

The Book of Recess

Official Guide to the Recess PHP Framework

The Book of Recess: Official Guide to the Recess PHP Framework

Copyright © 2009 The Recess PHP Framework

Table of Contents

1. Introduction	1
I. Getting Started	2
2. Installing Recess	3
Requirements	3
Downloading & Installing Recess	3
Installing to Lighttpd	4
Recess Release Log	4
3. The Structure of Recess Applications	6
4. Hello World	7
Starting an Application with Recess Tools	7
Creating a New Application Manually	10
The Controller & View	11
Routing	11
The Request Object	11
The Controller-View Relationship	12
The Response	13
The urlTo Helper	14
II. Writing a Recess Application	16
5. The Project	17
III. The Recess Framework	18
6. Recess Core	19
recess.lang as a language extensions to PHP	19
The Library Class & Autoloader	19
The Object Class	21
Hooks in Object while expanding Annotations and shaping ClassDescriptor	22
Attached Methods	22
!Wrappable Methods and the IWrapper Interface	24
The Mechanics of Wrapped Methods and Wrappers	27
Combining Method Wrappers with IWrapper's combine method	28
Using the !Before and !After Annotations	30
Annotations	30
Authoring an Annotation	30
7. Recess Framework	33
8. Routing Requests	34
Simple Routing Techniques	34
What is routing?	34
Parametric Routes	34
Multiple Routes per Method	35
Keeping it DRY	35
Advanced Routing Techniques	35
HTTP METHOD Aware Routes for RESTful Routing	35
Relative Routes	36
Implicit Routes	36
Recess Tools & Routing	37
Routing Performance	37
9. The Recess Controller	38
10. Views and Templates	39
A Leading Example	39
Fundamental Concepts	45
Registering & Selecting Views	45

View	46
Templates	46
Assertive Templates	46
Helpers	47
Views by Example with Layouts	47
Views	48
AbstractView	48
NativeView	48
LayoutsView	48
JsonView	48
Implementing Custom Views	48
View Helpers	49
Loading with View's loadHelper	49
Html	49
Url	49
Layout	49
Part	49
Buffer	49
Blocks	50
Abstract Block	50
HtmlBlock	50
PartBlock	50
11. View Helpers	51
12. Models	52
Generating Models with Recess Tools	52
Querying Models	52
Persisting Model state with insert , update , save , delete	53
13. Relationships Between Models	55
A Preview	55
Relationships	56
Naming Conventions	57
Model to Table	57
!BelongsTo Relationship	57
!HasMany Relationship	58
!HasMany , Through Relationship	59
14. Plugins & Extensibility	61
15. RESTful APIs in Recess	62
IV. Deploying Recess Applications	63
16. Deploying Applications	64
17. Production Mode	65
V. Contributing to Recess	66
18. Forking on GitHub	67
Working with Recess as a Git Submodule	67
19. Setting up PHPUnit for Recess	68
20. Useful Git Commands	69
21. Submitting Bug Reports	70
22. Contributing to Documentation	71
Index	72

Chapter 1. Introduction

Part I. Getting Started

Chapter 2. Installing Recess

Requirements

The minimum requirements to run Recess are:

Minimum requirements:

1. Apache 2.x
2. PHP > 5.2.4

Recommended requirements:

1. Apache 2.2 [<http://httpd.apache.org/>]
2. mod_rewrite [http://httpd.apache.org/docs/2.2/mod/mod_rewrite.html]
3. PHP >= 5.2.9 [<http://www.php.net/downloads.php>]
4. Advanced PHP Cache [<http://us2.php.net/apc>]

Downloading & Installing Recess

Recess can be downloaded from the Recess PHP Framework Web Site [<http://www.recessframework.org/>].

For the Edge Development branch the Recess Framework source tree is hosted at GitHub at <http://github.com/recess/recess>.

Figure 2.1. Steps to Install Recess

1. Unzip contents into your web documents path (i.e. public_html)
2. On a development machine make these directories writeable by PHP:
 - apps/
 - data/temp/
 - data/sqlite/
3. Open recess-conf.php and set RecessConf::\$defaultDatabase
 - If using MySQL: Uncomment the 'mysql:...' line and fill in DBNAME/USER/PASS
 - If using Sqlite: Uncomment the 'sqlite:...' line and name the database
4. Do you have mod_rewrite?
 - Yes: Open your browser to the location you unzipped
 - No: Open your browser to the location you unzipped followed by index.php
5. If you see "Welcome to Recess!" we're ready to rock.
6. The URL you are currently at will be referenced { \$installUrl }

Installing to Lighttpd

Recess makes use of .htaccess files to protect application code on Apache servers. Lighttpd does not make use of Apache style .htaccess but instead has a centralized configuration file. The following \$HTTP["url"] settings have been used to secure Recess on a lighttpd server:

Example 2.1. lighttpd configuration settings

```
-- protect the default source code layout under Lighttpd
$HTTP["url"] =~ "^/recess/|^/plugins/|^/data/|^/apps/" {
    url.access-deny = ("" )
    $HTTP["url"] =~ "/public/" {
        url.access-deny = ("~", ".inc")
    }
}
```

Recess Release Log

- **Version 0.12 on 2009-03-31**
 - jQuery included in distribution for off-line support
 - Fixed sporadic behavior when text editors like emacs create temporary files in src directories
 - Improved installation support detecting availability of PDO, SqlitePdo, and MySql PDO

- Many other minor bug fixes
- **Version 0.11.1 on 2009-02-18**
 - Unit Tests Support PHPUnit
 - Existing tests migrated from SimpleTest to PHPUnit
 - Smarty moved to an external package
 - Regression fix from 0.11
- **Version 0.11 on 2009-02-17**
 - Numerous bug fixes
 - Improved MySQL support
 - Scaffolding improvements
 - Regression fix from 0.11
- **Version 0.10 on 2008-12-25**
 - First public release of the Recess PHP Framework.

Chapter 3. The Structure of Recess Applications

A Recess application is broken down into three big categories of components: Models, Views, and Controllers.

Applications are stored in their own directory structure which is by default `/apps`. The Recess Framework has the ability to host multiple apps simultaneously. This is different from the Ruby on Rails or CakePHP model you may be familiar with. Within the apps directory there are subdirectories for each application: `/apps/{appName}/`, for example: `/apps/myBlog/`.

Within an application's directory the structure is as follows:

- `apps/`
 - `myBlog/`
 - `models/`
 - `views/`
 - `controllers/`
 - `MyBlogApplication.class.php`
 - `myKillerApp/`
 - `models/`
 - `views/`
 - `controllers/`
 - `MyKillerAppApplication.class.php`

The `MyBlogApplication.class.php` contains a class named, as you may guess, `MyBlogApplication`. `MyBlogApplication` extends from the class `Application` and is where application wide settings are specified.

In general the coding style of Recess encourages classes to be contained in their own file and requires these files to have the extension `.class.php`. This is an important convention for the Recess Library which handles the intelligent and high-performance inclusion of class files. This is kind of neat.

Chapter 4. Hello World

Starting an Application with Recess Tools

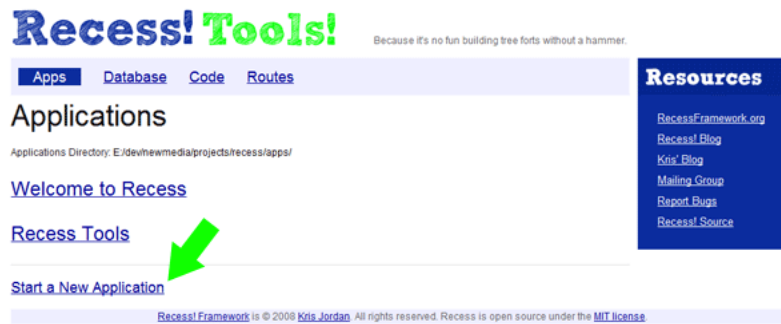
Let's create a simple, new application manually with the automated Wizard in Recess Tools, and then we'll walk through step-by-step what this wizard has done to set up a new application.

The New App Wizard in Recess Tools

To stay true to programming pedagogy let's create a "Hello World!" application.

Lets *open up Recess Tools* in a browser by navigating to the directory you installed recess to, /recess. This is probably `http://localhost/recess/`

Figure 4.1. Recess Tools - Start a new Application

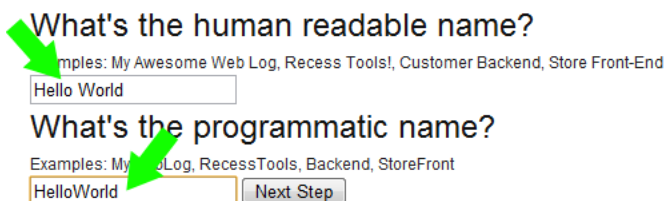


The wizard kicks off by asking us for two names, first a human readable name. This is used by Recess Tools to identify the application. *We'll name this application simply "Hello World"*. The programmatic name is what the wizard will use when generating the `Application` class and directory structure. *The programmatic name should not have spaces and should be a valid PHP identifier, like `HelloWorld`*.

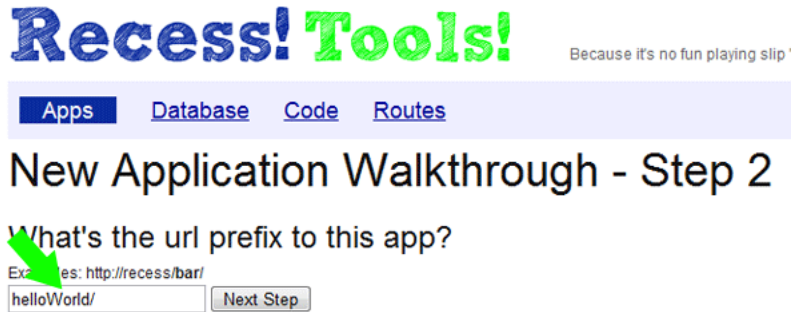
Figure 4.2. Naming an Application

New Application Walkthrough

Ready to start a new application? Great! This walkthrough is designed to step you through the process.



Next we'll select a URL prefix for the application. *A URL prefix determines how Recess will map a request URL to your application.* Recess Tools, for example, has the prefix `recess/`. For hello world lets use `helloWorld/`.

Figure 4.3. An Application's URL Prefix

At this point the wizard will generate a directory structure and some files that form the skeleton of an application. We will walk through what the wizard generates in detail in the following section. Let's keep going through the wizard which is asking us to place a string into the `RecessConf::$applications` array. The string `'helloWorld.HelloWorldApplication'` references the `HelloWorldApplication` class which the wizard generated which is located in the `apps/helloWorld` directory.

Figure 4.4. Installing the application in `recess-conf.php`

Last Step: Activate `helloWorld.HelloWorldApplication` in `recess-conf.php`

To enable your application open the Recess! config file: `E:/dev/newmedia/projects/recess/recess-conf.php`

Find the `RecessConf::$applications` array and add the following application string:

```
view plain copy to clipboard print ?
RecessConf::$applications
= array(
    'recess.apps.tools.RecessToolsApplication',
    'helloWorld.HelloWorldApplication', // <-- ADD THIS LINE
);
```

A green arrow points to the line `'helloWorld.HelloWorldApplication', // <-- ADD THIS LINE`.

Did you add that line? Great! Have fun building [Hello World!](#)

Open your `recess-conf.php` file and find the `RecessConf::$applications` array. Add the line to the array as instructed. This array of strings tells Recess which applications are installed. Your `recess-conf.php` file should look like this:

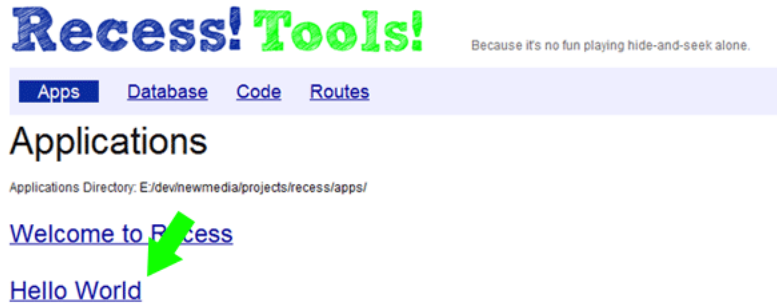
Figure 4.5. `recess-conf.php` Post-install

```
14 // RecessConf::DEVELOPMENT or RecessConf::PRODUCTION
15 RecessConf::$mode = RecessConf::DEVELOPMENT;
16
17 RecessConf::$applications
18     = array(
19         'recess.apps.tools.RecessToolsApplication',
20         'welcome.WelcomeApplication',
21         'helloWorld.HelloWorldApplication',
22     );
23 RecessConf::$defaultTimeZone = 'America/New_York';
24
25 RecessConf::$defaultDatabase
26     = array(
27         'sqlite:' . $_ENV['dir.bootstrap'] . 'recess/sqlite/defa
28         //mysql:host=localhost;dbname=DBNAME', 'USER', 'PASS'
```

A green arrow points to the line `'helloWorld.HelloWorldApplication',` in the `RecessConf::$applications` array.

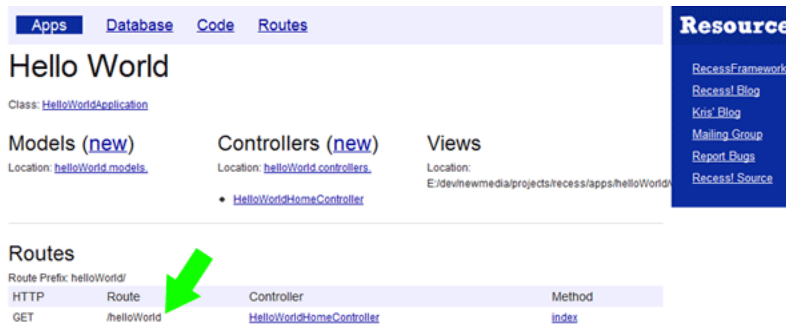
After saving the `recess-conf.php` file we have finished installing our 'Hello World' application. By navigating to 'Apps' we'll see our application 'Hello World' installed on the list of applications. Go ahead and follow the link.

Figure 4.6. Viewing an App in Recess Tools



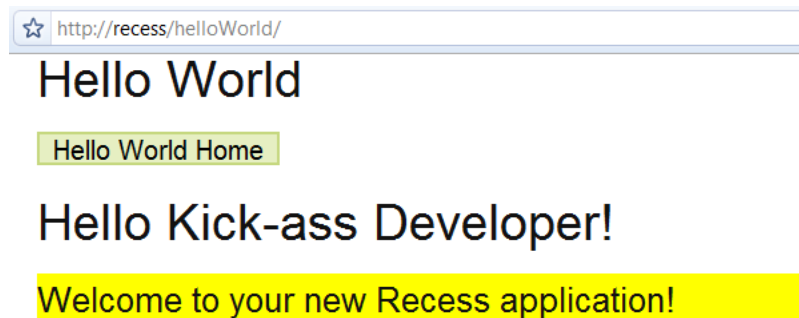
When viewing an application in Recess Tools we can see the Models, Controllers, Views, and Routes. Note the new application wizard created a HelloWorldHome controller for us.

Figure 4.7. Hello World Application



The routes section shows us which URLs our application will respond to and which method in a controller is being mapped. We can see the url `helloWorld/` will call the `index` method in our `HelloWorldHomeController` class. In a new tab try navigating to the location you installed Recess followed by `/helloWorld/` (probably `http://localhost/helloWorld/`). You should be greeted with the default new application landing page:

Figure 4.8. The Generated Hello World Application



We've now got a simple hello world application. In the next sections we'll explore what's going on in the Controller and View which the wizard generated.

Creating a New Application Manually

Note: This Section is Details Oriented

This section describes the files that Recess Tools will create automatically when starting an application. You can safely skip this section if you are not interested in those details right now.

Behind the scenes of the new application wizard a couple of steps happen to kick off a new application. There's nothing to stop developers from doing this manually should the need arise.

The first step is creating the application directory structure. After installing Recess open up the 'apps' directory. Create a new sub-directory with a name for your application like 'helloWorld'. Directories in Recess are typically named using the camelCaseConvention.

Now create the following 'views' and 'controllers' sub-directories so your structure looks like this:

- apps/
 - helloWorld/
 - models/
 - views/
 - controllers/

After creating the directory structure, *the next step is creating a sub-class of Application.* This class holds settings specific to the application such as the location of views in the directory structure and the prefix of models and controllers in the class path. In the apps/helloWorld/ directory create a new file named HelloWorldApplication.class.php. The '.class.php' extension is an important distinction for the Recess Library to know the file contains a PHP class named HelloWorldApplication. The class should be specified as follows:

Example 4.1. The HelloWorldApplication

```
<?php
Library::import('recess.framework.Application');

class HelloWorldApplication extends Application {
    public function __construct() {
        $this->name = 'Hello World';
        $this->viewsDir = $_ENV['dir.apps'] . 'helloWorld/views/';
        $this->modelsPrefix = 'helloWorld.models.';
        $this->controllersPrefix = 'helloWorld.controllers.';
        $this->routingPrefix = 'helloWorld/';
    }
}
?>
```

Now that the HelloWorldApplication class is setup our last step is to 'install' the application into the recess-conf.php file. Within recess-conf.php there is an array of strings named RecessConf::\$applications. These strings point to application classes in the class path. The apps folder is in the Library's class path so we can reference the newly created HelloWorldApplication using 'helloWorld.HelloWorldApplication'. If we were to put the HelloWorldApplication class in a

subdirectory called `app` within the `helloWorld` subdirectory the fully-qualified class name would be `helloWorld.app.HelloWorldApplication`. The convention of single classes per file and directories being broken up with dots is an influence from Java/C# namespaces. `Library` is covered in later documentation.

The Controller & View

In `Recess` a `Controller` is an object responsible for taking a web request, invoking the intended logic in a `Model`, and then selecting a `View` which will generate the response, likely in HTML, to be sent back to the client.

Having kicked-off a `HelloWorldApplication` in `Recess Tools`, let's take a quick look at some basic concepts.

In an `apps/helloWorld` directory there is a subdirectory `controllers` where `Controllers` are placed. Creating a new controller involves creating a new PHP file containing a class which extends `Controller`. Here is an example:

Example 4.2. A functionless Controller

```
<?php
class HelloWorldController extends Controller { }
?>
```

The file will be named `HelloWorldController.class.php`. The `.class.php` extension is important in `Recess` because it implies to the `Recess Library` that the file has only classes defined in it.

Let's add some functionality to the controller. We will begin by adding a simple method to print 'Hello World' and exit.

Example 4.3. Hello World! in Recess

```
<?php
class HelloWorldController extends Controller {
    function printIt() {
        print 'Hello World!'; exit;
    }
}
?>
```

If we now navigate to `helloWorld/printIt` we will see 'Hello World!' printed. How did the request for `helloWorld/printIt` wind up getting mapped to the `printIt` method? This process is called `Routing`.

Routing

`Routing` is the subsystem in `Recess` that maps a URL to a method in your controller. For more information on routing details refer to this article on `Routing`.

The Request Object

An HTTP request contains a variety of information: variables, headers, cookies, a URL, etc. `Routing` takes care of mapping an HTTP method (`GET/POST/PUT/DELETE`) to a controller method. Within your controller method you'll likely need to perform logic based on information in the request. `Controllers` have a `Request` object which can be referenced using `$this->request`. Let's take a look:

Example 4.4. Printing the Request object from a simple Controller

```
<?php
class HelloWorldController extends Controller {
    function printIt() {
        print $this->request->resource . '<br />';
        print_r($this->request);
        exit;
    }
}
?>
```

Now refresh `helloWorld/printIt` in your browser. If the output runs together in one line view the source. These are the variables available on the Request object. For example, `$this->request->resource` is `'/helloWorld/printIt'`. Try navigating to `'helloWorld/printIt?foo=bar'` and notice how `$this->request->get['foo']` is set to `'bar'`. In a Request object member variables `get`, `post`, and `put` hold the variables passed into the request when using the GET, POST, or PUT method like PHP's `$_GET` and `$_POST`.

The Controller-View Relationship

A Controller is responsible for indicating which view template to use. If no response is returned from a controller method the default view template will be `nameOfTheControllerMethod.php` in the `views/` directory. Try removing the body of the `printIt` method and refresh to get an error indicating no view template at `helloWorld/views/printIt.php` can be found. To change the view template file we can use the `ok` helper method.

Example 4.5. Returning an HTTP OK response and hinting the view template.

```
<?php
class HelloWorldController extends Controller {
    function printIt() {
        return $this->ok('the-view');
    }
}
?>
```

After refreshing the error message will change to indicate the new location `helloWorld/views/the-view.php`. The `ok` helper method will be discussed soon. Let's create a new file named `the-view.php` with the content below and save it into the `views` folder:

Example 4.6. A simple view template.

```
<html>
    <head><title>Hello World View</title></head>
    <body>
        <?php echo 'Hello World!'; ?>
    </body>
</html>
```

Try refreshing, you should see `'Hello World!'`. Lets pass some variables from the Controller to the View Template.

Example 4.7. Passing data from a controller to a view template.

```
<?php
class HelloWorldController extends Controller {
    function printIt() {
        $this->message = 'Hello World';
        $this->repeat = 10;
        return $this->ok('the-view');
    }
}
?>
```

Example 4.8. A view template taking input from a controller.

```
<html>
  <head><title>Hello World View</title></head>
  <body>
    <?php for($i = 0; $i < $repeat; $i++): ?>
      <?php echo $message, '<br />'; ?>
    <?php endfor; ?>
  </body>
</html>
```

Refresh to see 'Hello World!' printed 10 times. How do those variables propagate to the view template? The public instance variables in a controller are copied into the Response object which gets passed to a View, the View then sets those variables in the context of the view template. This process is vaguely similar to the Memento design pattern.

When writing view templates a great trick to quickly see what variables are available is to force an error in the view and look at the Recess Diagnostics screen. Try replacing the `for` loop in your view template with this code:

Example 4.9. An error in a view template invokes Recess Diagnostics

```
<html>
  <head><title>Hello World View</title></head>
  <body>
    <?php echo $fail; ?>
  </body>
</html>
```

This will trigger an undefined variable error in PHP that will bring up Recess Diagnostics. On the Diagnostics screen there is a 'Context' table that shows all of the variables available in the local context. Here we can see `$message` and `$repeat` set as well as some other variables, one of which is the Response object.

The Response

Controller methods are allowed to return either nothing at all or a Response object. When a controller method does not return anything Recess assumes an `OkResponse` is intended with a view template that has the same name as the controller method. The 'Ok' prefix of `OkResponse` corresponds to the HTTP 200 OK response code. A Recess Response object contains the information Recess needs

to respond to a request including: the response code, data to be passed to the view, headers to be sent back, cookies, a reference to the request, and some additional meta data used by Recess.

In the base `AbstractController` class there are a number of helper methods which will import and instantiate a response for you. The `ok` method is an example of a helper method which returns an `OkResponse`. Other helpers include: `conflict`, `redirect`, `forwardOk`, `forwardNotFound`, `created`, and `unauthorized`. The forwarding responses are a special kind of `Response`.

A `ForwardingResponse` causes Recess to handle another request and sends the body of that response to the client. For example, imagine you would like to build a PHP REST interface for creating Posts. After creating a Post you would like to send a 201 `CREATED` response that contains a `Location` header informing the client where to find that resource. For web browsers you likely want to send back meaningful content, perhaps the new list of Posts. In Recess this would look like:

Example 4.10. Responding with a `ForwardingResponse`

```
<?php
class PostsController extends Controller {
    /** !Route POST, /posts */
    function insertPost() {
        $post = Make::a('Post')->copy($this->request->data('Post'))->insert();
        return $this->created('/post/' . $post->id, '/posts');
    }
    /** !Route GET, /posts */
    function listPosts() {
        $this->posts = Make::a('Post')->all();
    }
    /** !Route GET, /posts/$id */
    function showPost($id) {
        $this->post = Make::a('Post')->equal('id', $id)->first();
    }
}
?>
```

The important line is `$this->created('/post/' . $post->id, '/posts');` The `created` helper method takes two arguments, the first is the URL to the created resource that will be sent in the `Location` header, the second is the URL to the 'content' to respond with. In this case the REST resource created is at `/posts/$id` but the response will render the HTML for the list of all posts at `/posts`.

The `urlTo` Helper

Introducing dependencies on specific URLs in your controllers (and views!) is a bad practice because these URLs may change due to refactoring. Recess decouples this knowledge by providing a helper method that returns the URL to a controller method. Lets take another stab at the `PostsController` using `urlTo`.

Example 4.11. Using the `urlTo` method in `PostsController`

```
<?php
class PostsController extends Controller {
    /** !Route POST, /posts */
    function insertPost() {
        $post = Make::a('Post')->copy($this->request->data('Post'))->insert();
        return $this->created(
            $this->urlTo('showPost', $post->id),
            $this->urlTo('listPosts'));
    }
    /** !Route GET, /posts */
    function listPosts() {
        $this->posts = Make::a('Post')->all();
    }
    /** !Route GET, /posts/$id */
    function showPost($id) {
        $this->post = Make::a('Post')->equal('id', $id)->first();
    }
}
?>
```

The `urlTo` helper method will return the URL which maps to the controller method passed as an argument. Notice that methods which take parameters must be passed the parameters as subsequent arguments as shown in `urlTo('showPosts', $post->id)`. Now if we change URLs using the relative routing techniques shown in the Routing Screencast [<http://www.recessframework.org/page/routing-in-recess-screencast>] we do not have to find all of the points where that URL was referenced. Also, if the name of a method changes and `urlTo('thatMethod')` is called Recess will throw an error which simplifies debugging.

Controllers in Recess simplify the process of accepting a request, delegating to application logic in Models, and passing off responsibility for responding to a view all in a RESTful manner. The conventions of selecting a view name based on the controller method name and returning a 200 OK response by default can be overridden with ease using helper methods. Finally, the `urlTo` method helps keep controllers and views DRY.

Part II. Writing a Recess Application

Chapter 5. The Project

Part III. The Recess Framework

Chapter 6. Recess Core

recess.lang as a language extensions to PHP

Recess takes a different approach from many PHP web frameworks. Fundamental to the Recess 'Framework' is Recess 'Core', a shallow layer of classes that rest just above PHP in the architecture of Recess applications which provide extensions to the PHP language. These classes, located in `recess.lang.*`, are web framework agnostic, as we will see in the examples through this chapter. The Recess Framework is built upon Recess Core. By extending Recess' `Object` class, additional functionalities like the ability to 'attach' methods to a class at runtime, define methods that can be 'wrapped' by objects implementing `IWrapper`, and use Recess-style `!Annotations` are all enabled for user-defined classes.

Where PHP stops short of providing language-level constructs for these useful capabilities found in other programming languages, Recess Core fills in the gaps with library-level support that is still distinctly PHP.

The Library Class & Autoloader

The `Library` class' purpose is to simplify the inclusion of dependencies between classes. Under the covers `Library` uses an autoloading mechanism and some other techniques to ensure high-performance in applications that use Recess Core's functionalities. These caching and script compilation techniques will be discussed in-depth later.

`Library`'s most useful static methods are `addClassPath` and `import`. `Library` maintains a list of directories it attempts to find an imported class in. The exact location of imported class files can be cached by `Library` to avoid hitting disk multiple times to locate single files. The order in which paths are added to `Library` using `addClassPath` is important. When attempting to import a class paths are checked from the most-recently added to the least-recently added using `addClassPath`. The Recess Framework uses this mechanism to allow classes defined in `plugins/` to take precedence over `recess/`, and classes in `apps/` to take precedence over `plugins/`.

`Library`'s `import` style is inspired by Java/C# imports and is a stylistic difference from libraries like the Zend Framework. Suppose the following directory structure:

```
/public_html
  /foo
    ClassA.class.php
  /bar
    ClassB.class.php
```

These classes can be imported using `Library` with the following snippet of PHP:

Example 6.1. Using Library's `addClassPath` and `import` methods

```
Library::addClassPath('/home/example/public_html');

Library::import('foo.ClassA');
$aClass = new AClass();

Library::import('foo.bar.ClassB');
$bClass = new BClass();
```

If you ever need to use `Library` and `Recess` classes in PHP files outside of the `Recess` Framework it requires setting up some environment variables and using `include` on `Library` directly. The following snippet can be placed into a php script and included by plain-old PHP files.

Note

`Recess'` convention is one class definition per file. PHP files that contain class definitions must use the same name for the file as the class' name, followed by the extension ".class.php". So, for a class defined with `class Foo { ... }` the filename would be `Foo.class.php`.

Example 6.2. `core-bootstrap.php` - Bootstrapping into `Recess` Core with `Library`

```
<?php
// core-bootstrap.php
// Bootstrap into Recess Core
date_default_timezone_set('America/New_York');

$_ENV['dir.bootstrap'] = str_replace('\\', '/', realpath(dirname(__FILE__))) . '/';
$_ENV['url.base'] = str_replace(basename(__FILE__), '', $_SERVER['PHP_SELF']);

$_ENV['dir.recess'] = $_ENV['dir.bootstrap'] . 'recess/';
$_ENV['dir.temp'] = $_ENV['dir.bootstrap'] . 'data/temp/'; // This directory must

require($_ENV['dir.recess'] . 'recess/lang/Library.class.php');
Library::addClassPath($_ENV['dir.recess']);
Library::addClassPath($_ENV['dir.bootstrap'] . 'plugins/'); // Add additional paths
Library::addClassPath($_ENV['dir.bootstrap'] . 'apps/');
?>
```

These directory paths may need to be tweaked depending on where you've extracted the `Recess` distribution. Explanations for each of these environment variables follows:

`$_ENV['dir.bootstrap']` - The absolute directory that contains the bootstrap file. This is a helper variable that simplifies defining the other environment directories by reference.

`$_ENV['dir.recess']` - The base directory for `Recess'` source files. This directory should contain two subdirectories: `recess/` and `test/`.

`$_ENV['dir.temp']` - A directory that is writable by PHP scripts. This directory is used by `Recess` to store temporary data such as cached data structures or compiled scripts.

`$_ENV['url.base']` - This is the base URL path that relative URLs should be constructed from. Outside of the `Recess` Framework this is less meaningful. `Recess` uses it for mapping routes, and generating URLs in view helpers.

The Object Class

`Object` is the base class for extensible classes in the Recess. `Object` introduces a standard for building a class descriptor through reflection and the realization of Recess Annotations. `Object` also introduces the ability to attach methods to a class at run-time and create wrappable methods.

`Object` is the superclass for the major components of Recess: `Models`, `Controllers`, and `Views`. The two most common types of classes created when using the Recess Framework are `Models` and `Controllers`. There is nothing prohibiting developers from extending `Object` in their own custom classes. In fact, if you are a developer feeling particularly ambitious and needing to roll a mini-framework with a different composition than Recess, starting with `Object` and other primitives like `Library` defined in Recess Core may make your job much more pleasant!

A primary purpose of any framework is to remove boilerplate code wherever possible. Object-oriented (OO) languages provide different means for removing code-duplication. The most commonly used techniques in well-known OO languages of Java and C++ are inheritance and composition. PHP has support for these common techniques and Recess makes use of them wherever possible. Other languages that support object-oriented programming like SmallTalk, Scala, and Ruby go beyond composition and inheritance and allow modules or traits to attach new units of functionality to class definitions (like properties and methods) dynamically. Recess Core provides a systematic way for simulating similar language features and allows for methods to be *attached* to classes that extend `Object` at run-time. This technique in combination with *annotations* is how the Recess ORM provides methods to access relationships, for example. Further discussion of *attached methods* see the section called “Attached Methods”.

Another form of extensibility in Recess that draws inspiration from a more sophisticated Aspect-Oriented Programming and the 'similar in spirit' Decorator OO pattern are *wrappable methods*. Classes extending `Object` can declare a method to be `!Wrappable`. Once wrappable, other classes implementing the `IWrapper` interface can dynamically wrap functionality around calls to the method. Key methods in the framework, like `Controller`'s `serve` and `Model`'s `insert`, `update`, `delete`, are wrappable. Why is this useful? It allows user-defined functionality to inject behavior just before or just after core framework method calls in a simple, standard way. For example, validations on a `Model` may take place just before `insert` or `update`, using wrappable methods the validation system does not need to be closely coupled to the `Model` class. Similarly with `Controllers`, authentication could happen before `serve` is called and the wrapping authentication class can subvert a request by redirecting to a login page. Further discussion on wrappable methods see the section called “!Wrappable Methods and the IWrapper Interface”

The final major unit of functionality exposed in `Object` is a system for realizing *Recess Annotations*. Annotations give programmers the ability to work in a more declarative, meta-programming style. What does this mean? Rather than telling PHP how to do something, you tell Recess what you want with an annotation and it is then up to Recess and the annotation's class to expand your declarative statement into PHP. Annotations can be written on three PHP constructs: classes, methods, and properties. Because PHP does not have first-class support for annotations, Recess Annotations are placed inside of doccomments, comments that begin with `/**`, which are a first-class construct in PHP available through PHP's Reflection API. Annotations often make use of attached and wrappable methods to inject units of functionality. For further discussion on annotations see the section called “Annotations”

Underlying *attached methods*, *wrappable methods*, and *annotations* is fundamental data structure, the `ClassDescriptor`. The `ClassDescriptor` is where the information used to describe Recess' language features like attached methods are stored. There is one `ClassDescriptor` per class and, when Recess is running in production mode, this data-structure is computed once and cached. Sub-classes of `Object` can use the class descriptor as a store for computed data structures by overriding hooks in the `Object` class. For example, `Model` uses this cache to hold database information and the meta-data

for relationships, and `Controller` uses it to store routing information. Annotations expand to 'shape' a class's `ClassDescriptor`. The following section describes the hooks available to sub-classes of `Object` for shaping class descriptors.

Hooks in Object while expanding Annotations and shaping ClassDescriptor

Initialize Class Descriptor - A class's descriptor may need to initialize certain properties. For example `Model`'s descriptor has default properties initialized for database table based on convention by the name of the class, primary key, etc.

`protected static function initClassDescriptor($class)` - Parameters: `$class` is the class' name as a string. Returns a `ClassDescriptor`.

Shape Descriptor with Method - Prior to expanding the annotations for a class method this hook is called to give a subclass an opportunity to manipulate its descriptor. For example `Controller` uses this in able to create default routes for methods which do not have explicit `Route` annotations.

`protected static function shapeDescriptorWithMethod($class, $method, $descriptor, $annotations)`- Parameters: `$class` string name of class whose descriptor is being initialized. `$method` is of type `ReflectionMethod`. `$descriptor` is the `ClassDescriptor`. `$annotations` is an array of `Annotations` found on method. Returns a `ClassDescriptor`.

Shape Descriptor with Property - Prior to expanding the annotations for a class property this hook is called to give a subclass an opportunity to manipulate its class descriptor. For example `Model` uses this to initialize the datastructure for a property before a `!Column` annotation applies metadata.

`protected static function shapeDescriptorWithProperty($class, $property, $descriptor, $annotations)` - Parameters: `$class` string name of class whose descriptor is being initialized. `$property` is of type `ReflectionProperty`. `$descriptor` is the `ClassDescriptor`. `$annotations` is an array of `Annotations` found on property. Returns a `ClassDescriptor`.

Finalize Class Descriptor - After all methods and properties of a class have been visited and annotations expanded this hook provides a sub-class a final opportunity to do post-processing and sanitization. For example, `Model` uses this hook to ensure consistency between `Model`'s descriptor and the actual database table's columns.

`protected static function finalClassDescriptor($class, $descriptor)` - Parameters: `$class` is the class' name as a string. `$descriptor` is the `ClassDescriptor` after all properties and methods have been visited. Returns a `ClassDescriptor`.

Attached Methods

Sub-classes of `Object` allow you to attach methods to a class dynamically at run-time. Once a method has been attached all instances of that class in existence, or instantiated thereafter, will have the method available. Attached methods also show up with reflection when using the `RecessReflectionClass`. Example usages of attached methods are: the implementation of relationship methods on models, and wrappable methods. To attach a method to a class you must first define the target method on another class. The following detailed example demonstrates attaching a method to a class.

Example 6.3. Attaching Methods to a sub-class of Object

```

<?php
include('core-bootstrap.php');
Library::import('recess.lang.Object');

class MyRecessObject extends Object {}

class AttachedBehavior {
    public $word = 'static';
    function targetMethod(Object $object , $lastWord) { echo "Hello, $this->word PHP"
}

$targetInstance = new AttachedBehavior();
$targetInstance->word = 'dynamic';
MyRecessObject::attachMethod('MyRecessObject', 'attachedMethod', $targetInstance,

$myInstance = new MyRecessObject();
$myInstance->attachedMethod('world');

// Output: Hello, dynamic PHP world!

```

The class we will attach a method to is `MyRecessObject`, it is important that it extends `Recess' Object` class.

The class `AttachedBehavior` contains the method we will attach to `MyRecessObject`: `targetMethod`. Attached methods must be defined within a class because attaching requires an instance of the class as well as the method name.

Notice that attached methods always take an instance of the class they are attached to as their first parameter. Subsequent parameters are the parameters to be passed by the attached method's call (6). We've created an instance of the `AttachedBehavior` class and given it some state. The ability to provide the method with some context is key to being able to do interesting things with attached methods on a class-by-class basis. For example, when a `!HasMany` or `!BelongsTo` annotation expands it attaches multiple methods to the class the annotation is defined on. Each of these target methods are defined on an instance of `HasManyRelationship` or `BelongsToRelationship` that have additional state, such as the relationship name, related table, foreign key, etc, that are vital state for the attached methods. One way to think of this is a poor man's closure.

The call to the static method `attachMethod` is where the magic mapping happens. `Object` defines `attachedMethod` which has four parameters. The first is an unfortunate one: the string classname the method is being attached to¹. The second parameter is the name we are assigning to the new method on our class. So, in this example the method's name will be `attachedMethod()`. Next we provide the target the attached method will call. We specify this by passing an instance of an object, and the string name of the method being attached.

We can now call `attachedMethod` on an instance of `MyRecessObject` and it will map to `targetMethod` on our `AttachedBehavior` instance.

Underneath the covers there is some simple indirection taking place in `Object`'s `__call` method that maps the attached method call to the target method based on entries in the `ClassDescriptor`. If methods are attached during the process which sub-classes of `Object` expand annotations and build-up `ClassDescriptors` then the target instance, its state, and the mapping will be automatically cached in production mode. For more information on `ClassDescriptor`, see the section called "Hooks in `Object` while expanding Annotations and shaping `ClassDescriptor`"

!Wrappable Methods and the IWrapper Interface

Wrappable methods are special methods on an Object class whose invocations be dynamically 'wrapped' with new behavior before and/or after the invocation of the wrapped method. For Python coders method wrappers are similar to method decorators, for Aspect-Oriented folks this is a poor man's join-point, for everyone else wrappable methods are a flexible extensibility point where you can flexibly inject new behavior.

Consider inheritance in traditional object-oriented programming. By overriding a method in a sub-class you can effectively wrap behavior around the parent class' method call. Here is some example code:

Example 6.4. Wrapping a Method Using Traditional Inheritance

```
<?php
class MyBaseClass {
    function foo() {
        echo "Foo!\n";
        return true;
    }
}

class MySubClass extends MyBaseClass {
    function echoFoo() {
        echo "Before!\n";
        $result = parent::foo();
        echo "After!\n";
        return $result;
    }
}

$obj = new MySubclass();
$obj->echoFoo();
// Output:
// Before!
// Foo!
// After!
?>
```

MyBaseClass defines a method `foo` that will be 'wrapped' using inheritance.

`foo` is overridden by `MySubClass` and first echos a message.

Now, we call the 'wrapped', or in this case overridden, parent class' `foo` method.

Finally, let's print an after message once the call to `MyBaseClass`' `foo` has returned.

In this simple example we've effectively wrapped `MyBaseClass`' `foo` method with some new behavior: we print 'Before!' and 'After!' `foo` is called on instances of `MySubClass`. What is so bad about pure inheritance?

- In many cases: Nothing, when you can use it, use it! If you can override methods to implement the functionality you need it is the best, most efficient way of 'wrapping' behavior around a method. Sometimes, though, it's painfully inflexible.
- Inflexibility - Inheritance is heirarchical. Our goal in `MySubClass` above was to print messages before and after a method call. Let's say we also wanted to log the outcome of the call to `foo`. We could create

a new `LoggedSubClass` class that subclasses `MySubClass` and logs the result. Now we're ready to deploy and we want to get rid of the wrapped echo'ing behavior, to do so we must either change the parent class of `LoggedSubClass` or comment/toggle out the behavior in `MySubClass`.

- Duplication of Code - Suppose we wanted to have printing and logging capabilities wrap a bunch of method calls on a bunch of other classes. Without reorganizing the entire inheritance hierarchy we must duplicate similar code in many places around the application.

Other languages and programming paradigms have a fairly simple solution to these limitations, as mentioned Python's decorators and AOP's join-points, that Recess looked towards for inspiration. Recess' solution is called 'Wrappable' methods. A `!Wrappable` method can be wrapped by, (or decorated with), unbounded `IWrappers` that can be composed dynamically at runtime. Let's take a look at how we could make `foo` a wrapped method:

Example 6.5. Specifying a Wrappable Method

```
<?php
include('core-bootstrap.php');
Library::import('recess.lang.Object');

class MyBaseClass extends Object {
    /** !Wrappable foo */
    function wrappedFoo() {
        echo "Foo!\n";
        return true;
    }
}

abstract class PrintWrapper implements IWrapper {}

$obj = new MyBaseClass();
$obj->foo();

// Output:
// Foo!
```

The `!Wrappable` annotation can be used on methods to create a 'wrappable' method. Notice that `foo` follows `!Wrappable`, this indicates that the wrappable method will be invoked with the name `foo`.

The actual name of the method is `wrappedFoo`. The only restriction on naming the wrapped method is that it can not have the same name as the wrappable method. By convention Recess prefixes wrapped methods with `wrapped`. I.E. `foo()` wraps `wrappedFoo()`

We can now invoke a method `foo()` on instances of `MyBaseClass`. Underneath the covers wrappable methods are surfaced using attached methods, for more info on attached methods see the section called "Attached Methods".

Great! We have a wrappable `foo` method, now let's wrap it with printing behavior. All we need to do is create a `PrintWrapper` class that implements the `IWrapper` interface.

Example 6.6. Implementing IWrapper

```
// ... replace starting at previous example's definition of PrintWrapper ...

class PrintWrapper implements IWrapper {
    function before($object, &$arguments) {
        echo "Before!\n";
    }

    function after($object, $returns) {
        echo "After!\n";
        return $returns;
    }

    function combine(IWrapper $wrapper) { return false; }
}

MyBaseClass::wrapMethod('MyBaseClass', 'foo', new PrintWrapper());

$obj = new MyBaseClass();
$obj->foo();
$obj->wrappedFoo();

// Output:
// Before!
// Foo!
// After!
// Foo!
```

`PrintWrapper` implements the three methods defined in the `IWrapper` interface. Let's discuss each:

`before` is called before the wrapped method is called. It is passed a reference to the object the wrappable method has been called on, as well as an array of the arguments passed to the wrappable method. The arguments are passed by reference so that wrappers can transform what is eventually passed to the wrapped method. Not shown here, but if a wrapper's `before` method returns a non-null value it will short-circuit the call and return the value immediately to the callee without calling the wrapped method.

`after` is called when the wrapped method returns. It is passed a reference to the object the wrappable method has been called on, and the return value of the wrapped method. The value `after` returns is passed to the callee so `after` has an opportunity to transform or replace the value returned from the wrapped method.

`combine` is an optimization method that will be discussed shortly.

Here we use the static method `wrapMethod` to wrap a `PrintWrapper` instance on the wrappable method `foo` on `MyBaseClass`.

By invoking `foo` we first work our way through `PrintWrapper`'s `before()`, then call `MyBaseClass`' `wrappedFoo()`, and finally back out to `PrintWrapper`'s `after()`.

For the sake of being thorough we show that you can still call `wrappedFoo` directly. Because wrapped methods are implemented at the Recess Core library level and not directly in PHP there is no way around this.

We now have the same printing behavior as in our Inheritance example, but without using inheritance. The power and flexibility of wrappable methods and wrappers is that multiple wrappers can wrap a wrappable method. So we could create a logging wrapper that also wraps `foo` and easily flip either wrapper on or off, dynamically at run-time, without having to reorganize our class hierarchy. For more detail see the

section called “The Mechanics of Wrapped Methods and Wrappers”. Lets point out exactly how wrappable methods address the downsides of inheritance:

- Wrappable methods avoid the *inflexibility* of inheritance-based overriding because methods can be wrapped dynamically. Wrapping methods with new functionality does not affect the type system or class hierarchy.
- Wrappable methods encapsulate a behavior around a method call. This presents a new way for PHP programs to package functionality to fight *code duplication*. To use Aspect-Oriented Programming terminology you could think of wrappers as a means for separating crosscutting concerns. Cross-cutting concerns are tasks like logging/printing debug messages as shown in the example above, authorization and access control, etc.

Given these two specific characteristics wrappable methods and wrappers provide a natural extensibility model. Plugin-developers can implement `IWrappers` that application-developers can easily incorporate in their projects because there is no need to modify a class heirarchy and the plugin's wrapper behavior is encapsulated in a simple class. For application-developers, instantiating wrappers and applying them with the `Object::wrapMethod` API can be awkward and cumbersome. This is where Recess Core's annotations come to the rescue, annotations provide the perfect vehicle for making declarative statements about a class or method which then employ wrappers and attached methods to do the leg work under the covers. For more information on annotations, see the section called “Annotations”

The Mechanics of Wrapped Methods and Wrappers

Where in the code base are wrapped methods implemented? What is the exact logic for processing methods wrapped with multiple wrappers? The answers to these questions are the focus of this section.

The implementation of wrapped methods can be thought of as a combination of the Observer and Strategy design patterns with specific semantics. Wrappers are observers of wrapped methods who are notified before and after the wrapped method is called. The `before` and `after` aren't vanilla notifications, though, and can return values that affect the logic of the call similar to a strategy. The logic of wrapped method invocation is implemented in the `recess.lang.WrappedMethod` class. The `addWrappedMethod` in `recess.lang.ClassDescriptor` brings wrapped methods onto a class' descriptor, and, finally the `recess.lang.WrappableAnnotation` abstracts away the pattern of making a plain-old class method a wrappable method.

When a wrapped method is invoked, the following process occurs:

1. Statement *S* invokes wrapped method *M* on object *O* with arguments *A**.
2. Each wrapper's `before` method is invoked in the reverse order that the wrappers were added²(LIFO). `before` is passed, by reference, *O* and an array of *A**. The wrapper, thus, has an opportunity to get or set public state from *O* or any argument in *A**. If a wrapper's `before` does not return a value or returns the value `null` then the next wrapper's `before` is called until all wrapper's `before` methods have been called. If a wrapper's `before` returns a non-null value this value does not pass go and does not collect two hundred dollars, it short-circuits the wrapper call-chain and is immediately returned to statement *S*.
3. The call to the wrapped method *M* is made using the (potentially transformed) arguments *A**. *M* returns value *R*.
4. Each wrapper's `after` method is invoked in the order that the wrappers were added (FIFO). `after` is passed arguments *O* and *R* (*M*'s return value). If the call to a wrapper's `after` returns a non-null value then this return value, *R'*, will override *R* in the remaining wrapper's calls to `after`, else *R'* is set to *R*.
5. The value *R'* is returned to *S*.

Warning

While nothing will stop you as an `IWrapper` author from writing the following at design-time, it should be noted that these practices will most likely cause errors and headaches at run-time and are considered *really bad practice*:

- In `before`: changing the types of arguments in `A*`, or changing the number of arguments in `A*`. `A*` must remain such that using its elements to call method `M` will result in a valid method call with the arguments `M` expects.
- In `before`: returning a value of any type other than `M` could be expected to return.
- In `after`: returning a value `R'` of any type other than `M` could be expected to return.

Combining Method Wrappers with `IWrapper`'s `combine` method

At runtime each wrapper is an instantiated object. In production mode these objects are deserialized on every request. Reducing the number-of wrappers is a boost to performance in time (extra method calls are expensive) and space so `Recess` gives `IWrapper` authors a simple way to combine similar wrappers. Imagine you've just created a `!Required` annotation that application developers can place on properties of a `Model` to denote they are required for `insert` and `update`. Behind the scenes you've written a `RequiredWrapper` that takes the name of a property and in the `before` method checks to make sure the property's value is non-null. Each annotation would thus expand to wrap `insert` and `update` with a new instance of `RequiredWrapper` for every property on the model. That could mean a lot of `IWrapper` objects to call `before` and `after` on to check requiredness! (It would also mean you couldn't check more than one field for requiredness because of short-circuit returns!)

When wrappers are applied to a `WrappedMethod` using `addWrapper` the `WrappedMethod` first iterates through each of the existing wrappers and calls their `combine` method, passing in the new wrapper. If the existing wrapper determines it can combine its state with the new wrapper's state it will do so and return `true` which indicates to the `WrappedMethod` *"do not add this new wrapper to your list, I've taken on its duties"*. If all existing wrapper's `combine` method returns `false` the new wrapper will be added to the list of registered wrappers. Let's take a look at an example:

Example 6.7. Combining Wrappers

```

<?php

class RequiredWrapper implements IWrapper {
    protected $properties = array();

    function __construct($property) {
        $this->properties[] = $property;
    }

    function before($model, $args) {
        $missing = array();
        foreach($this->properties as $property) {
            if($model->$property === null) {
                $missing[] = $property;
            }
        }

        if(!empty($missing)) {
            print("The following properties are required: " . implode(", ", $missing));
            return false;
        } else {
            return null;
        }
    }

    function after($model, $returns) { return $returns; }

    function combine(IWrapper $that) {
        if($wrapper instanceof RequiredWrapper) {
            $this->properties = array_merge($this->properties, $that->properties);
            return true;
        } else {
            return false;
        }
    }
}

MyModel::wrapMethod('MyModel', 'insert', new RequiredWrapper('fieldA'));

MyModel::wrapMethod('MyModel', 'insert', new RequiredWrapper('fieldB'));
MyModel::wrapMethod('MyModel', 'update', new RequiredWrapper('fieldB'));

?>

```

We'll store the list of required property names in `$this->properties`.

Here we short-circuit return false, so that the wrapped method will not get called. In this naive wrapper we're simply printing the error message in code.

Returning null is not necessary, it is the same as not returning, shown to illustrate that if all required fields are non-null the call will pass through to the wrapped call just fine.

Here we combine the state of two `RequiredWrappers` after checking for type sameness.

If we can combine we return true, and `$that` is not added to the list of wrappers because we have combined its state with `$this`.

If we cannot combine we return `false`.

The result of this code is that there will be two `RequiredWrapper` instances, one for `insert` and the other for `update`. The `RequiredWrapper` for `insert` will contain two properties in its `$properties` array. It is important to note that new instances of wrappers should be wrapped for each method, for example, if the same instance of a new `RequiredWrapper('fieldB')` had been wrapped around `insert` and `update` then `fieldA` would be required for `update` as well because of `combine`. (*Note: This doesn't pass the sniff test and exposes too much guts. Maybe we could change the implementation of `addWrapper` in `WrappedMethod` to clone the `Wrapper` before adding it.*)

Using the `!Before` and `!After` Annotations

Annotations

Annotations can be introduced by extending the abstract class `recess.lang.Annotation`. *User defined annotation classes must end in the word 'Annotation'*. For example: to have a `!Protected` annotation the classname must be `ProtectedAnnotation`. There are four abstract methods which must be implemented by custom annotations: `usage`, `isFor`, `validate`, and `expand`. These methods are used by the framework to make working with user-defined annotations feel like a first-class PHP construct. Following are descriptions of the purpose for each annotation:

`usage` - Returns a string representation of the intended usage of an annotation.

`isFor` - Returns an integer representation of the type(s) of PHP language constructs the annotation is applicable to. Use the `Annotation::FOR_*` consts to return the desired result.

Example 6.8. Examples of Annotation's `isFor` Method

```
// Only valid on classes
function isFor() { return Annotation::FOR_CLASS; }

// Valid on methods or properties
function isFor() { return Annotation::FOR_METHOD | Annotation::FOR_PROPERTY; }
```

`validate($class)` - `validate` is called just before expansion. Because there may be multiple constraints of an annotation the implementation of `validate` should append any error messages to the protected `$errors` property. Commonly used validations helper methods are provided as protected methods on the `Annotation` class.

`expand($class, $reflection, $descriptor)` - The expansion step of an annotation gives it an opportunity to manipulate a class' descriptor by introducing additional metadata, attach methods, and wrap methods. Parameters: `$class` the classname the annotation is applied to. `$reflection` The `PHP Reflection(Class|Method|Property)` object the annotation is located on. `$descriptor` is the `ClassDescriptor` that the annotation is being expanded on.

The power of annotations comes in the `expand` step. An annotation can modify a class' descriptor which allows for methods to be attached to a class and available on all instances of a class, as well as "wrap" methods of a class marked as `!Wrappable`

Authoring an Annotation

To demonstrate authoring a custom annotation, let's create an annotation for methods on a `Controller` that should be protected by a protected cookie key of "secret" and value of "password". Our goal

is to be able to place `!CookieProtected` on a method in a controller and redirect users to an optionally specified path (else, `"/`) if they do not have the necessary cookie. We'll craft the annotation in two-steps, first we'll implement the abstract methods of `Annotation`, then we'll take combine the annotation with the previously discussed `WrappedMethods` to finish off the functionality of the `CookieProtectedAnnotation` by wrapping a `Controller`'s `serve` method.

Example 6.9. The Beginnings of the `!CookieProtected` Annotation

```
<?php
Library::import('recess.lang.Annotation');

class CookieProtectedAnnotation extends Annotation {
    public function usage() {
        return '!CookieProtected [optional/redirect/path]';
    }

    public function isFor() {
        return Annotation::FOR_METHOD;
    }

    protected function validate($class) {
        $this->minimumParameterCount(0);
        $this->maximumParameterCount(1);
        $this->validOnSubclassesOf($class, Controller::CLASSNAME);
    }

    protected function expand($class, $reflection, $descriptor) {}
}
?>
```

Let's walk through each method. We can see `usage` simply returns a string demonstrating how to properly use an annotation in code. `isFor` shows that this annotation can only be applied to methods, so `Recess` will throw an error if the annotation is placed on a class or a property of a class. This is what we want as we are trying to protect methods of a controller. `validate` is making use of helper methods defined in `Annotation` for validating the formation of the annotation. These validations are used to check things like the number of parameters, which classes the annotation is meaningful for (in this case, only on subclasses of `'Controller'`), etc.

The full list of helper methods for `validate` is: `acceptedKeys`, `requiredKeys`, `acceptedKeylessValues`, `acceptedIndexedValues`, `acceptedValuesForKey`, `acceptsNoKeylessValues`, `acceptsNoKeyedValues`, `validOnSubclassesOf`, `minimumParameterCount`, `maximumParameterCount`, `exactParameterCount`. To understand the meaning of 'keys', 'indexed values', 'keyless values', you must understand the way `recess` interprets annotations. `Recess` parses an annotation by doing the following: 1) find an annotation by looking for an `!Bang` followed by whitespace, `"!Bang"`. 2) Convert everything after the start of an annotation into a parameters array, `!Bang foo, key: bar` becomes `array("foo", "key" => "bar")`. 3) Instantiate the class named `BangAnnotation` and pass the parameters into the `BangAnnotation`'s `init` method. Thus, every subclass of `Annotation` has a `$parameters` property which is this array. The helper methods provide convenient ways of checking the contents of this array during `validate`.

Once `validate` has been run, if errors are found in the annotation's `$errors` property a `RecessErrorException` is thrown. If no errors are found, then the final step before `expand` is called is to make all keyed values in the `$parameters` array properties of the annotation instance and to append

all keyless values of `$parameters` to the `$values` property. For example, the annotation `!Bang foo, Key: bar` becomes: `$bang = new BangAnnotation(); $bang->init(array('foo', 'Key' => 'bar'))`; Finally, after `Recess` calls `expandAnnotation` (which, in turn, calls your user-defined `expand`), your `expand` can reference `$this->values[0]` to find `'foo'` and `$this->key` to get `'bar'`.

Note

During the `init` method of `Annotation` all array keys are converted to lowercase.

Chapter 7. Recess Framework

Chapter 8. Routing Requests

Simple Routing Techniques

What is routing?

Routing is the machinery that takes a requested URL path like `/product/23` and ‘routes’ or dispatches control to a specific point in your application. In most frameworks, including Recess, this is to a method in a `Controller`. Using a framework that has fast, flexible, RESTful routing is important because URLs are fundamental to how people, search engines, and web services interact with you web application.

Let’s dive right into some code and take a look at how we can set up a route to a method in a controller:

Example 8.1. A simple GET route.

```
<?php
TestController extends Controller {
    /** !Route GET, /hello/world */
    function aMethod() {
        echo 'Hello PHP Community!'; exit;
    }
}
```

What’s that funny stuff above the function? It’s a Recess `RouteAnnotation`. Recess annotations may look a bit strange but they’re really simple. They are written inside of `doccomments`, a language construct in PHP which begins with a forward slash and two asterisks. Recess annotations are *banging*. Literally, they start with an exclamation point, or, BANG! (as opposed to the `@`-symbol if you’re used to Java style annotations). The `Route` annotation has two parameters. The first is the HTTP method such as `GET`, `POST`, `PUT`, or `DELETE` and the second is the URL path.

Parametric Routes

When a part of the route is preceded with a dollar sign it becomes a method parameter. Here is an example:

Example 8.2. Simple parametric routes using route `$`parameters

```
<?php
class TestController extends Controller {
    /** !Route GET, /hello/$first/$last */
    function aMethod($first, $last) {
        echo "Hello $first $last!"; exit;
    }
}
```

Now if we browse to `/hello/PHP/Community` the browser will print “Hello PHP Community”. Parametric routes are often used with ID or primary key columns in a database. For example, if I were building a store in Recess I might have a Product Details page that used a route like: `!Route GET, /product/$id`

Multiple Routes per Method

Controller methods can have multiple routes. For example, we can combine the previous two methods into one:

Example 8.3. Multiple routes on a single controller method

```
<?php
class TestController extends Controller {
    /**
     * !Route GET, /hello/world
     * !Route GET, /hello/$first/$last
     * */
    function aMethod($first = "PHP", $last = "Community") {
        echo "Hello $first $last!"; exit;
    }
}
```

If you accidentally add a route that conflicts with another somewhere else in your app Recess will tell you where the conflict occurred. The Recess Diagnostics error screen shows you where in your code the conflict occurred.

Keeping it DRY

If you're familiar with Cake or Rails you may be wondering what is the upside to specifying routes in-line with my methods? The long and short of it is, it is more DRY. With a separate routes file you must duplicate the name of a controller and method which a route maps to. So if you refactor your controller code you must remember to go and update your routes file as well. By keeping the two together it's never a mystery what URL will take you to the controller method you're working on.

Advanced Routing Techniques

HTTP METHOD Aware Routes for RESTful Routing

Routing in Recess is HTTP method-aware. To demonstrate this we can have two controller methods mapped to the same URL but differing HTTP METHODS:

Example 8.4. Routing with different HTTP Methods

```
<?php
class TestController extends Controller {
    /** !Route GET, /same/url */
    function comingFromGet() {
        echo '<form action="/same/url" method="post">';
        echo '<input type="submit" /></form>'; exit;
    }
    /** !Route POST, /same/url */
    function comingFromPost() {
        echo 'POSTed!'; exit;
    }
}
```

The first method will handle a GET to the url `/same/url` and the second a POST to `/same/url`. For an actual demonstration of this running check out the screen cast at minute 7:00! Having a routing system aware of HTTP methods is one way in which Recess helps simplify RESTful application develop in PHP.

Relative Routes

In Recess, Routes can be relative to their context. What does that even mean? There are three logical levels of organization for the purposes of Routing. The most general level is the application. Recess allows multiple applications to be installed at once a la Django. Within an application there may be multiple controllers and a controller can have many methods.

Relative routes are different from absolute routes in that they do not begin with a forward slash. Check out the `!Route` annotation for the `world` method below. By adding a `!RoutesPrefix` annotation to a controller we will prepend any relative route in the controller with `hello/` so we can now reach the `world` method using `hello/world/`.

Example 8.5. Relative routing in controller routes to `hello/world`.

```
<?php
/** !RoutesPrefix hello/ */
class TestController extends Controller {
    /** !Route GET, world */
    function world() {
        echo 'Hello World!'; exit;
    }
}
```

Implicit Routes

If you're familiar with Rails or Cake you're probably wondering why I needed specify routes for the `world` and `universe` methods. In many frameworks these routes would be implicit. In Recess they can be implicit too. We can delete the route annotations and still access the methods in the same way. The important difference between the way implicit routes work in Recess and other frameworks is that Recess does not rely on the name of a controller to determine the route, but rather on the routing prefixes of the application and the controller.

Example 8.6. Implicit routing in controller routes to `hello/world`.

```
<?php
/** !RoutesPrefix hello/ */
class TestController extends Controller {
    function world() {
        echo 'Hello World!'; exit;
    }
}
```

Because we've set up a route prefix for the `TestsController` we can reach the `world` method by using the URL: `hello/world`. Implicit routes can have parameters too.

Example 8.7. Implicit routing with parameters.

```
<?php
/** !RoutesPrefix hello/ */
class TestController extends Controller {
    function world($first, $last) {
        echo "Hello $first $last!"; exit;
    }
}
```

Now, `hello/world/Michael/Scott` will rest in the `world` method being called and printing "Hello Michael Scott!"

Recess Tools & Routing

With routes being sprinkled throughout controllers don't you lose the ability to look in a single place and get a sense of all of the routes in your application?

Recess Tools is first class Recess application that runs in the browser and is designed to help you along in development mode writing apps. With Tools we can see all of the routes explicit and implicit, relative and absolute, for any given application. The table lists the HTTP method and route corresponding which map to a controller class and method. With Recess Tools you can get a global picture of Routes in your application.

Routing Performance

How in the world can you expect to get any kind of performance out of an app when you have to reflect over every single controller in order to know all routes?

This is a great question. Routing changes as shown in the screencast take immediate effect. This is because the screencast was taken while Recess was in development mode. By switching to deployment mode the routing computation would only have to happen once because the routes won't change. On the first request in deployment mode Recess will build up the routing data structure, a tree, and cache it either to disk or memory depending on what your server has available. Subsequent requests simply unserialize the routing tree and Recess is off to the races. Perf has been a top priority while developing Recess and this technique enables great performance while allowing your code to stay simple, nimble, and DRY.

Chapter 9. The Recess Controller

Chapter 10. Views and Templates

Your controller has done all the heavy lifting and taken care of the "business logic", now it's time to get back to your client with a meaningful response. Where controllers are all about serving the client by doing and fetching, views are all about responding by presenting and rendering.

How you actually respond likely depends on: 1) what the HTTP client asks for "I want /person/1 in JSON", "Ok, the controller gave me the data for Person 1, here it is in JSON", and 2) the best way for you to get the job done: maybe it's using Smarty templates, maybe it's an automatic response, or maybe it's plain old PHP.

This brings us to the design goals of Recess' View system, which have been revamped in 0.20:

- Flexible - The view system must simplify handling different response types: HTML, JSON, XML, PDF, etc.
- Extensible - Users can create their own Views to accomodate preferences with template engine(s) or automating common responses.
- Predictable - The logic for selecting a view should be straightforward. Assertive templates, covered later in this chapter, are also encouraged for improved predictability when authoring templates.

A Leading Example

Before diving into specifics let's take a leading example of creating the view components for a hypothetical listing of blog posts. Suppose we have a Post model that has properties: title, author, and body. Our controller code will look like this:

Example 10.1. Controller Code to send a List of All Posts to the View Layer

```
/**
 * !RespondsWith Layouts
 * !Prefix post/
 */
class PostsController {
    /** !Route GET, list */
    function index() {
        $this->posts = Make::a('Post')->all();
    }
}
```

We'll get into the details of the relevant annotations later, for now just take our word that the view class rendering the response will be `LayoutsView`, and it will be looking for the view template `post/index.html.php` found in the `apps/[appname]/views/` sub-directory. The variable `$postSet` will be available to this template which represents a set of `Post` models.

Our first-pass, naive template implementation of `post/index.html.php` may look like this:

Example 10.2. A Naive View Template Implementation of `post/index.html.php`

```
<html>
  <head><title>A List of Posts</title></head>
  <body>
    <ul>
      <?php foreach($posts as $post): ?>
        <li class="post">
          <span class="title"><?php echo $post->title ?></span> by
          <span class="author"><?php echo $post->author ?></span>
        </li>
      <?php endforeach ?>
    </ul>
  </body>
</html>
```

This works, but as our application grows in complexity we will want to remove redundant HTML to layouts shared by many views. Let's try creating a layout that this simple view can extend to remove common HTML.

Example 10.3. A Simple Layout: `master.layout.php`

```
<?php
Layout::input($title, 'string');
Layout::input($body, 'string');
?>
<html>
  <head><title><?php echo $title ?></title></head>
  <body>
    <?php echo $body ?>
  </body>
</html>
```

Example 10.4. Making `post/index.html.php` extend `master.layout.php`

```
<?php
Layout::extend('master');

$title = 'A List of Posts';

$body = '<ul>';
foreach($posts as $post) {
  $body .= '<li class="post">' .
    '<span class="title">' . $post->title . '</span> by ' .
    '<span class="author">' . $post->author . '</span>' .
    '</li>';
}
$body .= '</ul>';
?>
```

As you can see, we've extracted the redundant HTML to a master layout. You'll notice that the master layout specifies its inputs. The child/parent templates do not share scope, the parent must define which inputs it requires and the type the variable should be. This improves *predictability* because you know exactly what variables a parent layout is expecting. If your child template fails to provide a variable required by the

parent you will get a simple error message notifying you of the problem instead of an obscure 'variable not defined' message from somewhere deep in the parent template.

Unfortunately, our listing template has regressed. Storing the output of the list of posts in a string is painful. This is where `Buffer`'ing and `Block` save the day! `Buffer` is a class that uses PHP's output buffering to fill `Blocks`. The buffer fills a block. Let's take a look:

Example 10.5. Using `Buffer::to($bodyBlock)` in `post/index.html.php`

```
<?php
Layout::extend('master');
$title = 'A List of Posts';
?>
<?php Buffer::to($body); ?>
  <ul>
    <?php foreach($posts as $post): ?>
      <li class="post">
        <span class="title"><?php echo $post->title ?></span> by
        <span class="author"><?php echo $post->author ?></span>
      </li>
    <?php endforeach ?>
  </ul>
<?php Buffer::end(); ?>
```

Now we don't have to worry about string concatenation, we let `Buffer` fill our `$bodyBlock`. If we were to run this code we would get a type check error in the master layout: `$body` is expected to be a string, but now it is an instance of `HtmlBlock` which is a sub-class of `Block`. Why not just have `Buffer` fill strings? Shortly we'll see how `Block`'s type-hierarchy enables really powerful features. The immediate benefit, though, is that it allows layouts to specify they are expecting a block of HTML, not just a string. Let's update our master layout.

Example 10.6. Updating the `master.layout.php` to assert `$body` is a `Block`

```
<?php
Layout::input($title, 'string');
Layout::input($body, 'Block');
?>
<html>
  <head><title><?php echo $title ?></title></head>
  <body>
    <?php echo $body ?>
  </body>
</html>
```

Easy enough! Notice all we needed to change was the expected type on the input. We can echo a `Block` instance just like we can a string. Imagine we had some other places in our application where we'd really like to print out the list-item format of a `Post`. Enter: Parts to save the day! Parts are kind of like partial templates you may have used in other frameworks. Let's take a look at how we would define a part for `post`.

Example 10.7. Defining post/li.part.php

```
<?php
Part::input($aPost, 'Post');
?>
<li class="post">
    <span class="title"><?php echo $aPost->title ?></span> by
    <span class="author"><?php echo $aPost->author ?></span>
</li>
```

Notice the similarity with a layout: parts and layouts define their inputs. The term we coined for this style of a template is an *Assertive Template*. By being assertive about the inputs a template expects we can make working with these templates much more pleasant. Let's take a look at how we would use this part in `post/index.html.php`.

Example 10.8. Using a Part in post/index.html.php

```
<?php
Layout::extend('master');
$title = 'A List of Posts';
?>
<ul><?php
    foreach($posts as $post) {
        Part::draw('post/li', $post);
    }
?></ul>
```

We can 'draw' a Part by passing the path to the part in the views directory, minus `.part.php`, and the rest of the inputs in the order specified by the part. You pass a part inputs just like you call a function with arguments. Notice that the name of the variable passed to draw and the name of the input do not have to match, again, like functions. This is contrasted by most PHP partial libraries that would require calling the template with an array like `array('aPost' => $post)`.

You'll also notice we're no longer buffering `$body`. But the code still works! What's going on? By default, if your child template does not specify a `$body` block, any output will automatically create a `Block` named `$body` that will be passed to its parent layout.

Let's say we wanted to make it possible to append another class to the `post/li` part. We can create an additional input that is optional by passing a third argument to `Part::input` which is the default value of an input.

Example 10.9. Specifying a Default Input Value in post/li.part.php

```
<?php
Part::input($aPost, 'Post');
Part::input($class, 'string', '');
?>
<li class="post<?php if($class != '') echo " $class" ?>">
    <span class="title"><?php echo $aPost->title ?></span> by
    <span class="author"><?php echo $aPost->author ?></span>
</li>
```

Now, by default, the only class will be 'post', but others could be appended by passing an additional input to the part. Let's go back to our `post/index.html.php` template and make the additional class name of 'odd' be appended to every other item in our list. This way we can style every other post differently using CSS.

Example 10.10. Give odd posts an additional class of 'odd' in post/index.html.php

```
<?php
Layout::extend('master');
$title = 'A List of Posts';
?>
<ul><?php
    $i = -1;
    foreach($posts as $post) {
        if(++$i % 2) == 0) {
            Part::draw('post/li', $post);
        } else {
            Part::draw('post/li', $post, 'odd');
        }
    }
?></ul>
```

You can see by passing an additional input to `post/li` we are now appending the 'odd' class to odd posts. Suppose that throughout our project we use this even/odd classname technique in a number of places. Could we do any better and eliminate this redundancy? Yes, we can! We'll simply create a part for it.

Now, take a deep breath and hold on. Here comes the sexy stuff that makes our parts different from everyone else's partials. Earlier we alluded to there being some 'power' in passing around instances of `Block` instead of strings. You saw an example of how `Buffer` filled an `HtmlBlock`. There is a special `Block` for `Parts`, too, called `PartBlock`. Before we summon a `PartBlock`, though, let's first create the higher-order part that abstracts toggling between two blocks.

Example 10.11. Creating a Higher Order Part with `each-toggle.part.php`

```
<?php
Part::input($items, 'array');
Part::input($even, 'Block');
Part::input($odd, 'Block');
$i = -1;
foreach($items as $item) {
    if(++$i % 2) == 0) {
        $even->draw($item);
    } else {
        $odd->draw($item);
    }
}
?>
```

There are two interesting things going on here. First, notice that we're passing in two `Block` instances named `$even` and `$odd`. Second, notice that we're using the `draw` method of `Block`. Every `Block` has two methods: `__toString` and `draw`. For an `HtmlBlock` created using `Buffer`, `draw` doesn't take any arguments (but won't complain if you give them). For a `PartBlock`, though, passing arguments to `draw` will apply those arguments where you left off in creating the `PartBlock`. Well, then, how do you create a `PartBlock`?

Example 10.12. Creating PartBlocks for use in Higher-Order Parts

```
<?php
// Assume $post is already defined as an instance of Post
$partBlock = Part::block('post/li');
$partBlock->draw($post);
$partBlock->draw($post, 'odd');
// Is the same as:
$partBlock2 = Part::block('post/li', $post);
$partBlock2->draw();
$partBlock2->draw('odd');
?>
```

So, by using the `block` method instead of `draw`, the method returns a `PartBlock` with the parameters you've passed stored. You can think of this like *currying* in functional languages like Scala / OCaml. For the astute reader you'll notice we have a problem on our hands, though. In order to use our 'each-toggle' part `$post` must be the final argument passed to `draw`. We can solve this in one of two ways, first we could do the obvious and re-order the inputs so that our 'post/li' is passed a class string first and a `Post` second. That makes things ugly, though, because we really want class to be an optional parameter. Recess has a mechanism for assigning optional arguments to a `PartBlock` after it has been created. Let's take another look.

Example 10.13. Specifying Out-of-order Inputs in PartBlocks for use in Higher-Order Parts

```
<?php
// Assume $post is already defined as an instance of Post
$partBlock = Part::block('post/li');
$partBlock->draw($post);
$partBlock->class('odd')->draw($post);
// Is the same as:
$partBlock2 = Part::block('post/li', $post);
$partBlock2->draw();
$partBlock2->class('odd')->draw();
?>
```

This style of assignment is inspired by the jQuery library and uses method chaining to make it easier to specify multiple optional arguments. Notice that the 'class' method corresponds to the `$class` input of the 'post/li' part. Now we're finally equipt to use our higher-order 'each-toggle' template! Let's take a look:

Example 10.14. Refactoring post/index.html.php using PartBlocks and a Higher-order Part

```
<?php
Layout::extend('master');
$title = 'A List of Posts';
?>
<ul>
<?php
    $li = Part::block('post/li');
    Part::draw('each-toggle', $posts, $li, $li->class('odd'));
?>
</ul>
```

Voila! Simple and beautiful. If we wanted to be really pedantic we could even make a part for a list of posts. In this case it's a little excessive, but let's do it just to demonstrate how parts can be composed. Let's start with the code we'd like to end up with in `post/index.html.php` template and design the part from it.

Example 10.15. Our final version of `post/index.html.php`

```
<?php
Layout::extend('master');
$title = 'A List of Posts';
$body = Part::block('post/ul', $posts);
?>
```

Example 10.16. A Part that uses Parts - `post/ul.part.php`

```
<?php Part::input($posts, 'array'); ?>
<ul>
<?php
    $li = Part::block('post/li');
    Part::draw('each-toggle', $posts, $li, $li->class('odd'));
?>
</ul>
```

This wraps up our leading tutorial of the new view system in Recess 0.2! Blocks, Parts, and Layouts are the keywords. *Blocks* are chunks of HTML that haven't been rendered yet and can be static with `HtmlBlock` or wrap around parts with `PartBlock`. Layouts are advanced includes that safely transfer variables from a child template to a parent layout. This is possible because parent layouts specify their inputs. Parts, kind of like partials, also specify their inputs. Parts and Layouts are considered to be *Assertive Templates* for this reason. Parts can become blocks, kind of like functions can become lambdas, to enable powerful higher-order part templating.

Fundamental Concepts

Registering & Selecting Views

The `!RespondsWith` annotation is used on a controller to register one or more Views that should be used for responding to a request. The controller does not ultimately decide which view to use, rather the `IPolicy` does. `!RespondsWith` is a simple way of hinting to an `IPolicy` which view should be used to deliver a response. Recess' default (and only) policy, aptly named `DefaultPolicy`, looks at the views hinted and finds the first which can handle the response with the desired mime-type/format. Let's look at an example:

Example 10.17. The `!RespondsWith` Annotation

```
<?php
/**
 * !RespondsWith Layouts, Json
 * !Prefix my/
 */
class MyController {
    /** !Route GET, hello/ */
    function hello() { return $this->ok('hello'); }
}
?>
```

Now suppose the following request is handled by in: `GET /my/hello.json`. This will get routed to the `hello` method which returns an `OkResponse` with `'hello'` as the suggested template name. `DefaultPolicy` will first ask `LayoutsView` if it can respond with the `OkResponse` object by looking to see if the file `/app/views/my/hello.json.php` exists, if so `LayoutsView` will be chosen to handle the response. If not, `DefaultPolicy` will then ask `JsonView` if it can respond and, because it can automatically respond to JSON request, it always will. If the annotation on `MyController` gave priority to `JsonView` over `LayoutsView`, with: `!RespondsWith(Json, Layouts)`, then all JSON requests would be handled by `JsonView` without ever asking `LayoutsView`.

View

A View has only two operations: 1) to decide whether it can or cannot render a `Response`, 2) to render a `Response`. These are represented in a simple interface with the abstract methods of `AbstractView`: `canRespondWith(Response $response)` and `respondWith(Response $response)`. Some Views, like `JsonView`, can automatically render a response so developers never touch presentation code. For other Views, like `LayoutsView`, their job is accomplished by dispatching to user-defined scripts/templates from the `/apps/[app]/views/` directory where the user has specified *templates*.

Templates

Templates are where developers spend most of their time designing presentation. Views dispatch control to templates and pass in variables from the response. By default, `Recess` ships with native PHP templates so templates are plain-old-php files. Templates could also be `Smarty` templates when paired with `SmartyView`. Templates are placed in the `/apps/[app]/views/` directory which is somewhat of a misnomer because they're *templates*, not *views*.

Assertive Templates

If you've programmed presentation logic in a PHP application you've likely dealt with include a good amount. When designing applications that have common UI it is a good practice to extract common parts of the UI to their own script files. The problem with simple includes is that they share scope. This makes includes difficult to reason about, shown in the following example, even if you know `yourguess.php` isn't going to halt, you don't know whether "Hello World?" will get printed, or some other string, or perhaps a notice error that `temp` is not defined:

Example 10.18. The Danger in Sharing include Scope

```
<?php
$title = "Hello World?";
include('print-heading.php');
echo $title;
?>
===
<?php // print-heading.php
echo "<h1>$title</h1>\n";
unset($title);
?>
===
Output:
<h1>Hello World</h1>
Notice: Undefined variable: title
```

Using includes can lead to spaghetti-code where you must understand the entirety of every include before you can understand a single script that uses includes. Included files which expect certain variables to be

available (perhaps `$title` is used to fill in the `<title>` tag) require the includee to wade through the script and determine which variables to have available in the context. For these reasons Recess, as of version 0.20 encourages not using includes in your templates and instead using Assertive Templates.

Assertive templates are a name given to a certain style of template: one that asserts the presence and type of every input variable at the top of the script. Here is an example:

Example 10.19. An Assertive Template

```
<?php // print-heading.part.php
Part::input($title, 'string');
Part::input($level, 'int');

//////// Presentation Logic //////////
echo "<h$level>$h1</h$level>\n";
unset($title);
?>
```

Layouts and Parts use Assertive Templates to address the problem to make presentation logic more predictable. Here is a similar snippet to the first example using the Part helper:

Example 10.20. Avoiding the Danger of Include

```
<?php
$title = "Hello world?";
Part::draw('print-heading', $title, 1);
echo $title;
?>

===
Output:
<h1>Hello world?</h1>
Hello world?
```

Note that even though `print-heading.part.php` unsets `$title`, it has no effect on our top-level template. Also note that, with the way drawing the Part does not require knowing the names of the variables within `print-heading.part.php`. We simply just have to pass the right number of variables in. If this feels more like a function than an include, you're right, it's a lot like a function with typed inputs.

We'll go into more depth in the section describing the class `AssertiveTemplate`, the common superclass of `Layouts` & `Parts`. Although there is additional effort in defining input, it simplifies your reasoning about view logic and makes working with templates (especially other people's!) much more predictable.

Helpers

Helpers are classes, typically abstract and static, that provide methods to simplify common tasks when authoring presentation code. There are helpers, for example, for composing URLs to controller methods, importing CSS from your app's `public/assets` directory, etc. For more info on the helpers that ship with Recess see the section called “View Helpers”

Views by Example with Layouts

Views

Views have two fundamental jobs:

1. Tell an `IPolicy` whether or not the view can respond with the `Response` object returned from a `Controller`.
2. Respond using the state in a `Response` object by sending headers (`AbstractView::sendHeadersFor()`) and rendering the response (`AbstractView::render()`).

AbstractView

`AbstractView` is the abstract base class of all views in `Recess`. Its two abstract methods are of interest to View developers:

1. `canRespondWith(Response $response)` - Returns a boolean indicating whether the `View` can respond with `$response`. It is the view's job to check the requested mime-type/format `$response->request->accepts->format()`. Responding true implies that the view can render the response in the format requested.
2. `render(Response $response)` - Using the data in `$response`, render a response in the desired format.

NativeView

`NativeView(!RespondsWith Native)` is a simple `View` implementation that takes a response and maps it to a template. This mapping is done based on the requested format, so if a controller returns `$this->ok('viewTemplate')` and the requested format is JSON, then `NativeView` will look for `'viewTemplate.json.php'`. `NativeView` loads no helpers and is as close to native PHP templates as you can get in `Recess`. `NativeView` does not load any helpers by default.

Format-based Template Selection

As already aluded to, `NativeView` will select a template based on the requested format. In `Recess`, a format can be requested in one of two ways: 1) It can be forced by appending the format to the end of a URL, i.e. `/foo/bar.json`, 2) It can be negotiated using the `Accept` HTTP header. In the first case `Recess` will only look for a `"*.json.php"` template.

LayoutsView

`LayoutsView(!RespondsWith Layouts)` extends from `NativeView` but auto-loads key helpers and initializes the `Layout` helper. This is the default `View` for `Recess` applications. The helpers available in templates loaded from `LayoutsView` are: `Html`, `Url`, `Layout`, `Buffer`, and `Part`. Your templates are rendered using the `Layout` helper's `draw` method.

JsonView

Implementing Custom Views

View Helpers

Loading with View's loadHelper

Html

- `specialchars($str, $double_encode = true)`
- `anchor($uri, $title = NULL, $attributes = NULL)`
- `css($style, $media = FALSE)`
- `link($href, $rel, $type, $suffix = FALSE, $media = FALSE)`
- `js($script)`
- `img($src = NULL, $alt = NULL)`
- `attributes($attrs)`

Special, global function: `h($var, $encode_entities=true)`

Url

- `action($actionControllerMethodPair)`
- `asset($file = "")`
- `base($suffix = "")`

Layout

- `extend($parent)`
- `draw($templateFile, $context)`

Part

- `draw($partTemplate, [$args*])`
- `block($partTemplate, [$args*])`
- `drawArray($partTemplate, $array)`

Buffer

- `to(&$block)`
- `appendTo(&$block)`
- `prependTo(&$block)`

- end()

Blocks

Abstract Block

HtmlBlock

PartBlock

Chapter 11. View Helpers

Chapter 12. Models

Generating Models with Recess Tools

The Recess Framework includes a web app to aid development called 'Recess Tools'. Generating new Models for an application and creating corresponding tables in the database is quick work. By browsing to your application and selecting 'new' Model you'll be taken to the new Model helper. After providing a name, table information, and properties the model and, if needed, table will be generated.

A quick peak at the code generated for `Post`:

Example 12.1. `Post` is simple Recess Model

```
<?php
/**
 * !Database Default
 * !Table posts
 */
class Post extends Model {
    /** !Column PrimaryKey, Integer, AutoIncrement */
    public $id;

    /** !Column String */
    public $title;

    /** !Column String */
    public $body;

    /** !Column Integer */
    public $authorId;
}??>
```

Lots of annotations! Why so many? Being explicit is a good thing- especially when you don't have to do any of the extra writing. Starting from the top: the Database annotation is what allows Recess to have Models from multiple data sources in a single app. The Table annotation is straightforward: the name of the table in the database the Model maps to. Following is a HasMany relationship using a join table with the Through argument. For Rails folks this should look decently familiar. More to come on relationships.

Each property uses a `!Column` annotation to provide additional semantic typing information. Specifying properties and column mappings is optional in Recess but it is encouraged for three reasons. One, you can look at a Model's code and know exactly which properties are available. This is different from Rails or Cake models. Two, Recess checks to ensure your annotations and the database types match. Three, Recess can regenerate tables from Models marked up with annotations.

Querying Models

Queries are constructed using a simple API. They're also lazy so queries are not executed until you need the results. The following are some example queries we could perform with the `Post` model:

Example 12.2. Querying the Post Model

```

<?php

$post = new Post();

$allPosts = $post->all(); // Select all Posts

$postsWithPhp = $allPosts->like('title', '%PHP%');

$lastFivePhpPosts = $postsWithPhp->orderBy('id DESC')->limit(5);
// Logically equivalent to:
$lastFivePhpPosts = $allPosts
    ->orderBy('id DESC')
    ->limit(5)
    ->like('title', '%PHP%');

$postWithAuthorId5 = $post->equal('authorId', 5);

```

The `all()` method is essentially a `SELECT *` on `Post`'s table. Notice how we can take the result of `all()` and continue refining our query with a `LIKE` clause on line 3. As mentioned above, Recess Models use lazy evaluation. The code above would not issue any SQL queries unless later in code the results were accessed in a `foreach` loop or with the array index syntax (i.e. `$postsWithAuthorId[0]`). Other simple query operators include: `notEqual`, `between`, `greaterThan`, `greaterThanOrEqualTo`, `lessThan`, `lessThanOrEqualTo`, `notLike`.

Persisting Model state with insert , update , save , delete

Persisting changes is as simple as calling `save()` for `INSERT`s or `UPDATE`s and `delete()` for `DELETE`s. Let's take a look:

Example 12.3. Persisting changes with save and delete

```

<?php
$post = new Post();
$post->title = 'Hello World';
$post->body = 'Welcome to Models in Recess!';
$post->save(); // internally calls $post->insert()
$post->title = 'Hello World! With a Bang!';
$post->save(); // internally calls $post->update()
echo 'New Post ID: ', $post->id;

$soldPost = new Post(10); // Post with ID 10
if($soldPost->exists()) {
    $soldPost->delete();
}

$postsWithRuby = new Post();
$postsWithRuby->like('title', '%Ruby%')->delete();
?>

```

In lines 1-7 a new `Post` is created saved and updated with some filler values. Lines 9-12 delete a `Post` with an ID of 10. Finally lines 14-15 delete all posts containing 'Ruby' in the title. Recess has support for cascading deletes across relationships which is discussed in the following chapter, `Model Relationships`.

Chapter 13. Relationships Between Models

A Preview

Lets say we need a controller method that is given a user `$id` and `$keyword`. The method will find the tags of all posts whose title contains `$keyword` that was written by the user with id of `$id`.

Example 13.1. Querying across relationships

```
<?php
/** !Route GET, user/$id/keyword/$keyword/tags */
function getTagsForUserByPostTitleKeyword($id, $keyword) {
    $this->user = new User($id);
    $this->tags = $user
        ->posts()
        ->like('title', "%$keyword%")
        ->tags();
}
```

In a view we can iterate through the tags with a simple foreach:

Example 13.2. Iterating through a query's results

```
<?php
foreach($tags as $tag) {
    echo $tag->name, '<br />';
}
```

Now, let's look at the code for the User, Post, and Tag models that make the above code snippet possible.

Example 13.3. Relationships between classes

```
<?php
/** !HasMany posts */
class User extends Model { }

/**
 * !BelongsTo user
 * !HasMany tags, Through: PostsTags
 */
class Post extends Model { }

/**
 * !HasMany posts, Through: PostsTags
 */
class Tag extends Model { }

/**
 * This model represents the join table between the many-to-many
 * Posts <-> Tags relationship.
 *
 * !BelongsTo post
 * !BelongsTo tag
 */
class PostsTags extends Model { }
```

Relationships

A Post belongs to a User and has many Tags. Lets take another look at how this is represented using annotations:

Example 13.4. !HasMany and !BelongsTo on a Post

```
<?php
/**
 * !BelongsTo user, Key: authorId
 * !HasMany tags, Through: PostTag
 */
class Post extends Model { /** Stripped for Brevity */ }
?>
```

The `!BelongsTo` annotation denotes the 'one' side of a one-to-many relationship. We specify some additional information using the `Key` modifier to say that the foreign key column name is actually `authorId` instead of `userId` which is what it would be by convention. With a belongs to relationship Post has an attached method of `user()` which will return the User model a Post is associated with. It also adds attached methods for setting and unsetting the user: `setUser($user)` `unsetUser()`. Attached methods are a low-level feature of Recess written in `Object` which allow methods to be added to classes dynamically at run time.

The `!HasMany` annotation is a special variant of `!HasMany` because it uses the `Through` modifier. This tells the `HasMany` relationship to use a join table thus making it a many-to-many relationship instead of a one-to-many. The `!HasMany` annotation attaches the following methods to the Post class: `tags()`, `addToTags($tag)`, and `removeFromTags($tag)`.

Naming Conventions

In Recess Models use a variation on the ActiveRecord pattern to provide an Object-Relational Mapping facility in the Recess Framework.

Model to Table

The assumed table name is the *Model's class name in lowercase* . To override this add a `!Table [table_name]` annotation to your model. When providing your own name for the table be sure to follow the case-sensitivity expectations of your RDBMS & OS MySQL [<http://dev.mysql.com/doc/refman/5.0/en/identifier-case-sensitivity.html>].

Example 13.5. Default naming conventions for Model Class -> Table

```
<?php
class Post extends Model {} // Table: post
class Person extends Model {} // Table: person
```

Example 13.6. Overriding the default naming convention of Model Class -> Table

```
<?php
/** !Table posts */
class Post extends Model {} // Table: posts

/** !Table people */
class Person extends Model {} // Table: people
```

!BelongsTo Relationship

Example 13.7. Example naming convention for !BelongsTo

```
<?php
/** !BelongsTo person */
class Post extends Model {}

class Person extends Model {}

$post->person(); // querying the relationship
```

Table 13.1. Variables used in following table

Variable	Value
BelongsToName	person

Table 13.2. Naming conventions for !BelongsTo

For	Convention	Example	Override
Related Class	ucfirst(BelongsToName)	Person	Class: ClassName
Foreign Key	BelongsToName . 'Id'	personId	Key: ColumnName

Example 13.8. Example overriding !BelongsTo conventions

```
<?php
/** !BelongsTo author, Class: Person, Key: personId */
class Post extends Model {}

class Person extends Model {}

// usage:
$post->author();
```

!HasMany Relationship

Example 13.9. Example naming convention for !HasMany

```
<?php
/** !BelongsTo author, Class: Person, Key: personId */
class Post extends Model {}

/** !HasMany post */
class Person extends Model {}

$person->post(); // usage
```

Table 13.3. Variables used in following table

Variable	Value
HasManyName	post
Class	Person

Table 13.4. Naming conventions for !HasMany

For	Convention	Example	Override
Related Class	ucfirst(HasManyName)	Post	Class: ClassName
Foreign Key	lcfirst(Class). 'Id'	personId	Key: ColumnName

Example 13.10. Example overriding naming convention for !HasMany

```
<?php
/** !BelongsTo author, Class: Person, Key: personId */
class Post extends Model {}

/** !HasMany posts, Class: Post, To: author */
class Person extends Model {}

// usage:
$author->posts();
```

!HasMany, Through Relationship

Example 13.11. Example naming conventions for !HasMany, Through

```
<?php
/**
 * !BelongsTo author, Class: Person, Key: personId
 * !HasMany tag, Through: TagsPosts
 */
class Post extends Model {}

/** !HasMany post, Through: TagsPosts */
class Tag extends Model {}

/** !BelongsTo post */
/** !BelongsTo tag */
class TagsPosts extends Model {}
// usage:
$post->tag();
```

Table 13.5. Variables used in following table

Variable	Value
HasManyName	tag
Class	Post
ThroughClass	TagsPosts

Note

The following "From" and "To" are a planned convention not yet supported.

Table 13.6. Naming conventions for !HasMany, Through

For	Convention	Example	Override
Related Class	ucfirst(HasManyName)	Tag	Class: ClassName
Through Class	ThroughClass	TagsPosts	Through: ThroughClass
Through's From Rltn	lcfirst(Class)	post	Through: ClassName, From: localRelation
Through's To Rltn	HasManyName	tag	Through: Classname, To: foreignRelation

Example 13.12. Example of overriding naming conventions for !HasMany, Through

```
<?php
/** !HasMany tags, Through: TagsPosts, From: postRltn, To: tagRltn */
class Post extends Model {}

/** !HasMany posts, Through: TagsPosts, From: tagRltn, To: postRltn */
class Tag extends Model {}

/** !BelongsTo postRltn, Class: Post, Key: post_id */
/** !BelongsTo tagRltn, Class: Tag, Key: tag_id */
class TagsPosts extends Model { }
// usage:
$post->tags();
```

Chapter 14. Plugins & Extensibility

Chapter 15. RESTful APIs in Recess

Part IV. Deploying Recess Applications

Chapter 16. Deploying Applications

Chapter 17. Production Mode

Part V. Contributing to Recess

Chapter 18. Forking on GitHub

Forking Recess Edge on GitHub is easy, and the complete instructions can be found over at GitHub: <http://github.com/guides/fork-a-project-and-submit-your-modifications>

Working with Recess as a Git Submodule

When developing with the edge version of Recess it can be a pain to maintain source control for your application. Luckily Git has a notion of submodules which allow other git repositories to be embedded within parent repositories. Follow these steps to setup Recess Edge as a submodule within your application's git repository:

1. cd into your project's directory
2. git submodule add git://github.com/recess/recess.git ./edge/
3. cp edge/.htaccess .
4. cp edge/recess-conf.php .
5. cp edge/bootstrap.php .
6. cp -R edge/apps apps/
7. cp -R edge/data/ data/
8. cp -R edge/plugins/ plugins/
9. (this is excessive - getting public assets from edge's tools dir into your app dir)
- 10.mkdir recess
- 11.mkdir recess/recess
- 12.mkdir recess/recess/apps
- 13.mkdir recess/recess/apps/tools
- 14.cp -R edge/recess/recess/apps/tools/public recess/recess/apps/tools/public
- 15.Modify recess-conf.php

```
// Paths to the recess, plugins, and apps directories
RecessConf::$recessDir = $_ENV['dir.bootstrap'] . 'edge/recess/';
```

Chapter 19. Setting up PHPUnit for Recess

The Recess unit tests are written for PHPUnit [<http://www.phpunit.de/>]

In order to run the unit tests for Recess take the following steps:

Figure 19.1. Setting up PHPUnit for Recess

1. Install PEAR - Typically this is done by executing the **go-pear** script in your PHP installation directory.
2. Install PHPUnit [<http://www.phpunit.de/manual/3.3/en/installation.html>]
 - a. First add the PHPUnit channel to PEAR: **pear channel-discover pear.phpunit.de**
 - b. Next install PHPUnit with **pear install phpunit/PHPUnit**
3. Setup MySQL & DSN for Database Tests
 - a. The default MySQL test DSN is `'localhost,dbname=recess, user=recess, password=recess'`
 - b. To use settings other than default modify `recess/test/recess/database/PdoDsnSettings.php`
4. At a command prompt, navigate to `recess/test/`
5. Run the Unit Tests: **phpunit --bootstrap bootstrap.php AllTests.php**

To contribute unit tests to Recess fork the source on GitHub, commit your unit tests to the fork, and request a pull from Recess. Happy testing!

Chapter 20. Useful Git Commands

Chapter 21. Submitting Bug Reports

Chapter 22. Contributing to Documentation

Index

Symbols

!BelongsTo, 57
!Column, 52
!HasMany, 58
!HasMany, Through, 59
!Route,
!RoutePrefix, 36

A

Application
 create manually, 10
 create with Recess Tools, 7
 directory structure,

C

Controller, 11
 Basics, 11
 Relationship with Views, 12
 urlTo method, 14

D

Diagnostics, 13

F

ForwardingResponse (see Response)

I

Install
 from GitHub, 3
 release, 3

M

Models,
 !Column (see !Column)
 delete(), 53
 Generating, 52
 insert(), 53
 Persisting, 53
 Querying, 52
 Relationships (see Relationships)
 save(), 53
 update(), 53

P

PEAR,
PHPUnit,

R

recess-conf.php

 RecessConf::\$applications, 8
Recess Tools, 7
 Routing, 37
Relationships,
 !BelongsTo (see !BelongsTo)
 !HasMany (see !HasMany)
 !HasMany, Through (see !HasMany, Through)
 Naming Conventions, 57
 Querying, 55
Request Object, 11
Response, 13
 ForwardingResponse, 14
RouteAnnotation, 34
Routing,
 Implicit, 36
 Multiple per method, 35
 Parameters, 34
 Parametric, 34
 Performance, 37
 Relative vs. Absolute, 36

U

Unit Testing
 Recess,
 urlTo (see Controller)

V

View
 Basics, 12