PHOCOA User Guide

PHOCOA User Guide

Table of Contents

Chapter 1. Introduction to PHOCOA

Overview

PHOCOA is an object-oriented, event-driven, componentized, MVC (model-view-controller) web application framework based on Apple's Cocoa architecture.

PHOCOA's primary intent is to make developing web applications in PHP easier, faster, and with fewer bugs. The framework handles most of the "dirty work" of programming by removing the need to write much glue code for data binding (moving data between the model layer and view layer of your application), validation, error handling, request processing, etc. Most of your time will be spent designing your GUI and writing application-specific logic rather than dealing with form data, database calls, etc.

As much as possible, PHOCOA aims to make it so that as much of the code you write as possible is specific to your own application.

External Technologies

PHOCOA relies on several major technologies to work its magic:

- PHP 5 - programming language
- Smarty - user interface / presentation layer
- Propel - database abstraction / model layer

A complete list of dependencies is available.

PHOCOA Technologies

PHOCOA itself contributes several technologies to the framework.

- Request Controller - handles automatically handing off web requests to various custom "modules" that will implement the application's logic.
- Page Controller - handles initializing, displaying, and maintaining state of UI objects.
- Bindings - provides the ability to "bind" UI objects to data objects and thereby elimintate the need to write glue code to move data between the model and the view layers.
- PHOCOA Builder - a GUI app (for Mac OS X) that lets you easily configure the UI and bindings for your application.

Approach and Concepts

Developing web applications presents a variety of challenges. Listed below are a number of classic web development architectural problems, and how PHOCOA provides solutions.

- Separation of Data, Presentation, and Glue layers of the code.

Presentation templates should be clearly separated from other code, allowing designers to easily edit the look and feel without breaking the appilcation or dealing with business logic.

Data model code should represent only the pure data model of your system, and have no code related to UI.

Glue code links the model layer to the view layer. Glue code is typically the code for each "web page". It loads up the right data and view pieces for that page, and links the two together.

This problem is typically solved with a MVC design.

PHOCOA uses a MVC architecture as part of its framework.

PHOCOA has some flexibility in the model layer, but we recommend Propel for model objects that are stored in an RDBMS. Of course, not all of your model objects will be persisted, and you can write your own classes as well. You can use other persistence solutions as well, but this may break some of the functionality of the framework.

For the presentation layer, PHOCOA uses a template system to separate the presentation layer from the Model and Controller layers. PHOCOA uses the Smarty template engine, but it is possible that the system could be extended to use other template engines.

PHOCOA's controller layer is one of the most powerful parts of the framework. The PHOCOA controller layer provides automatic state-maintenance of form data, a formalized data validation mechanism, and a bindings layer. All of these items will be discussed in more detail below, but have the net effect of drastically reducing the amount of code you have to write.

- Persistence / Retrieval of Model objects to a database.

  Model objects should be accessible through a single interface and easily retrieved and persisted. Typically this means saving complex relational model data in a RDBMS, as well as querying the DB and restoring model state.

  This problem is typically solved by Object-Relational mapping tools.

  PHOCOA currently uses a modified version of Propel to provide object persistence. The modification that is made is simply to make the Propel BaseObject subclass the PHOCOA framework's base object, WFObject. This allows Propel objects to provide the requisite interfaces for integrating with PHOCOA.

- "Skinning" - having an overall look and feel for a site that is easily switched out. Also, having different look & feel for different parts of a site. Also, getting data into the HEAD section of the page.

  All web sites have graphics that are used for every page. We call this a "skin". Some sites allow different themes of a single skin, and others use different skins for different parts of the site.

  This problem is typically solved with include files and logic to switch between files.

  PHOCOA has a complete infrastructure for skinning built into the framework. A web application can have an unlimited number of skin setups for different parts of the site. Additionally, each of these skin setups can have an unlimited number of skins with an unlimited number of sub-themes. Skins in PHOCOA are completely arbitrary and impose absolutely no design restrictions.

  The undlerlying skin mechanism also allows you to easily customize HEAD information such as title and meta tags.

  PHOCOA also includes a menu system infrastructure for managing hierachical menu systems.

- Maintaining State of a Form Between Requests.

  A nice web application will keep track of all of the data the user has entered into the form. In the case that the form's action cannot be completed due to validation or other error, the form will need to be re-displayed, and it sure is nice if it looks the same as when the user pressed "Submit".

  This problem is typically solved by a lot of glue code, or with tools like HTML_QuickForm or pat-Forms.

  PHOCOA provides a complete UI state-maintainance mechanism. A full set of widgets are provided that represent all HTML input types. Each widget automatically maintains its own state. It is also possible to create custom widgets to promote re-use of complex widgets such as color pickers, date pickers, etc.

- Centralized Dispatch Architecture

  Web applications typically have many "actions" that can be performed by the user. Developers need a way to easily determine which action was requested, and a way to dispatch this action to the correct handler.

  This problem is typically solved with a Front Controller pattern.

  PHOCOA implements a Front Controller that locates "modules" in unlimited folder structure. PHOCOA also allows your modules to access PATH INFO data to promote friendly-URL use.

  PHOCOA is an event-driven architecture as well, automatically dispatching control to your action handlers in response to FORM submissions.

- Request Variable Normalization

  Web applications should take care to prevent XSS (cross-site scripting) attacks caused by user manipulation of the request data.

  This problem is typically solved by writing code to filter all incoming data to make sure that it's valid and meaningful.

  PHOCOA's UI state management only responds to manifested FORM parameters, reducing XSS attacks.

- Data Validation and Normalization

  Data input to a web application, whether via Path Info or Form Submission, needs to be propagated to the model and validated. Ideally, you should be able to detect multiple problems at the same time to make the site easier to use. At some point the data must also be normalized. The errors must then of course be shown to the user.

  This problem is typically solved by a number of methods: pre-validating data, integrated validation code in model components, etc.

  PHOCOA provides a complete solution for data validation and normalization. A concept called Key-Value Coding provides a common infrastructure for writing validation methods on any object. Normalization of the data occurs inside the validator. Beyond that, the PHOCOA infrastructure provides a centralized location to track all errors in a single request. PHOCOA also provides an easy way to display these errors in the UI. You can list all errors as well as errors for each individual widget.

- Web Application Configuration and Deployment

  There are many settings that are application-wide and need to be accessible by all parts of the application. This global data can be broken down into two types of data: data that is the same whether the ap-

plication is running on a Production or Development server, and data that is always needed globally, but changes based on the Production/Development status.

Typically this problem is solved by include files.

PHOCOA configuration handles both of these situations. Application-specific configuration is typically handled by the WFWebApplication object via callbacks, and Deployment-specific configuration is handled via a conf file.

- Session Management

  Sometimes web applications need to keep state across multiple requests that is linked to a user's session.

  This problem is typically solved using PHP's Session API.

  PHOCOA doesn't yet provide any session infrastructure. You are of course free to use PHP's $_SESSION global to manage your own session data.

- User Authorization and Authentication

  Many web sites today have a "login" capability. Users can log in to access additional functionality or personalization. While the specifics of what a logged in user gets vary greatly among applications, they all share the need to perform authentication.

  This problem is typically solved with a special web page to perform authentication, followed by keeping track of the user in a session context.

  PHOCOA has a simple authorization infrastructure that requires implementation of only a few methods to login-enable your application. A simple interface allows your back-end application to use whatever authentication protocol is appropriate. PHOCOA's authorization manager also includes support for "Remember Me" functionality.

- Component Re-Use and Compositing

  Many web applications have "snippets" of functionality that need to be re-used througout the site. Search, login, headlines, are often self-contained "portlets" of functionality that should be re-usable.

  This problem is typically solved via includes.

  PHOCOA provides a rich compositing architecture. Each module you create is automatically reusable in other modules, or in the skin itself. PHOCOA provides a variety of compositing and targeting capabilities for ushering users through the site.

- Data Pagination

  Pagination is the "chunking" of large data sets into smaller portions. This prevents a web page from being infinitely long. Pagination also typically includes sorting options for the data set.

  While many data access layers already have pagination support, developers are typically left to handle the UI portions of pagination on their own.

  PHOCOA provides out-of-the-box support for pagination of PHP arrays, Propel Criteria-based data sets, and raw SQL data sets via Creole. Propel includes several pagination widgets to aid in the display of the paginator navigation, and to help with the process of switching pages, sorting, etc.

- Clean URLs

Modern web sites need to have clean, simple URLs (i.e., such as www.mydomain.com/products/myProduct) to provide a easy-to-remember URLs and be search-engine friendly.

This problem is typically solved by parsing the PATH_INFO from a URL into your own data structure, then displaying the desired page.

PHOCOA's request parameter system makes it easy declare a clean URL interface for any page.

- AJAX

  Interactive web sites built with AJAX are becoming more popular as web users demand that web applications become as interactive and responsive as their desktop counterparts.

  This problem is typically solved using AJAX libraries and writing a lot of JavaScript code and custom PHP pages to deliver data.

  PHOCOA's AJAX implementation wraps industry standard AJAX libraries such as prototype.js and Yahoo's YUI library to make it a snap to build rich, interactive web pages.

Chapter 2. Installation

This chapter explains how to install the PHOCOA framework on your system.

Framework Installation

Dependencies

PHOCOA depends on many other great projects. Some of these are optional, and some are required. PHOCOA works best if the following are installed on your system:

- PHP 5.2.x+
- phing
- Smarty
- PEAR::Log, PEAR::Mail, PEAR::Mail_Mime, PEAR::Net_SMTP
- A YAML parser - either Horde/Yaml [http://pear.horde.org/index.php?package=yaml], or syck [http://whytheluckystiff.net/syck/] (great installation instructions here [http://trac.symfony-project.com/wiki/InstallingSyck]l)
- Propel 1.2 or 1.3 (1.3 is recommended)
- PhpDocumentor

Below is a quick reference for installing these dependencies. For complete details, see Appendix A.

Installing PHP5 is beyond the scope of this documentation.

You can install many of PHOCOA's dependencies with pear:

```
pear install Log Mail Mail_Mime Net_SMTP PhpDocumentor

pear channel-discover pear.phing.info
pear install phing/phing

pear channel-discover pear.horde.org
pear install Horde/Yaml

pear channel-discover pear.phpdb.org
pear config-set preferred_state beta
pear install phpdb/propel_generator phpdb/propel_runtime
```

To install smarty, follow the directions from the Smarty web site [http://www.smarty.net/manual/en/installing.smarty.basic.php] on a smarty package you download [http://www.smarty.net/download.php].

Where to Install Dependencies

Depending on where you install your non-PEAR dependencies (Smarty, etc), you need to tweak the PHP include_path setting to ensure that PHOCOA can see all of your dependencies.

PHOCOA assumes that things are installed in PEAR-like directory structures. That is, PHOCOA will look for propel via `require('propel/Propel.php')`, and assume that there is a directory propel with the propel code in your include path.

The best way to set up non-PEAR dependencies is to create a directory somewhere called "phplib", and in that folder, put all of your dependencies:

```
$ ls -1 /opt/phplib
propel
smarty
```

Then, edit the include_path in webapp.conf to include /opt/phplib.

Propel Integration

Propel is PHOCOA's equivalent of Core Data. Propel allows you to model your data objects in a simple format, automatically handles database retrieval and persistence, and provides you with a place to add custom business logic to your model.

Note

One minor modification needs to be made to the Propel code to work with PHOCOA. See Appendix B for instructions. A similar change would need to be made for any other ORM you would want to use.

Note

PHOCOA works best with Propel 1.3. You can use 1.2, but you have to manage your own Propel class autoloading and some code-gen capabilities have not been tested.

Note

Although PHOCOA includes support for Propel out-of-the-box, the two projects are very loosely coupled. You could theoretically use any PHP ORM/DB solution with PHOCOA.

The PHOCOA framework is contained in its own directory. Your PHOCOA-based web application will live in its own directory, separate from the framework code. This makes it easy to keep the two separated for purposes of backup, upgrading, etc.

Note

Eventually we will create a pear channel for installing phocoa.

PHOCOA Directory Structure

First let's install the PHOCOA framework. Unpack the PHOCOA tarball. The directory structure looks like:

```
$ ls -l phocoa/
total 0
drwxr-xr-x    4 alanpins staff         136 Aug  2 10:34 classes
drwxr-xr-x    5 alanpins staff         170 Oct 15 16:53 conf
drwxr-xr-x    7 alanpins staff         238 Oct 17 14:57 docs
drwxr-xr-x   34 alanpins staff        1156 Oct 17 13:57 framework
drwxr-xr-x    9 alanpins staff         306 Oct 16 12:04 modules
drwxr-xr-x    6 alanpins staff         204 Oct 15 16:50 phing
drwxr-xr-x    4 alanpins staff         136 Aug  2 10:33 skins
```

```
drwxr-xr-x    5 alanpins staff         170 Aug  2 10:33 smarty
drwxr-xr-x    5 alanpins staff         170 Aug  2 10:34 wwwroot
```

The `classes` directory contains a skeleton of the basic application infrastructure needed to have a functional application. This will be copied to your application's modules directory during install.

The `conf` directory contains default versions of all configuraiton files. These will be copied to your application's conf directory during install.

The `docs` directory contains a complete PHPDoc API reference for the framework. Once you have PHOCOA installed, this can be reached from http://your-server.com/docs.

Note

The docs directory shipped with PHOCOA does not include built documentation. Use PHPDOC to build the docs with the following command. Eventually, this will be set up as a Phing task. The docs are also available online at http://phocoa.com/docs/ [http://phocoa.com/docs].

```
$ cd phocoa
$ phpdoc -dn framework-base -t docs/phpdocs -ti "PHOCOA Documentation" -o HTML:frames:d
--ignore test/ -d framework -f "smarty/plugins/*" -f "conf/webapp.conf"
```

The `framework` directory contains most of the framework's code.

The `modules` directory contains modules that are used by the core framework, or are bundled with the framework.

The `phing` directory contains the phing buildfiles for PHOCOA.

The `skins` directory contains bundled skins. This is just a single skin, so that your application has some skin when it starts. These will be copied to your application's skins directory during install.

The `smarty` directory is where templates and plugins used by the framework go.

The `wwwroot` directory is the public wwwroot. This wwwroot contains the bootstrapping code for a PHOCOA application and a directory for all public www documents.

Chapter 3. Getting Started With PHOCOA Tutorial - Building a Blog

PHOCOA has a very large set of technologies. Because it is based on Cocoa (Apple's development infrastructure), if you are a Cocoa programmer things will make a lot of sense to you. If you are not familar with Cocoa, there is a bit of a learning curve. But trust us, it's worth it. The power of PHOCOA will allow you to deliver robust web applications with minimal coding in record time.

Instead of starting off by explaining all of the concepts and technologies, we will first walk you through the steps to build a simple application (a blog) to show you how easy it is to write PHOCOA applications. You will be much more motivated to learn the concepts when you realize how much time PHOCOA can save you.

This chapter will walk you through the development of a simple blog application that shows off all of the basic concepts. We will explain each concept as simlpy as possible for the example.

Starting a New Project

PHOCOA comes with a shell script to help you manage common PHOCOA tasks. This script is aptly named **phocoa**.

Note

If you're not using a PEAR install of PHOCOA, the **phocoa** command won't be in your path. The phocoa command is located at phocoa/phing/phocoa. We recommend that you alias it to avoid having to type the entire path each time.

Let's use **phocoa** to create a new workspace for our blog project.

```
$ phocoa newProject
phing -f /Users/alanpinstein/dev/sandbox/phocoa/phocoa/phing/build.xml -Dusing.pho
-Dphocoa.dir=/Users/alanpinstein/dev/sandbox/phocoa/phocoa -Dphocoa.project.name=
Buildfile: /Users/alanpinstein/dev/sandbox/phocoa/phocoa/phing/build.xml

phocoa > prepareGeneral:
     [echo] PHOCOA framework base dir at: /Users/alanpinstein/dev/sandbox/phocoa/p

phocoa > newProject:
Enter the name of the new project: [] blog
     [echo] The container directory for your PHOCOA project will be used to place
Enter the path to the project container directory: [/Users/alanpinstein/dev/sandbo
[realpathexpandhome] Resolved /Users/alanpinstein/dev/sandbox/blog to /Users/alanp
Enter the name of the server (ie dns name) that will host this application: [local
Enter the IP of the server that will host this application: [127.0.0.1] 10.0.1.201
Enter the PORT of the server that will host this application: [80] 8080
[phingcall] Calling Buildfile '/Users/alanpinstein/dev/sandbox/phocoa/phocoa/phing

phocoa > setupProjectContainer:
     [echo] Creating project container directories and setting up permissions
    [mkdir] Created dir: /Users/alanpinstein/dev/sandbox/blog
    [mkdir] Created dir: /Users/alanpinstein/dev/sandbox/blog/log
    [chmod] Changed file mode on '/Users/alanpinstein/dev/sandbox/blog/log' to 777
    [mkdir] Created dir: /Users/alanpinstein/dev/sandbox/blog/runtime
    [chmod] Changed file mode on '/Users/alanpinstein/dev/sandbox/blog/runtime' to
    [mkdir] Created dir: /Users/alanpinstein/dev/sandbox/blog/runtime/smarty/templ
    [chmod] Changed file mode on '/Users/alanpinstein/dev/sandbox/blog/runtime/sma
```

```
     [echo] Creating project directory: /Users/alanpinstein/dev/sandbox/blog/blog
    [mkdir] Created dir: /Users/alanpinstein/dev/sandbox/blog/blog
     [echo] Copying PHOCOA templates...
     [copy] Copying 9 files to /Users/alanpinstein/dev/sandbox/blog/blog
    [mkdir] Created dir: /Users/alanpinstein/dev/sandbox/blog/blog/wwwroot/www
    [mkdir] Created dir: /Users/alanpinstein/dev/sandbox/blog/blog/modules
     [echo] Setting up configuration files...
     [copy] Copying 3 files to /Users/alanpinstein/dev/sandbox/blog/blog
[filter:ReplaceTokens] Replaced "##SERVER_IP##" with "10.0.1.201"
[filter:ReplaceTokens] Replaced "##SERVER_PORT##" with "8080"
[filter:ReplaceTokens] Replaced "##SERVER_NAME##" with "blog.phocoa.com"
[filter:ReplaceTokens] Replaced "##PHOCOA_APP_DIR##" with "/Users/alanpinstein/dev
[filter:ReplaceTokens] Replaced "##PHOCOA_APP_DIR##" with "/Users/alanpinstein/dev
[filter:ReplaceTokens] Replaced "##PHOCOA_BASE_DIR##" with "/Users/alanpinstein/de
[filter:ReplaceTokens] Replaced "##PHOCOA_APP_DIR##" with "/Users/alanpinstein/dev
[filter:ReplaceTokens] Replaced "##PHOCOA_APP_DIR##" with "/Users/alanpinstein/dev
[filter:ReplaceTokens] Replaced "##PHOCOA_BASE_DIR##" with "/Users/alanpinstein/de
[filter:ReplaceTokens] Replaced "##PHOCOA_APP_DIR##" with "/Users/alanpinstein/dev
[filter:ReplaceTokens] Replaced "##PHOCOA_APP_CONTAINER_DIR##" with "/Users/alanpi
[filter:ReplaceTokens] Replaced "##PHOCOA_APP_CONTAINER_DIR##" with "/Users/alanpi
[filter:ReplaceTokens] No token defined for key "##PHOCOA_APP_DIRBASE_DIR##"
[filter:ReplaceTokens] Replaced "##PHOCOA_APP_DIR##" with "/Users/alanpinstein/dev
[filter:ReplaceTokens] Replaced "##PHOCOA_BASE_DIR##" with "/Users/alanpinstein/de
[filter:ReplaceTokens] Replaced "##PHOCOA_APP_CONTAINER_DIR##" with "/Users/alanpi
[filter:ReplaceTokens] Replaced "##PHOCOA_APP_CONTAINER_DIR##" with "/Users/alanpi
[filter:ReplaceTokens] Replaced "##PHOCOA_APP_DIR##" with "/Users/alanpinstein/dev
[phingcall] Calling Buildfile '/Users/alanpinstein/dev/sandbox/phocoa/phocoa/phing

phocoa > prepareGeneral:
     [echo] PHOCOA framework base dir at: /Users/alanpinstein/dev/sandbox/phocoa/p

phocoa > prepareProject:
     [echo] 1
      [php] Evaluating PHP expression: $_ENV['_']
     [echo] PHOCOA project dir at: /Users/alanpinstein/dev/sandbox/blog/blog
[realpathexpandhome] Resolved /Users/alanpinstein/dev/sandbox/blog/blog/.. to /Use
     [echo] PHOCOA project container dir at: /Users/alanpinstein/dev/sandbox/blog
 [property] Loading /Users/alanpinstein/dev/sandbox/blog/blog/conf/build.propertie
 [property] Unable to find property file: /Users/alanpinstein/dev/sandbox/blog/blo

phocoa > httpdconf:
     [echo] PHOCOA requires some httpd configurations to work its magic. You must
Select httpd configuration mode: 1=httpd.conf, 2=.htaccess [1] 1
     [echo] Make sure your httpd.conf file contains the line: Include /Users/alanp
Will this project use database access via Propel?(yes/no) [1] yes
[phingcall] Calling Buildfile '/Users/alanpinstein/dev/sandbox/phocoa/phocoa/phing

phocoa > prepareGeneral:
     [echo] PHOCOA framework base dir at: /Users/alanpinstein/dev/sandbox/phocoa/p

phocoa > prepareProject:
     [echo] 1
      [php] Evaluating PHP expression: $_ENV['_']
     [echo] PHOCOA project dir at: /Users/alanpinstein/dev/sandbox/blog/blog
[realpathexpandhome] Resolved /Users/alanpinstein/dev/sandbox/blog/blog/.. to /Use
     [echo] PHOCOA project container dir at: /Users/alanpinstein/dev/sandbox/blog
 [property] Loading /Users/alanpinstein/dev/sandbox/blog/blog/conf/build.propertie
 [property] Unable to find property file: /Users/alanpinstein/dev/sandbox/blog/blo

phocoa > addpropel:
     [echo] Setting up PHOCOA project for Propel in dir: /Users/alanpinstein/dev/s
Select the path for executable: propel-gen:  /Users/Shared/src/propel-1.3.0beta2/g
[selectExecutable] Using propel-gen at /Users/Shared/src/propel-1.3.0beta2/generat
    [mkdir] Created dir: /Users/alanpinstein/dev/sandbox/blog/blog/propel-build
```

```
Enter the database type:(pgsql,mysql,mssql,sqllite,ldap) pgsql
Enter the database name: blog
Enter the database username: blog
Enter the database password:
Enter the database host: [localhost]
[writeconffile] Writing conf file: /Users/alanpinstein/dev/sandbox/blog/blog/prope
     [echo] Building Propel... setup conf file, reverse engineer database, build d
     [copy] Copying 1 file to /Users/alanpinstein/dev/sandbox/blog/blog/propel-bui
[filter:ReplaceTokens] Replaced "##LOG_DIR##" with "/Users/alanpinstein/dev/sandbo
[filter:ReplaceTokens] Replaced "##PHOCOA_PROJECT_NAME##" with "blog"
[filter:ReplaceTokens] Replaced "##DB_NAME##" with "blog"
[filter:ReplaceTokens] Replaced "##DB_NAME##" with "blog"
[filter:ReplaceTokens] Replaced "##PROPEL_DATABASE##" with "pgsql"
[filter:ReplaceTokens] Replaced "##PROPEL_DATABASE##" with "pgsql"
[filter:ReplaceTokens] Replaced "##DB_NAME##" with "blog"
[filter:ReplaceTokens] Replaced "##DB_USER##" with "blog"
[filter:ReplaceTokens] Replaced "##DB_HOST##" with "localhost"
[filter:ReplaceTokens] Replaced "##DB_PASS##" with ""
[filter:ReplaceTokens] Replaced "##PROPEL_DATABASE##" with "pgsql"
[filter:ReplaceTokens] Replaced "##DB_HOST##" with "localhost"
[filter:ReplaceTokens] Replaced "##DB_NAME##" with "blog"
[filter:ReplaceTokens] Replaced "##DB_USER##" with "blog"
[filter:ReplaceTokens] Replaced "##DB_PASS##" with ""
     [exec] Executing command: /Users/Shared/src/propel-1.3.0beta2/generator/bin/p
     [exec] Buildfile: /Users/Shared/src/propel-1.3.0beta2/generator/build.xml
     [exec] [resolvepath] Resolved /Users/alanpinstein/dev/sandbox/blog/blog/prope
     [exec]
     [exec] BUILD FAILED
     [exec] Target 'convert-props' does not exist in this project.
     [exec] Total time: 0.1039 seconds
     [echo] Propel general setup complete.
     [echo] To complete propel integration, complete the following manual tasks:
     [echo] 1. Make sure that propel is available in your include_path. If not, ed
     [echo] 2. Add define('PROPEL_CONF', APP_ROOT . '/conf/blog-conf.php'); to you
     [echo] 3. Add Propel::init(PROPEL_CONF); to your WFWebApplicationDelegate's i
     [echo] 4. Edit the propel/runtime/classes/propel/om/BaseObject.php BaseObject
If your database already exists, we can generate a PHP interface to your database
     [echo] Skipping Propel code generation. You can always generate your classes
     [echo] /Users/Shared/src/propel-1.3.0beta2/generator/bin/propel-gen /Users/al
     [echo] Done adding Propel support.
     [echo] New Project setup complete.

BUILD FINISHED

Total time: 36.6166 seconds
```

You should now have a directory `blog` containing your new PHOCOA application.

```
$ ls -l blog
total 0
drwxr-xr-x   8 alanpins  showcase  272 Sep 11 14:20 blog
drwxrwxrwx   2 alanpins  showcase   68 Sep 11 14:20 log
drwxrwxrwx   3 alanpins  showcase  102 Sep 11 14:20 runtime
```

This directory contains your PHOCOA deployment structure. This directory is called the container directory because it contains your web application, including your PHOCOA application project. Typically, log and runtime (tmp / cache) files are not part of your source code, so PHOCOA sets up a container directory for these items.

The `blog` directory inside of this deployment structure is your PHOCOA application directory.

Note

The `application` directory is the root directory of your PHOCOA application code, and is what you should check in to your version control system. The `log` and `runtime` directories are not versioned resources, thus we keep them one level up in the container directory for organizational purposes.

Application Directory Structure

Now let's have a look at the directory structure.

```
$ ls -l blog/blog
total 0
drwxr-xr-x   3 alanpins  showcase  102 Sep 11 14:20 classes
drwxr-xr-x   6 alanpins  showcase  204 Sep 11 14:21 conf
drwxr-xr-x   2 alanpins  showcase   68 Sep 11 14:20 modules
drwxr-xr-x   4 alanpins  showcase  136 Sep 11 14:21 propel-build
drwxr-xr-x   3 alanpins  showcase  102 Sep 11 14:20 skins
drwxr-xr-x   4 alanpins  showcase  136 Sep 11 14:20 wwwroot
```

The `classes` directory is where all of your classes go. These are classes specific to your application.

The `conf` directory contains all configuraiton files.

The `modules` directory is where all components go. These components are the building blocks of your application and include both entire pages and sub-components.

The `skins` directory is where all skins go.

The `wwwroot` directory is a public wwwroot that contains the front controller for the PHOCOA project. All public documents (i.e. the tradiditional public www root) go in wwwroot/www/.

Initial Configuration

A new PHOCOA project will have 2 config files. The first is the Apache config file, `httpd.conf`, followed by the web app config file, `webapp.conf`.

```
$ ls -l conf
total 16
-rw-r--r--   1 alanpins  showcase  2594 May 10 15:41 httpd.conf
-rw-r--r--   1 alanpins  showcase  2770 Sep 11 23:03 webapp.conf
```

Usually, you won't need to edit any of these files to get things working.

You will need to edit your main Apache conf file to include the conf file for this host, like so:

```
Include /path/to/blog/blog/conf/httpd.conf
```

Now restart Apache.

Hello, PHOCOA!

If everything worked, you should be able to go to your web host your specified in the newproject build and see the PHOCOA examples page.

At this point, you should be able to access the site via the web. You should be able to access the application via http://servername/ and see the PHOCOA examples page.

Now, we can move on to the tutorial!

Building the Data Model

Before we can build our blog application, we need to create our blog data model, both in the database and PHP code. We will use postgres in our example app.

Let's create our database, the blog table, and then use Propel to create our PHP object model.

Create a database for our project:

```
$ createuser -U postgres blog
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
CREATE ROLE
$ createdb -U postgres blog
CREATE DATABASE
$ psql -U postgres blog -c 'alter database blog owner to blog;'
ALTER DATABASE
```

Next, create the table for our blog posts. Execute the SQL below in your blog database.

```
create table blog
(
    blog_id serial,
    post_dts timestamptz,
    title varchar(100) not null,
    post text not null,
    PRIMARY KEY(blog_id)
);
```

Now, we tell propel to reverse-engineer our database, build our data model, and update the Propel runtime conf file:

```
$ cd blog/blog/propel-build
$ propel-gen . creole om convert-conf
Buildfile: /Users/Shared/src/propel-1.3.0beta2/generator/build.xml
[resolvepath] Resolved . to /Users/alanpinstein/dev/sandbox/blog/blog/propel-build

propel-project-builder > check-project-or-dir-set:

propel-project-builder > check-project-set:

propel-project-builder > set-project-dir:

propel-project-builder > check-buildprops-exists:

propel-project-builder > check-buildprops-for-propel-gen:

propel-project-builder > check-buildprops:

propel-project-builder > configure:
     [echo] Loading project-specific props from /Users/alanpinstein/dev/sandbox/bl
 [property] Loading /Users/alanpinstein/dev/sandbox/blog/blog/propel-build/./build
```

```
propel-project-builder > creole:
    [phing] Calling Buildfile '/Users/Shared/src/propel-1.3.0beta2/generator/build
 [property] Loading /Users/Shared/src/propel-1.3.0beta2/generator/./default.proper

propel > creole:
     [echo] +-----------------------------------------------+
     [echo] |                                               |
     [echo] | Generating XML from Creole connection !       |
     [echo] |                                               |
     [echo] +-----------------------------------------------+
[propel-creole-transform] Propel - CreoleToXMLSchema starting
[propel-creole-transform] Your DB settings are:
[propel-creole-transform] driver : (default)
[propel-creole-transform] URL : pgsql://blog:@localhost/blog
[propel-creole-transform] DB connection established
[propel-creole-transform] Processing database
[propel-creole-transform] Processing table: blog
[propel-creole-transform] Writing XML to file: /Users/alanpinstein/dev/sandbox/blo
[propel-creole-transform] Propel - CreoleToXMLSchema finished

propel-project-builder > check-project-or-dir-set:

propel-project-builder > check-project-set:

propel-project-builder > set-project-dir:

propel-project-builder > check-buildprops-exists:

propel-project-builder > check-buildprops-for-propel-gen:

propel-project-builder > check-buildprops:

propel-project-builder > configure:
     [echo] Loading project-specific props from /Users/alanpinstein/dev/sandbox/bl
 [property] Loading /Users/alanpinstein/dev/sandbox/blog/blog/propel-build/./build

propel-project-builder > om:
    [phing] Calling Buildfile '/Users/Shared/src/propel-1.3.0beta2/generator/build
 [property] Loading /Users/Shared/src/propel-1.3.0beta2/generator/./default.proper

propel > check-run-only-on-schema-change:

propel > om-check:

propel > mysqli-check:

propel > om:
     [echo] +-----------------------------------------+
     [echo] |                                         |
     [echo] | Generating Peer-based Object Model for  |
     [echo] | YOUR Propel project!                    |
     [echo] |                                         |
     [echo] +-----------------------------------------+
[phingcall] Calling Buildfile '/Users/Shared/src/propel-1.3.0beta2/generator/build
 [property] Loading /Users/Shared/src/propel-1.3.0beta2/generator/./default.proper

propel > om-template:
[propel-om] Processing: schema.xml
[propel-om] Processing Datamodel : schema.xml
[propel-om]   - processing database : blog
[propel-om]     + blog
[propel-om]             -> BaseBlogPeer [builder: PHP5ComplexPeerBuilder]
[propel-om]             -> BaseBlog [builder: PHP5ComplexObjectBuilder]
[propel-om]             -> BlogMapBuilder [builder: PHP5MapBuilderBuilder]
```

```
[propel-om]             -> (exists) BlogPeer
[propel-om]             -> (exists) Blog

propel-project-builder > check-project-or-dir-set:

propel-project-builder > check-project-set:

propel-project-builder > set-project-dir:

propel-project-builder > check-buildprops-exists:

propel-project-builder > check-buildprops-for-propel-gen:

propel-project-builder > check-buildprops:

propel-project-builder > configure:
     [echo] Loading project-specific props from /Users/alanpinstein/dev/sandbox/bl
 [property] Loading /Users/alanpinstein/dev/sandbox/blog/blog/propel-build/./build

propel-project-builder > convert-conf:
    [phing] Calling Buildfile '/Users/Shared/src/propel-1.3.0beta2/generator/build
 [property] Loading /Users/Shared/src/propel-1.3.0beta2/generator/./default.proper

propel > convert-conf:
     [echo] +-----------------------------------------+
     [echo] |                                         |
     [echo] | Converting runtime config file to an    |
     [echo] | array dump for improved performance.    |
     [echo] |                                         |
     [echo] +-----------------------------------------+
     [echo] Output file: blog-conf.php
     [echo] XMLFile: /Users/alanpinstein/dev/sandbox/blog/blog/propel-build/./runt
[propel-convert-conf] Processing: schema.xml
[propel-convert-conf] Adding class mapping: BlogMapBuilder => blog/map/BlogMapBuil
[propel-convert-conf] Adding class mapping: BlogPeer => blog/BlogPeer.php
[propel-convert-conf] Adding class mapping: Blog => blog/Blog.php
[propel-convert-conf] Creating PHP runtime conf file: /Users/alanpinstein/dev/sand

BUILD FINISHED

Total time: 0.7053 seconds
```

We now have PHP classes to represent our data model, thanks to Propel. Propel automatically puts these classes in `blog/blog/classes/blog/*`.

Module / Page Component Architecture and Action Events

Now that the data model is built, we can move on to the User Interface. Before we actually build a user interface for our blog application, let's explore the concepts used by PHOCOA to build your UI.

One of the drawbacks of "normal" web application development is that you have to do a lot of work completely unrelated to your application to make it work as a web app. The conceptual goal of PHOCOA (or any web application framework, for that matter) is to provide you with an infrastructure to minimize the amount of non-application-specific code you have to write. PHOCOA is organized in such a way as to abstract away all of the complications of building software for the web, and lets you focus entirely on building the logic and UI for *your* application.

PHOCOA accomplishes this by handling all aspects of the web application, and calling into your applic-

ations' custom code at the appropriate times to allow you to implement your business and interface logic.

The user interface is managed by the View and Controller portions of the MVC programming model. PHOCOA borrows from Cocoa again, translating the concept of nib files and the controller layer to the web.

Modules

The basic unit of work in a PHOCOA application is the module. A module is a collection of web pages (the Views of MVC) and code (the Controllers of MVC) related to a single function of your web application. For instance, you might have a module for editing and previewing a blog post, and a separate module for displaying the blog post to the public.

Each PHOCOA module is a single PHP file that contains a single `WFModule` subclass, and optionally a `WFPageDelegate` class for each page in the module.

There is a naming convention to the `WFModule` subclass names. The subclass should be named `module_<moduleName>` where <moduleName> is the name of the directory that the module sits in. This helps prevent name collisions with other classes. The page delegates follow a similar convention; the page delegate class should be named `module_<moduleName>_<pageName>`.

Because each module contains all of the code and web pages to handle a specific function, each module in PHOCOA is also an easily reusable web component. The components can be used one-at-a-time to build complete pages, or you can composite modules together to create complex layouts and behaviors.

When you finish a module, you have a set of functional web pages to manage a certain aspect of your web application. You also have an API for accessing these functions.

Modules are invoked via an invocationPath which looks like path/to/module/pageName/param1/param2. Obviously this looks a lot like a URL. When you go to a URL of a PHOCOA application, the request controller parses out the invocationPath from the URL and executes the module. Modules that include other modules simply supply the invocationPath directly.

Modules contain the shared objects used by the module's pages. These are declared in the `shared.yaml` file. Think of `shared.yaml` as your nib file, and the module class as the File's Owner.

Pages

Each module can contain an arbitrary number of pages. A page is simply a single web view that a user can see. You can think of it as a web page, a view, a screen, whatever works for you. For instance, you may have one view that is an input form, and another view that is used for telling the user the form's action succeeded.

While in Cocoa you would instantiate all of the UI widgets in the nib file, in PHOCOA each page has its own yaml file which contains all of the UI widgets for that page.

Actions

Each page has a number of actions. Actions are trigger by the user submitting a form. When the user submits a form, PHOCOA will hand off control to an action handler in your module where you can respond to the action.

It should be noted that not all requests have actions. It is possible to just load a page in a module WITHOUT an action. In this case, the module can display the default page, or could look in the parameters passed to it to find information used to load default data into the page.

Actions are implemented by the user by implementing an action callback handler, a method with the signature:

```
function doAction($page, $params);
```

Where "doAction" is replaced with the name of your action (which is by default the name of the submit widget).

Additionally, if there is NO action to be taken on this request, this special callback is executed:

```
function noAction($page, $params);
```

Page Life Cycle

PHOCOA handles all aspects of the web page request automatically, and allows your page to simply "chime in" at appropriate times to implement business logic. The WFPageDelegate interface documents all of the different callbacks your page can implement to create your page's business logic. Below is a brief description of the page life cycle callbacks, in the order in which they occur.

```
function pageInstancesDidLoad($page); // the page's UI has been restored from the serialized state (YAML file).

function parameterList(); // called to get the list of parameters supported by this page.

function parametersDidLoad($page, $params); // the page's parameters are loaded. typically you will load your data in this callback.

function willPushBindings($page, $params); // the changes made on the client are about to be batch-applied to the current state of the application.

function doAction($page, $params); // the action method is called all batch-applied changes are made without error

function noAction($page, $params); // the page has loaded, but no action has been executed

function setupSkin($page, $parameters, $skin); // allow the page to customize the skin being used

function willRenderPage($page, $parameters); // page is about to be rendered

function didRenderPage($page, $parameters, &$output); // page has been rendered but not output to client
```

Code Generation

Now that everything is set up properly, and you understand the concept of how PHOCOA handles web pages, we will actually create a page to edit our blog.

We will start out by creating a new module, "blog". This module will allow us to search, browse, edit, create, and delete blog posts.

First, cd into the `modules` subdirectory.

```
$ cd blog/blog/modules
```

PHOCOA has a code-generation feature that will quickly build a module providing basic CRUD functionality (Create, Read, Update, Delete) for the selected table. This is done using the **phocoa** command-line utility:

```
$ phocoa createSkeletonFromPropel
phing -f /Users/alanpinstein/dev/sandbox/phocoa/phocoa/phing/build.xml -Dusing.pho
Buildfile: /Users/alanpinstein/dev/sandbox/phocoa/phocoa/phing/build.xml

phocoa > prepareGeneral:
     [echo] PHOCOA framework base dir at: /Users/alanpinstein/dev/sandbox/phocoa/p

phocoa > prepareProject:
     [echo] 1
      [php] Evaluating PHP expression: $_ENV['_']
     [echo] PHOCOA project dir at: /Users/alanpinstein/dev/sandbox/blog/blog
[realpathexpandhome] Resolved /Users/alanpinstein/dev/sandbox/blog/blog/.. to /Use
     [echo] PHOCOA project container dir at: /Users/alanpinstein/dev/sandbox/blog
 [property] Loading /Users/alanpinstein/dev/sandbox/blog/blog/conf/build.propertie
 [property] Unable to find property file: /Users/alanpinstein/dev/sandbox/blog/blo

phocoa > createSkeletonFromPropel:
Database Name: [blog]
Table Name: blog
Column name of single column that best describes the table: title
Module Name: [blog]
      [php] Evaluating PHP expression: require('/Users/alanpinstein/dev/sandbox/bl
     [exec] Executing command: /opt/local/bin/php /Users/alanpinstein/dev/sandbox/
     [exec] Writing blog/blog.php
     [exec] Done building module blog!
     [exec] Executing command: /opt/local/bin/php /Users/alanpinstein/dev/sandbox/
     [exec] Setting up page: search
     [exec] Adding shared instance: Blog
     [exec] Adding shared instance: 'paginator'
     [exec] Saving updated shared.instances
     [exec] Adding config for shared instance: Blog
     [exec] Adding config for shared instance 'paginator'
     [exec] Saving updated shared.config
     [exec] Saving updated search.instances
     [exec] Saving updated search.config
     [exec] Creating search.tpl file.
     [exec] Setting up page: edit
     [exec] Saving updated shared.instances
     [exec] Saving updated shared.config
     [exec] Adding instance: blogId
     [exec] Adding config for instance: blogId
     [exec] Adding instance: postDts
     [exec] Adding config for instance: postDts
     [exec] Adding instance: title
     [exec] Adding config for instance: title
     [exec] Adding instance: post
     [exec] Adding config for instance: post
     [exec] Saving updated edit.instances
     [exec] Saving updated edit.config
     [exec] Creating edit.tpl file.
     [exec] Saving updated confirmDelete.instances
     [exec] Saving updated confirmDelete.config
     [exec] Creating confirmDelete.tpl file.
     [exec] Saving updated deleteSuccess.instances
     [exec] Saving updated deleteSuccess.config
     [exec] Creating deleteSuccess.tpl file.
     [exec] Setting up page: detail
```

```
     [exec] Saving updated shared.instances
     [exec] Saving updated shared.config
     [exec] Adding instance: blogId
     [exec] Adding config for instance: blogId
     [exec] Adding instance: postDts
     [exec] Adding config for instance: postDts
     [exec] Adding instance: title
     [exec] Adding config for instance: title
     [exec] Adding instance: post
     [exec] Adding config for instance: post
     [exec] Saving updated detail.instances
     [exec] Saving updated detail.config
     [exec] Creating detail.tpl file.
     [exec] Suggested module code in suggested_code.php
     [exec] Executing command: /opt/local/bin/php /Users/alanpinstein/dev/sandbox/
     [exec] Converting shared setup to YAML for module at: 'blog'
     [exec] Checking .
     [exec] Checking ..
     [exec] Checking blog.php
     [exec] Checking confirmDelete.config
     [exec] Checking confirmDelete.instances
     [exec] Checking confirmDelete.tpl
     [exec] Converting page: confirmDelete.tpl to YAML
     [exec] Checking deleteSuccess.config
     [exec] Checking deleteSuccess.instances
     [exec] Checking deleteSuccess.tpl
     [exec] Converting page: deleteSuccess.tpl to YAML
     [exec] Checking detail.config
     [exec] Checking detail.instances
     [exec] Checking detail.tpl
     [exec] Converting page: detail.tpl to YAML
     [exec] Checking edit.config
     [exec] Checking edit.instances
     [exec] Checking edit.tpl
     [exec] Converting page: edit.tpl to YAML
     [exec] Checking search.config
     [exec] Checking search.instances
     [exec] Checking search.tpl
     [exec] Converting page: search.tpl to YAML
     [exec] Checking shared.config
     [exec] Checking shared.instances
     [exec] Checking shared.yaml
     [exec] Checking suggested_code.php
   [delete] Deleting 12 files from /Users/alanpinstein/dev/sandbox/blog/blog/modul
     [exec] Executing command: mv suggested_code.php blog.php 2>&1

BUILD FINISHED

Total time: 4.4874 seconds
```

The **phocoa createSkeletonFromPropel** command automatically creates a new module and code for all of these functions.

Note

You should not run the code generator more than once for the same module. This may be supported in a future version; right now the effect is untested.

You can now visit your blog editing module at http://servername/blog.

You will notice that this is a fully-functional blog CRUD module. It is also easily maintainable and extensible.

The next few sections will provide a tour of PHOCOA's core technologies and what they do for you. We'll also make a bunch of modifications along the way to improve the generated code.

Customizing the UI

The first thing you'll probably notice about the generated code is that the input types aren't ideal for our application. Let's make a few improvements.

Changing UI Widgets

The first change we want to make is to have the blog post itself to be in HTML. To do this, simply edit the `edit.yaml` file, updating the class of the post object to `WFHTMLArea`.

Note

On Mac OS X, you can use `PHOCOA Builder.app` to make these edits to the UI setup.

Once that change is made, just reload the page and you will see the HTML Editor (thanks, FCKEditor!).

Note

Technically, PHOCOA classifies all UI objects into two major classes: Views and Widgets. Views are read-only UI objects used for layout. Widgets, which are subclasses of views, add support for accepting input and writing data back to bound data objects. View classes can read data via bindings, but they cannot accept input or push data back to bound objects.

Controlling Widget Visibility

Next, we want to hide the Post date field for new entries as it will be added automatically.

To do this, we will add a hidden binding to the post_dts setup in the `edit.yaml` file:

```
postDts:
  class: 'WFTextField'
  bindings:
    value:
      instanceID: 'Blog'
      controllerKey: 'selection'
      modelKeyPath: 'postDts'
    hidden:
      instanceID: 'Blog'
      controllerKey: 'selection'
      modelKeyPath: 'isNew'
```

If you reload now, you'll notice that the Post dts field is now missing when editing a new post. However, the field label is still there. PHOCOA has a handy smarty block element to help you eliminate extra layout markup around hidden widgets. Edit the edit.tpl file as shown:

```
{WFViewHiddenHelper id="postDts"}
<div>
    <label for="postDts">Post Dts:</label>
    {WFView id="postDts"}{WFShowErrors id="postDts"}
</div>
{/WFViewHiddenHelper}
```

You will now see that the field and supporting labels and markup are automatically hidden.

Go ahead and enter in a sample title and post, and create the blog post.

Formatters

Once you have created the new post, the form switches to edit mode by hiding the "Create Post" button and adding in "Save" and "Delete" buttons. Also, you will now see that our "Post Dts" field is visible since the object is no longer new.

However, you'll notice that the "Post Dts" is blank. That's because we aren't setting up a default value for this field, due to a limitation of the current version of Propel. Without getting into the details, suffice it to say that you can't set default dates with Propel at this time that work properly. So, we'll just fix this in our `Blog` class (`classes/blog/Blog.php`) by overriding the `save` method. While we're at it, we'll fix up the `getPostDts` method to return a UNIX date, which we also need:

```
class Blog extends BaseBlog {
    public function save(PDO $con = null)
    {
        if ($this->isNew())
        {
            $this->setPostDts( "now" );
        }
        return parent::save($con);
    }
    public function getPostDts()
    {
        $dt = parent::getPostDts();
        if ($dt)
        {
            return $dt->format('U');
        }
        return NULL;
    }
} // Blog
```

If you go back now and add another Blog entry, you'll see that the "Post Dts" is now filled in.

However, you'll probably notice that the format of the timestamp is very generic. We'll fix that using PHOCOA's formatters, which are small classes that automatically convert between human-readable forms and their machine counterparts.

To add a formatter, we must declare it as a shared instance in the module via `shared.yaml`:

```
postDtsFormatter:
  class: WFUNIXDateFormatter
  properties:
    formatString: 'D, M j H:m:s'
```

Now, we attach our formatter to our "Post Dts" field in `edit.yaml`:

```
    postDts:
      class: 'WFTextField'
      properties:
        formatter: '#module#postDtsFormatter'
      bindings:
        value:
          instanceID: 'Blog'
          controllerKey: 'selection'
```

```
    modelKeyPath: 'postDts'
  hidden:
    instanceID: 'Blog'
    controllerKey: 'selection'
    modelKeyPath: 'isNew'
```

Note

Notice the #module# preceding the shared instance ID in the formatter; since you can use the YAML mechanism to set up values such as strings, integers, booleans, and doubles, we needed a special flag to indicate that you want to link it to the value of an instance variable of the module, which is what shared instances are.

Reload, and you'll see the new date format. A few neat things to notice about formatters; the formatters try to take any human-readable value and convert it to something useful. For instance, try entering and saving a "Post Dts" of "tomorrow", or "next tuesday". Also, try entering garbage, and you will see that formatters automatically detect error and display error conditions.

Validating Data and Presenting Errors

Right now, the only error handling we've seen is on the "Post Dts" field, thanks to the formatter. We want to make some of our fields required and potentially check for other issues. PHOCOA has a built-in data normalization and validation mechanism.

It's time once again to introduce another concept: Key-Value Validation. The Key-Value Validation mechanism looks for specially-named functions of your class, with the following prototype:

```
boolean validate<Key>(&$value, &$edited, &$errors)
```

It is very important to notice the pass-by-reference used on all three parameters.

So, we can implement our two validators by adding the following code to our model object `Blog` in `Blog.php`:

```
function validateTitle(&$value, &$edited, &$errors)
{
    $value = trim($value);
    $edited = true;
    $ok = true;
    if (!$value)
    {
        $errors[] = new WFError("The blog post must have a title.");
        $ok = false;
    }
    if (strlen($value) > 20)
    {
        $errors[] = new WFError("The blog post title must be 20 characters or
        $ok = false;
    }
    return $ok;
}

function validatePostDts(&$value, &$edited, &$errors)
{
    if (!$value and !$this->isNew())
    {
        $errors[] = new WFError("You must enter the Posting date.");
        return false;
```

```
    }
    return true;
}

function validatePost(&$value, &$edited, &$errors)
{
    $value = trim($value);
    if (!$value)
    {
        $errors[] = new WFError("You have not entered your blog post!");
        return false;
    }
    return true;
}
```

Validators in PHOCOA are a bit different than some validation mechanisms. Validators are called BEFORE the actual value is set, by the Bindings system, as a pre-flight mechanism. Doing it this way prevents invalid data from ever being in the class.

The validators can also do normalization of the data. Since the value is passed by reference, you can normalize the data in any appropriate way. If you do alter the data, be sure to set edited to true.

Notice also that the errors parameter is an array. This allows you to specify multiple errors during validation.

If you look now in `edit.tpl`, you can see how we display errors in PHOCOA. The tag {WFShowErrors} will display all errors. By adding an id attribute to the tag, it will show just the errors associated with that UI element. You will see this done next to the individual widgets.

Now try the form again, but this time enter invalid data for the title (blank, or more than 20 chars) and date (blank) fields. You'll see that all of the errors are listed above the form, and the specific errors are again repeated next to the relevant form elements.

Handling Actions and Responding

Taking Action

Once PHOCOA determines that your form data is valid, it will call the action method corresponding to the button pressed by the user. If the validation fails at all, then the action method will not be called.

The action method is simply a method of the page delegate named "`<action>Action`". For instance, find the `saveAction` method in `blog.php`, which is called when the Save button is pressed.

Sometimes of course, there will be no errors during the bindings phase, but errors may still occur when processing the action. During the bindings phase, the validators are processed through simple Key-Value Validation. This by definition occurs one property at a time. However, some validation can't be done until all properties have been set. For instance, to validate a particular property, you may need to know the value of one or more other properties. Or, during the database update, the database might return an error. In this case, you can easily add additional errors using the `addError` function of `WFPage`. Now, of course you don't want your model classes to make `WFPage` method calls, so your model classes would simply throw an exception, and you would catch the exception from the action method and add the error to the page at that point. The `saveAction` implementation is an excellent example of this kind of handling.

Responding With a Different Page

Sometimes, to process your action, you just want to route the user to a different page. The `deleteAction` is an excellent example of this. Examine this method and notice the `setupResponsePage` call, which tells PHOCOA to render the response using a different page in the same module.

Alternatively, you can even redirect execution to a completely different module by throwing a `WFRequestController_InternalRedirectException` or `WFRequestController_RedirectException`, passing the desired URL as the first parameter.

Setting HTML Top-Level Attributes

The skin system makes it easy to implement web page best practices such as implementation of HTML Title, Meta Keywords, and Meta Description tags on each page to keep the information relevant and help with search engine optimization, as well as providing user-friendly web page titles for history navigation.

During the page life cycle, your module/page is given the opportunity to alter the skin being used to render the current page via the following method:

```
function setupSkin($page, $parameters, $skin)
```

Examine the `setupSkin` implementations in your Blog module to see an example of using the skin infrastructure to manage HTML head-element tags.

Editing the Skin

You might have started to notice that it's getting a bit annoying to navigate back-and-forth between the list view and edit views. Plus, the default menu across the top is inappropriate for our blog. Let's go ahead and adjust the menu with more useful options.

Open up the skin delegate file for the default skin, at `skins/simple/simple_SkinDelegate.php`, and edit the mainMenu content to return the following:

```
            case 'mainMenu':
                return array(
                        'Home' => '/',
                        'All Posts' => WFRequestController::WFURL('blog/search'),
                        );
                break;
```

Reload the page, and you'll now see the menu is updated with our new menu options.

Creating Another Module

Click on the home page link, and you'll notice it is taking us to the PHOCOA examples page. We definitely don't want this as the home page of our blog. Instead, we'll create a new home page to show the 10 latest blog posts.

To accomplish this, we're going to create one module that will handle displaying blog posts to the public. Then, we'll create a home page implementation that leverages our public-facing blog page to create a nice home portal.

Create a New Module

First, let's create a new module for our public-facing blog access. CD to the modules directory and use

the **phocoa** utility to do this:

```
$ phocoa createModule
phing -f /Users/alanpinstein/dev/sandbox/phocoa/phocoa/phing/build.xml -Dusing.pho
Buildfile: /Users/alanpinstein/dev/sandbox/phocoa/phocoa/phing/build.xml

phocoa > prepareGeneral:
     [echo] PHOCOA framework base dir at: /Users/alanpinstein/dev/sandbox/phocoa/p

phocoa > prepareProject:
     [echo] 1
      [php] Evaluating PHP expression: $_ENV['_']
     [echo] PHOCOA project dir at: /Users/alanpinstein/dev/sandbox/blog/blog
[realpathexpandhome] Resolved /Users/alanpinstein/dev/sandbox/blog/blog/.. to /Use
     [echo] PHOCOA project container dir at: /Users/alanpinstein/dev/sandbox/blog
 [property] Loading /Users/alanpinstein/dev/sandbox/blog/blog/conf/build.propertie
 [property] Unable to find property file: /Users/alanpinstein/dev/sandbox/blog/blo

phocoa > createModule:
Module name: blogview
Default page [blank for none]: list
     [exec] Executing command: /opt/local/bin/php /Users/alanpinstein/dev/sandbox/
     [exec] Writing blogview/blogview.php
     [exec] Done building module blogview!
     [exec] Writing list.tpl
     [exec] Writing list.yaml
     [exec] Done!

BUILD FINISHED

Total time: 37.1642 seconds
```

This new module/page is accessible at http://servername/blogview. Notice how PHOCOA automatically redirects the user to http://servername/blogview/list since we've set up our list view as the default page.

Setup our Shared Instances and Load Data

Now we need to add code to this module to display our list. First off, we need to create a shared instance to hold our array of blog posts, by editing `shared.yaml`:

```
Blog            :
  class     : WFArrayController
  properties:
    automaticallyPreparesContent: false
    class                       : Blog
    classIdentifiers            : blogId
postDateFormatter:
  class     : WFUNIXDateFormatter
  properties:
    formatString: M j \a\t H:m
```

Next, we need to have our list page delegate load the 10 most recent blog posts into our Blog array controller. Add the following code to `blogview.php`:

```
class module_blogview_list
{
    function parametersDidLoad($page, $params)
    {
        // load the 10 most recent blog posts in desc order.
        $c = new Criteria;
```

```
        $c->addDescendingOrderByColumn(BlogPeer::POST_DTS);
        $c->setLimit(10);
        $page->sharedOutlet('Blog')->setContent( BlogPeer::doSelect($c) );
    }
}
```

Set up UI Widgets

Now we need to add our page UI objects to display the blog posts. For each post, we'll show the title (hyperlinked to the full post) and the post date.

```
postDts:
  children  :
    postDtsPrototype:
      bindings  :
        value:
          controllerKey: '#current#'
          instanceID   : Blog
          modelKeyPath : postDts
      class     : WFLabel
      properties:
        formatter: '#module#postDateFormatter'
  class     : WFDynamic
  properties:
    arrayController: '#module#Blog'
title  :
  children  :
    titlePrototype:
      bindings:
        label:
          controllerKey: '#current#'
          instanceID   : Blog
          modelKeyPath : title
        value:
          controllerKey: '#current#'
          instanceID   : Blog
          modelKeyPath : blogId
          options      :
            ValuePattern: /blogview/read/%1%
      class   : WFLink
  class     : WFDynamic
  properties:
    arrayController: '#module#Blog'
```

There's a lot going on in this setup! Let's break it down.

PHOCOA has a special mechanism for automatically creating UI widgets for each iteration in a loop. A special widget called `WFDynamic` is used to handle the creation of widgets dynamically, one for each item in the associated array controller. A `WFDynamic` should have one child, named <id of WFDynamic>Prototype, which is used as a prototype for each widget created. This prototype can have formatters, bindings, etc on it, and PHOCOA will create each widget and attach data bindings as appropriate based on your prototype.

Notice the binding on the titlePrototype; it has a ValuePattern option. This is a mechanism similar to `printf` to make it easy to contstruct strings dynamically. The value property of a UI widget supports multi-value bindings, which allows you to string together one or more distinct values along with a format string into a single value to be used by the widget. The value property is mapped to %1%, value2 to %2%, etc. This makes it very easy to create URL's on the fly within the bindings mechanism.

Code Page HTML

Now, let's create the view code needed to display our list. Edit `list.tpl`:

```
<table>
{section name=posts loop=$__module->valueForKeyPath('Blog.arrangedObjectCount')}
    <tr>
        <td>{WFView id="title}</td>
        <td>{WFView id="postDts"}</td>
    </tr>
{sectionelse}
    <tr><td>No blog posts.</td></tr>
{/section}
</table>
```

This simple bit of smarty creates our entire blog post list. Notice the $__module variable used to generate the loop count for section. PHOCOA automatically assigns a few variables to the template for you. See the `WFPage` API docs for a complete list.

Also notice that we're using Key-Value Coding to access the loop count from our array controller.

Reload the page, and you'll now see the working version of the last 10 blog posts. If you click on one of our links, you'll notice you get a 404 error since we haven't yet created the page to read the blog post. Let's do that now.

```
$ phocoa createPage
phing -f /Users/alanpinstein/dev/sandbox/phocoa/phocoa/phing/build.xml -Dusing.pho
Buildfile: /Users/alanpinstein/dev/sandbox/phocoa/phocoa/phing/build.xml

phocoa > prepareGeneral:
     [echo] PHOCOA framework base dir at: /Users/alanpinstein/dev/sandbox/phocoa/p

phocoa > prepareProject:
     [echo] 1
      [php] Evaluating PHP expression: $_ENV['_']
     [echo] PHOCOA project dir at: /Users/alanpinstein/dev/sandbox/blog/blog
[realpathexpandhome] Resolved /Users/alanpinstein/dev/sandbox/blog/blog/.. to /Use
     [echo] PHOCOA project container dir at: /Users/alanpinstein/dev/sandbox/blog
 [property] Loading /Users/alanpinstein/dev/sandbox/blog/blog/conf/build.propertie
 [property] Unable to find property file: /Users/alanpinstein/dev/sandbox/blog/blo

phocoa > createPage:
Page Name: read
     [exec] Executing command: /opt/local/bin/php /Users/alanpinstein/dev/sandbox/
     [exec] Writing read.tpl
     [exec] Writing read.yaml
     [exec] Done!

BUILD FINISHED

Total time: 2.9682 seconds
```

Create a Page to Display a Post

Our shared instances are already set up from the list page. So all we need to do is load the correct data in the page delegate for the read page.

For kicks, we'll implement this template in good old-fashioned smarty, just to show that you can seamlessly fall back on lower-level coding techniques if needed.

```
class module_blogview_read
{
    function parameterList()
    {
        return array('blogId');
    }
    function parametersDidLoad($page, $params)
    {
        // load the 10 most recent blog posts in desc order.
        $post = BlogPeer::retrieveByPK($params['blogId']);
        if (!$post) throw( new WFRequestController_NotFoundException("That post is
        $page->assign('blog', $post);
    }
}
```

Now we need to implement the template.

```
<h2>{$blog->getTitle()}</h2>
<p>{$blog->getPostDts()|date_format}</p>
<div>{$blog->getPost()}</div>
```

Our public-facing blog posts pages are now done. Let's move on to the home page!

Creating a Home Page Through Composition

Now let's create our home page. CD back to the top of the modules directory and run:

```
$ phocoa createModule
phing -f /Users/alanpinstein/dev/sandbox/phocoa/phocoa/phing/build.xml -Dusing.pho
Buildfile: /Users/alanpinstein/dev/sandbox/phocoa/phocoa/phing/build.xml

phocoa > prepareGeneral:
     [echo] PHOCOA framework base dir at: /Users/alanpinstein/dev/sandbox/phocoa/p

phocoa > prepareProject:
     [echo] 1
      [php] Evaluating PHP expression: $_ENV['_']
     [echo] PHOCOA project dir at: /Users/alanpinstein/dev/sandbox/blog/blog
[realpathexpandhome] Resolved /Users/alanpinstein/dev/sandbox/blog/blog/.. to /Use
     [echo] PHOCOA project container dir at: /Users/alanpinstein/dev/sandbox/blog
 [property] Loading /Users/alanpinstein/dev/sandbox/blog/blog/conf/build.propertie
 [property] Unable to find property file: /Users/alanpinstein/dev/sandbox/blog/blo

phocoa > createModule:
Module name: pages
Default page [blank for none]: home
     [exec] Executing command: /opt/local/bin/php /Users/alanpinstein/dev/sandbox/
     [exec] Writing pages/pages.php
     [exec] Done building module pages!
     [exec] Writing home.tpl
     [exec] Writing home.yaml
     [exec] Done!

BUILD FINISHED

Total time: 2.8443 seconds
```

Now that we have a home page, we want this page to actually load when we click the home link! To do this, we adjust the `defaultInvocationPath` on our `WFWebApplicationDelegate` object, in

`classes/MyWebApplicationDelegate.php`:

```
    function defaultInvocationPath()
    {
        return 'pages/home';
    }
```

Now we can click on our Home link and will be taken to our newly created home page.

Of course, our home page is currently empty. We want it to have a little introduction and then list the 10 most recent posts.

First let's add the list of the recent posts. Since we have already created a module to do exactly that, we can use the convenient UI widget `WFModuleView` to include that information. Edit the `home.yaml` file as follows:

```
topPosts:
  class: WFModuleView
  properties:
    invocationPath: 'blogview/list'
```

Now, let's create the home page template:

```
<h1>Welcome to the PHOCOA blog!</h1>
{WFView id="topPosts"}
```

Reload the home page, and you'll see the finished product.

Congratulations! You've now seen all of the steps involved in creating a basic web application in PHOCOA.

Chapter 4. Advanced Topics

This chapter contains a conceptual overview of additional PHOCOA programming topics.

(more to come)

Skin / Template System

The PHOCOA skin system provides a simple yet flexible way to control the template being used to render each page. Instead of having to include header and footer templates on each page, you actually design the layout in the skin system, and the skin where the "page body" (the result of the module rendering) goes in the skin. This separation of concerns makes it simpler to manage complex layouts and keeps your attention focused on the relevant part of your page design.

The skin system has three layers:

- *Skin Type* At the top level is the Skin Type. Each skin type can have its own distinct infrastructure of common components, which are managed by its delegate, implementing `WFSkinDelegate`. The skin type is not a layout in and of itself; rather it is just an implementation of the layout from a semantic perspective. The Skin Delegate provides an interface for the skins (explained below) to access the data common to the layout. Many applications have only one skin type. However, you might have different portions of your site that have different sets of data to present, and this is a reason that you might create a new skin type. For instance, you might create an additional skin type for the "Admin" interface of your web site since it has a different set of "components" that need to be displayed on each page.
- *Skin* A skin is an actual layout implementation of the elements defined by a *skin type*. Each *skin type* can have any number of *skin* implementations.
- *Skin Theme* Each *skin* can optionally be partitioned into themes. Themes provide an easy way to use the same basic layout (the Skin) while changing colorschemes, graphics packages, etc. This is done effectively by swapping out a css file and the "path" to the skin's assets (images, etc).
- *Template Type* Each *skin* by default has just one template file, `template_normal.tpl`. If you want minor variations of the same look and feel, such as for a popup or mobile version, this is the way to do it. Different template types of the same skin will still be "themed".

By default, a new PHOCOA application has a single Skin Type (simple) and two Skins (topnav, sidenav). topnav has only one theme, while sidenav has two themes set up.

The bundled "SkinInfo" module provides an interactive browser for the installed skins on any PHOCOA application to make it easy to understand and browse the skin infrastructure. This is also available at http://phococa.com/examples/skinInfo.

Determining When To Use The Various Aspects of the Skin System

How do you know when you should create a new skin type, or just a new skin, or just a new skin theme? Here are a few pointers:

- If you just want a slight variation on an existing look and feel (i.e., different header graphics, different colors), then you should use just one *skin type* with one *skin* and make a *skin theme* for each variation.

- If you are keeping the same look and feel, but want a slightly different layout (maybe for a pop-up window or a mobile version), then you should make a new *template type* for your skin.
- If you want a different look and feel, but need to display all of the same elements on the page, you should create a new *skin*.
- If you need to represent an entirely different set of elements on the web page, such as for an admin interface vs. the public interface, then you should create a new *skin type*.

Most applications will have a single skin type, and a single skin, with one or more themes, and maybe a few extra template types. Don't get too overwhelmed with the options. The flexibility in the skin system is primarily intended for CMS applications or large web applications with multiple "sections".

Authoring Skins

When authoring skins, you will need to know how the skin systems stores its files and also how to create URL's pointing to the themed CSS files and other assets (images, js, etc) of your skin.

For convenience, all files related to a skin are stored along with the skin delegate code. PHOCOA takes care of mapping incoming URL requests to the proper folders. All skins are stored under the `skins` directory at the top level of your PHOCOA project.

`skins/` - Contains only directories; each directory represents a *Skin Type*.

`skins/<skinType>/` - For each *skin type*, pick a unique name and create a directory.

`skins/<skinType>/<skinType>_SkinDelegate.php` - A php file containing exactly one class, named `<skinType>_SkinDelegate`, that is the `WFSkinDelegate` for the *Skin Type*.

`skins/<skinType>/<skinName>/` - Also in the *skin type* directory are other directories, one for each *skin* that can be used for the *Skin Type*.

`skins/<skinType>/<skinName>/<skinName>_SkinManifestDelegate.php` - The `WFSkinManifestDelegate` for the *skin* <skinName>.

`skins/<skinType>/<skinName>/` - Other files in here are the various tpl and css files used for this *skin*.

`skins/<skinType>/<skinName>/www/` - Web root of the *skin*. Nothing actually goes in this folder but other folders.

`skins/<skinType>/<skinName>/www/shared/` - Files that need to be accesible to the WWW and are shared by multiple themes of this *skin* go here.

`skins/<skinType>/<skinName>/www/<themeName>/` - Files that need to be accessible to the WWW and are specific to a *theme* go here. Each *theme* has its own folder to contain "themed" versions of resources. Typically every *theme* has the same set of resources, but of course customized for that *theme*.

PHOCOA automatically adds some template variables to make it easy to access *skin* assets.

- skinDir Absolute URL path to the `skins/<skinType>/<skinName>/www/<themeName>/` directory.
- skinDirShared Absoulte URL path to the `skins/<skinType>/<skinName>/www/shared/` directory.

- skinBody The raw output of the current PHOCOA request. This is where the "page content" goes.
- skinHead The raw output PHOCOA generates to go in the <head> section of the web page. You can "override" default head elements in your own skin by placing references to css, js, etc. after skinHead.
- skinThemeVars The data returned by your skin's `loadTheme($theme)` function.

In addition to these variables, PHOCOA also includes a Smarty plugin to generate the correct CSS reference to include themed CSS files. The syntax is:

```
{WFSkinCSS file="myFile.css" media="screen"}
```

The media attribute is optional.

You don't have to use this method to include CSS files. You can simply put them in your theme's www directory, or in the shared directory. The difference, however, is that if you use the themed css syntax, your CSS file will be processed as a template and thus have access to the skinThemeVars variable. This is particularly useful if the only difference between your different theme css files is just a different colorscheme, for instance. Instead of having to maintain multiple copies of nearly identical CSS files, you can just have one that is run through the themed CSS system, and simply subsitute the colorscheme info via template variables.

Previewing Skins

PHOCOA includes a useful utility for browsing and previewing installed skin types, skins, themes, and template types. See http://phocoa.com/webapp/examples/skininfo/skinTypes for a demo.

Including Module/Page Content in a Skin

At this time, skin template files cannot make use of PHOCOA widgets or other PHOCOA technologies for building web pages. However, a special Smarty plugin is provided to allow you to include the output of a module/page in a skin. This is useful for cases such as building menu navigation systems, including a login form on every page, etc.

PHOCOA ships with a module/page called "menu" which makes it easy to include a YUI-based menu system on your site. You simply include the menu module with the `WFSkinModuleView` plugin, passing the name of the "namedContent" you want to use from your skin, and optionally a 1 to indicate that your menu is a horizontal menu bar.

For example, this creates a horizontal menu bar with drop-down menus, using the content supplied by the skin's "mainMenu" content:

```
{WFModuleView invocationPath="menu/menu/mainMenu/1"}
```

Security / Authentication

PHOCOA includes a simple yet extensible security architecture. PHOCOA's WFAuthorizationManager is in charge of coordinating security and authentication.

PHOCOA maintains a session object which is an instance of `WFAuthorizationInfo`. This class contains basic information about the currently logged in user (if there is one), such as the user's id, is the user a "super-user", has the user authenticated "recently", etc. This is enough information for most applications, but if you have more complicated access control, you can subclass `WFAuthorizationInfo` and add any information you need for your system.

The basic granularity of security is at the module level. This assumption is made since modules typically include related pages which share the same access restrictions. Thus, usually a user that can access one page in a module should be able to access all others.

Access control to a module is determined by the `checkSecurity` method. The default implementation allows unrestricted access. To secure access to a module, you simply override the `checkSecurity` method in your WFModule subclass. The only parameter to this function is the current `WFAuthorizationInfo` instance, and you return either `WFAuthorizationManager::ALLOW` or `WFAuthorizationManager::DENY`. You can use whatever logic you want to in this method to determine access rights.

```
function checkSecurity (WFAuthorizationInfo $authInfo)
{
    if ($authInfo->isSuperUser()) return WFAuthorizationManager::ALLOW;
    return WFAuthorizationManager::DENY;
}
```

Of course, data security also is important at the object level. For instance, a logged-in customer should only be able to edit his own record. To implement security at this level, the programmer should check the credentials of the logged in user against the data being edited. If the logged in user should be denied access, this can be done simply by throwing a `WFAuthorizationException`.

PHOCOA automatically handles logins. When a client attempts to access a module, and the `checkSecurity` method returns `DENY`, PHOCOA will automatically handle the situation. If the user is already logged in, they will be shown an "Access Denied" message. If there is no logged in user, the client will be redirected to the login page, and upon successful login, will be redirected back to the original request.

PHOCOA's login system handles all of the details of login except for determining if a given user/pass is valid, and setting up the "permissions" for the user. Your application need only supply PHOCOA with a `WFAuthorizationDelegate` instance to provide that functionality.

Your application tells PHOCOA which `WFAuthorizationDelegate` instance to use. We recommend that you do this in your `WFWebApplicationDelegate`'s `initialize()` delegate method:

```
WFAuthorizationManager::sharedAuthorizationManager()->setDelegate( new MyAuthoriza
```

`WFAuthorizationDelegate` is an informal protocol that contains a number of methods which help PHOCOA automate authorization-related tasks. There is only one required method, `login()`:

```
object WFAuthorizationInfo login (string $username, string $password, boolean $pas
```

All this function needs to do is determine whether or not the username/password combo is valid, and then create a WFAuthorizationInfo instance for the user's session. If the login is not valid, then return NULL.

Notice the `$passIsToken` parameter; this is used for a "Remember me" feature. If the user chooses to preserve his login info on his computer, and the session times out, PHOCOA will automatically try to log in the user again with a stored token. If this parameter is true, the password will be a hash of the password stored in the user's cookie rather than the actual password.

Note

If you think need to use multiple delegates in your application, we recommend that you set a different `php_ini` "session.name" for each authorization delegate. Since PHOCOA stores the `WFAuthorizationInfo` instance in a session, it is important to ensure that the proper instance is recalled for the current request.

Chapter 5. Additional Resources

The resources in this chapter provide more educational information on using PHOCOA.

PHOCOA API Documentation

The complete PHOCOA documentation is available online at http://phocoa.com/docs/ [http://phocoa.com/docs].

Further Explorations

PHOCOA Key-Value Coding and Bindings Primer

PHOCOA bindings make it easy to show information from your Key-Value Coding Compliant objects, which includes Propel objects if you've implemented the Propel changes noted in the appendix.

You can use WFObject's valueForKeyPath() function to access properties of your objects. For instance, if you have a Book object, and want to access the birthdate of the author, you could use:

```
$authorBirthDate = $book->valueForKeyPath("author.birthDate");
```

Which is equivalent to:

```
$authorBirthDate = $book->getAuthor()->getBirthDate();
```

While it doesn't seem that advantageous from just looking at the code, it becomes much more powerful when coupled with PHOCOA bindings.

Let's say you want to have a web page that shows the details of a book. Normally, you'd have to assign the $book object to your template engine and then put something like {$book->getAuthor()->getBirthDate()} in your template code.

With PHOCOA, instead of assigning the $book object to your template, you instead make the $book object one of the instance variables of your module. Then, you set up a WFLabel object and bind the "value" of the WFLabel to the $book variable, and set a keyPath of "author.birthDate" as the modelKeyPath.

Now, even this doesn't seem that different. However, as you might be starting to notice, the keyPath for the WFLabel object isn't *programming*, it's *configuration*. So, if you want to change the value of the label, you just edit the configuration of the keyPath and the new value will be used.

Furthermore, because you're using PHOCOA GUI widgets, you get access to a host of additional functionality without having to add any code. Want to truncate the string after 30 characters? Just configure the "ellipsisAfterChars" property of the label. Want to hide the label in certain circumstances? Just bind the "hidden" property of the WFLabel to a function returning a boolean value and PHOCOA does the rest. Want the birthdate to show up as a properly formatted date? Just provide a formatter to the WFLabel. Want to use a formatString to combine multiple values? Use the ValuePattern binding! This is all done via the configuration file, not via coding.

Further Resources

Visit http://phocoa.com for the latest news, downloads, etc. There are also a mailing lists that you can

join at PHOCOA Mailing Lists [http://www.phocoa.com/webapp/public/pages/mailinglists].

Appendix A. Using PHOCOA With Propel

PHOCOA is designed to use Propel as an analog for Core Data. By making a 1-line change to the Propel code, your Propel objects will automatically be Key-Value Coding compliant, making it very simple to make your PHOCOA application interact with Propel.

To update your Propel to work with Phocoa, simply edit the propel/om/BaseObject.php file to make the Propel BaseObject a subclass of the PHOCOA WFObject base class:

```
abstract class BaseObject extends WFObject {
```

If you also want to use Key-Value Coding methods with the Peer classes, you will need to make 2 additional edits. First, edit the BasePeer object simlarly to BaseObject:

```
class BasePeer extends WFObject
```

Next, add the following attribute to the database tag schema.xml:

```
basePeer="BasePeer "
```

Notice the SPACE at the end of BasePeer. This is a necessary trick with Propel 1.3, as if the basePeer is "BasePeer" the build scripts don't have Base<Object>Peer extend anything.

That's it! Your Propel install is now fully compatible with PHOCOA to act as the data store for your objects.