, testing becomes an integral part of the system.

For Protean, PHPUnit is the test framework of choice. Every module has a *test/* directory, and in that is a *unit* directory, along with several test classes. At a minimum, test coverage should cover all domain objects within your module. Protean's moving toward a continuous integration mindset, and every module's test directory will be automatically executed regularly and its results reported. Plus, if you have good test coverage of your domain objects, you're practically guaranteed better, more stable code when refactoring or adding functionality. And not to mention the better sleep you'll get knowing your code passed its smoke tests!

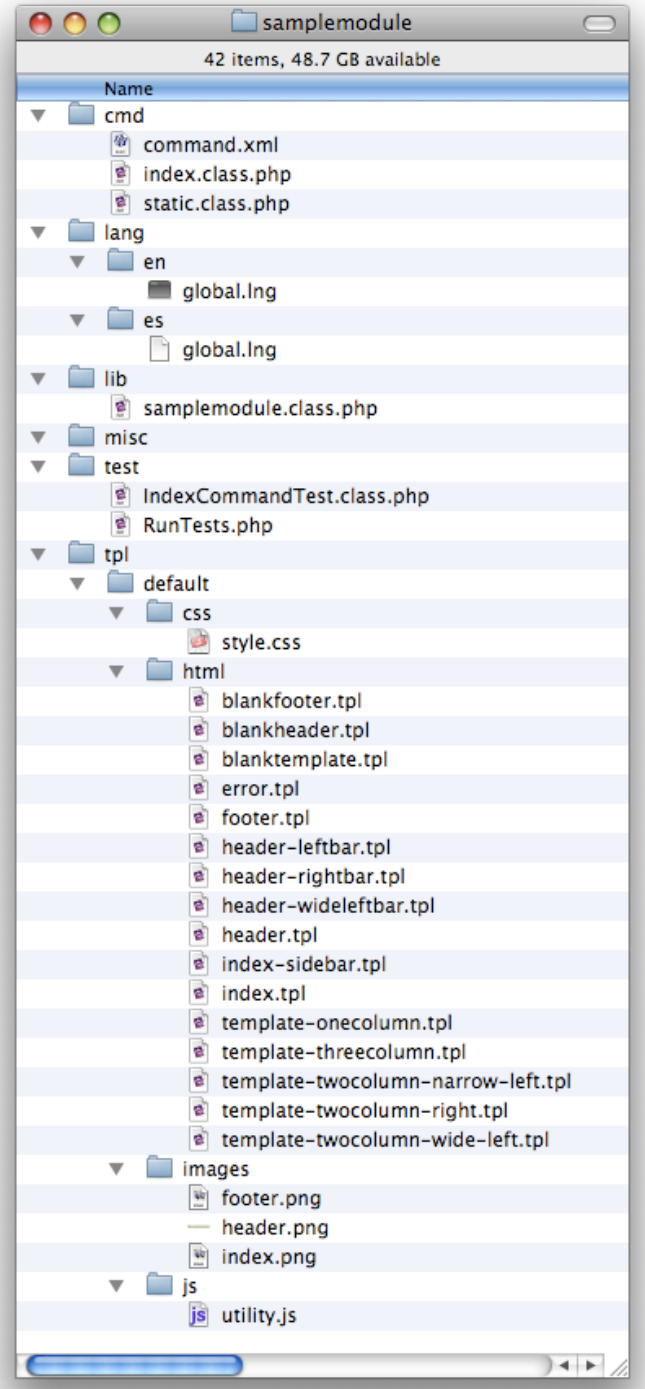
Please see the sample module's *test/* directory for a simple example on how to write tests for your module.

## A sample module

A module is what we call anything that has some cohesive bundle of functionality from a business standpoint. Modules can and often do depend on other modules. However,

modules should be attempted to be kept at a reasonable number. For most Protean installs, the default module is *content/* while other common ones are *registration/* and *shop/*.

There are some special modules. One of these is the *db/* module. This module doesn't adhere to the directory structure of the other modules, as it contains the generated ORM class files, configuration files, and SQL files for Propel. Another special module is *thirdparty/* which is a container for third-party libraries. Below you'll see a typical module layout. Each module has the following directories:




**cmd**
cmd stores all command files, along with the command.xml file for this module.

**lang**
lang contains the language files for multi-language support. The default language is en.

**lib**
Here is where you store all of your domain objects.

**misc**
If you have template-independent miscellaneous files you need access to within a module, place them here. Past uses for misc within modules has been to hold PDF downloads and to store images uploaded by users.

**test**
Like its name implies, *test/* stores all of your unit tests and the RunTests.php script.

**tpl**
This stores everything you need for your view, including HTML templates, CSS files, images, and JavaScript. Furthermore, the "default" directory name within *tpl/* is called a *Theme* and it's easy to switch between themes within the command.xml file. Designing a new layout? Make another folder next to "default", include the same files, and redesign them. Now you can switch entire layouts with a simple <theme> tag. Neat!