test Documentation

Release 0.1.0

foo

May 24, 2015

Contents

CHAPTER 1

MVC

1.1 Introduction

This document describes the Maintainable PHP Framework. It is a based around the Model-View-Controller pattern and is modeled after Ruby on Rails 1.2. It is compatible with PHP version 5.1.4 and later.

This software is offered under a BSD license.

1.1.1 Git Repository

The Maintainable PHP Framework is hosted at GitHub:

https://github.com/maintainable/framework

The Git repository is the only way to obtain this software.

1.1.2 Directory Structure

All applications built using the Maintainable Framework share the exact same directory structure. This keeps projects consistent, allows team members to easily transition between projects, and allows for tooling that runs under the assumption of this structure.

Most of the server-side application development will take place in the `app/` and `test/` directories. Client-side CSS, images, and JavaScript will be in the `public/` directory. Vendor libraries such as the Maintainable framework itself and its dependencies reside in `vendor/` to permit easy upgrading.

Application Code

The application code resides under `app/`:

It is split into three different layers. Each layer has a directory under `/app`: `models/`, `views/`, and `controllers`. We also have an additional `helpers/` directory under here for view helper methods.

- Models: `/app/models/Users.php`
- Controllers: `/app/controllers/UsersController.php`
- Views: `/app/views/users/show.html`
- Helpers: `/app/helpers/UsersHelper.php`

Web-accessible

Images, CSS, and JavaScript are all stored in `public/`:

Configuration

Environments

The three different runtime environments are:

- `development`
- `production`
- `test`

Every request will include the common `config/environment.php` file and then its respective `/config/environments/{environment}.php` file. These files include constants and configuration used throughout the application.

Normal MVC code that goes in `app/` will never need `require()` statements as this is done automatically. Vendor (library) code in the `vendor/` directory also does not need to be explicitly required because it will be autoloaded by the PEAR convention (which all vendor files must abide).

URL Routes

Request Routing is configured in `/config/routes.php`. This file defines what code gets run when a particular URL is requested. This is explained in more detail under *Request Routing*.

Vendor Libraries

All vendor libraries, including the Maintainable framework itself, are located under `vendor/`. The framework does not invent its own plugin system or other exotic loading techniques. Libraries must simply reside in this directory and abide by the PEAR naming conventions. The framework libraries are all under `vendor/Mad/` and hence the classes are prefixed `Mad_` by this convention.

Naming Conventions

The framework has important naming conventions that are crucial to how the code integrates together. By sticking to these conventions, it makes code more consistent and less work gluing the pieces together.

Models

Table	Class	File
users	User	models/User.php

Controllers

URL	Class	File	Method
/users/show	UsersController	controllers/UsersController.php	show()

Views

URL	HTML File	Ajax Response
/users/show	views/users/show.html	views/users/show.js

1.2 Models

1.2.1 Overview

The `Mad_Model` component is an object-relational mapping (ORM) layer for the framework. We follow the ActiveRecord pattern where tables map to classes, rows map to objects, and columns to object attributes. The implementation is close to that of Ruby on Rails.

The model layer is where the domain (business) logic of the application lies. This involves data retrieval, manipulation, and validation.

> *An informal test I like is to imagine adding a radically different layer to an application, such as a command-line interface to a Web application. If there's any functionality you have to duplicate to do this, that's a sign of where domain logic has leaked into the presentation.* – Martin Fowler

1.2.2 Generating Stubs

The framework provides a tool to generate stub files for new Mad_Model, and related classes. We can use `script/generate` script to do this. This script should be run from the application's root directory as such:

```
$> cd /app

$> php ./script/generate model User
```

This will create the following 4 file stubs which include the model class, the unit test stub file, the migration, and the yml fixture file:

- `/app/app/models/User.php`
- `/app/test/unit/UserTest.php`
- `/app/db/migrate/001_create_users.php`
- `/app/test/fixtures/users.yml`

We will talk more about the testing files in the *Unit Testing* chapter.

1.2.3 Tables and Classes

When you create a subclass of Mad_Model you are creating a wrapper for a database table. The mapping of the table class is determined by specific naming conventions. The table should be named as the plural and underscored version of the model class name.

Table Name	Model Name	Model File	Test File	Fixture File
users	User	models/User.php	test/units/UserTest.php	test/fixtures/users.yml

We can create an object to access data in this table by instantiating a new `User` object:

```
// wrap the 'users' table by creating a User object
$user = new User;
```

1.2.4 Columns and Attributes

Each Model instance corresponds to a row in the database table. The object's attributes will correspond directly to the database columns, and are dynamically added using introspection of the database structure:

```
// find a user record with the primary key of "1"
$user = User::find(1);

// get specific column info
$name  = $user->name;
$phone = $user->phone;

// set attributes and save back to db
$user->name  = 'Donny';
$user->phone = '555-1212';
$user->save();
```

1.2.5 Migrations

When we generated the User model, the generator also created our users migration file. All migrations are stored within the `db/migrate/` directory of our application, and keep a version history of database changes within the source tree. You'll see that the migration file for our `users` table has a numbered prefix of 001, which designates it as version 1 of our database history.

This gives us a powerful tool for applying and rolling back any changes we make to the database. This is especially useful for teams that need to keep in sync with each other's database changes. Migrations are written in PHP, which lets you easily make applications that are more platform and database independent.

In each migration file, we'll see the `up` and `down` methods. These will instruct our migration what to do when migrating `up` to revision number 1 of our database, or reverting back `down` to revision number 0:

```
class CreateUser extends Mad_Model_Migration_Base
{
    public function up()
    {
        $t = $this->createTable('users', $options = array());
            $t->column('username',   'string');
            $t->column('company_id', 'integer');
        $t->end();
    }

    public function down()
    {
        $this->dropTable('users');
    }
}
```

When we migrate up in this migration, we'll be creating the users table. When migrating down, well drop the users table again.

We can execute the migration with `script/task db:migrate`. Navigate to your application's root directory to run this:

```
$> php ./script/task db:migrate
== 001 CreateUsers: migrating ================================================
-- createTable(users)
   -> 1.1460s
== 001 CreateUsers: migrated (1.1460s) =======================================
```

Running this script will migrate to the newest version of your database schema, which in our case has successfully updated us to version 1. It will determine the newest version by scanning the filenames of the files in `db/migrate/` to find the highest sequentially numbered migration. To instruct the task to migrate to a specific version, we can add the `VERSION=` argument to the script:

```
$> php ./script/task db:migrate VERSION=0
== 001 CreateUsers: reverting ============================================
-- dropTable(users)
   -> 2.0070s
== 001 CreateUsers: reverted (2.0070s) ===================================
```

Here we have specified in the `migrate` command to revert back to `VERSION=0`. When executed, the migration drops the user table that we had specified in the `down` method of this migration. The framework keeps track of the migration version you are on by automatically creating a table named `schema_info` the first time you run a migration. This table use a single column named `version` to remember the version number:

```
mysql> use my_app_development;
Database changed
mysql> select * from schema_info;
+---------+
| version |
+---------+
|       0 |
+---------+
```

We can run migrations in production mode by adding the `MAD_ENV=production` to the list of arguments to `script/task db:migrate`.

Let's now take a look at all the different operations we can perform within a migration file.

Create a Table

Each `$t->column()` call within the `createTable('users')` block specifies a column for the table we are creating. The first argument is the column name, and the second is the data type. Since column type keywords vary across different database platforms, the framework uses a database independent syntax to specify the type of column we are creating. The valid types are `binary`, `boolean`, `date`, `datetime`, `decimal`, `float`, `integer`, `string`, `text`, `time`, `timestamp`.

The last argument to the column creation method is an associative array of options for the column. This is where you can specify if this column uses a null constraint, default value, or character limit. We've taken advantage of these options to limit our `password` column to 40 characters, and add a default value of 0 to the `is_admin` column:

```
$t = $this->createTable('user', $options = array());
    $t->column('username',  'string',  array('null' => false));
    $t->column('password',  'string',  array('limit' => 40));
    $t->column('company_id' 'integer');
    $t->column('is_admin',  'boolean', array('default' => '0'));
    $t->column('profile',   'text');

    // magic cols
    $t->column('created_at', 'datetime');
    $t->column('updated_at', 'datetime');
$t->end();
```

A primary key column named `id` will be automatically created for each table.

There are a couple reserved names for special columns used to store the date and time of when user record was created or updated. These columns are named `created_at` and `updated_at`. Mad_Model will automatically insert the

current time into these columns when we insert or update user records. We'll typically add these columns to all tables that have data being modified by the application.

An optional `$options` array can be given as the second argument to `createTable()`:

- `primaryKey`: create the primary key (id) for the table (defaults to true)
- `force`: drop any existing table by the same name (boolean)
- `temporary`: create a temporary table (boolean)
- `*`: other options can be added to append to the create statement

Rename a Table

Rename the table `users` to `clients`:

```
$this->renameTable('users', 'clients');
```

Drop a Table

Drop the `users` table:

```
$this->dropTable('users');
```

Add a Column

An a `fax_number` column to the `users` table:

```
$this->addColumn('users', 'fax_number', 'string', array('limit' => 10));
```

Remove a Column

Remove the `fax_number` column from the `users` table:

```
$this->removeColumn('users', 'fax_number');
```

Change Column Default

Change the default value of the `is_admin` column of the `users` table:

```
$this->changeColumnDefault('users', 'is_admin', '1');
```

Change a Column

Change the type and limit of the `phone` column of the `users` table:

```
$this->changeColumn('users', 'phone', 'integer', array('limit' => '10'));
```

Change a column's precision/scale:

```
$this->changeColumn('users', 'cash_on_hand', 'decimal',
                    array('precision' => '5', 'scale' => '2'));
```

Rename a Column

Rename the `phone` column to `phone_number`:

```
$this->renameColumn('users', 'phone', 'phone_number');
```

Add an Index

Add an index on a single column:

```
$this->addIndex('users', 'company_id');
```

Add an index on multiple columns:

```
$this->addIndex('users', array('name', 'company_id'));
```

Add a unique index:

```
$this->addIndex('users', 'email', array('unique' => true));
```

Specify the name of an index instead of using the framework's default:

```
$this->addIndex('users', 'is_admin', array('name' => 'admin'));
```

Remove an Index

Remove an index on a single column:

```
$this->removeIndex('users', array('column' => 'company_id'));
```

Remove an index on multiple columns:

```
$this->removeIndex('users', array('column' => array('name', 'company_id')));
```

Remove an index by its name:

```
$this->removeIndex('users', array('name' => 'admin'));
```

Executing SQL

Even though we have methods to cover most operations you'll need to perform on a table, you can always drop down to SQL to do what you need:

```
$this->execute("INSERT INTO users (id, name) VALUES (1, 'Fred')");
```

1.2.6 CRUD

Mad_Model makes it very to perform the four basic operations on database tables: Create, Read, Update, and Delete. The operations in this section work work with a `Folder` class to describe how to manipulate data in a table named `folders`.

Creating New Rows

Since tables are represented as classes, and each object represents a row in the database, it would make sense that we would create a new object to insert a new record. We have to make sure that we use `save()` to insert the record or it only exists in memory:

```
// insert folder by setting properties
$folder = new Folder;
$folder->name        = 'My New Folder';
$folder->description = 'Folder Description';
$folder->save();
```

Mad_Model objects also take an array as an optional constructor argument. This can be used as a shortcut for loading attributes for a new object:

```
// set the properties using an attribute array
$folder = new Folder(array('name'        => 'My New Folder',
                           'description' => 'Folder Description'));
$folder->save();
```

You'll notice we didn't pass in the primary key to this object before saving. This is because the primary key for this particular object is auto-incremented. We can get the id by referencing it after the object has been saved:

```
// save and get the newly inserted id
$folder->save();
$newFolderId = $folder->id;
```

Another way to insert records is using the convenience method `create()`, which allows us to insert data without instantiating the object first:

```
// create single records
$folder = Folder::create(array('name'        => 'My New Folder',
                               'description' => 'Folder Description'));
```

We can also create multiple objects by passing in an array:

```
$folders = Folder::create(array(
                           array('name'        => 'Folder 1',
                                 'description' => 'Folder Description 1'),
                           array('name'        => 'Folder 2',
                                 'description' => 'Folder Description 2')));
```

Find Existing Rows

The simplest way of specifying a row in the table is by using its primary key. Every model supports the `find()` method which is very versatile. Rows can be retrieved using a single primary key, or an array of primary keys:

```
// retrieve a single folder by primary key
$folder = Folder::find(123);

// retrieve a collection of folders by primary key
$folders = Folder::find(array(123, 456, 789));
```

If any of the IDs given do not exist, the find() will throw a Mad_Model_Exception_RecordNotFound. This is because Model assumes that when searching by primary keys, that the specific IDs given should be present in the database (otherwise, where would those IDs come from?).

More often than not you will need more power. The above example just scratches the surface of `find()`. Find has a completely different method of working when you pass it either `all` or `first` as the first argument::. The `first`

string when passed in will restrict the result set to a single record, and the `all` string will return an array of Folder objects that match the given conditions:

```
// retrieve the first Folder
$folder = Folder::find('first');

// retrieve all Folders
$folders = Folder::find('all');
```

Finder Options

The real power of `find()` comes in its second argument, which is an array of options that can be passed in to build the SQL statement. Let's start with the `conditions` option to see how Mad_Model works with SQL:

```
// find folders within the parent_id=181 with more than 10 documents
$folders = Folder::find('all', array('conditions' => 'parent_id = :parent_id AND
                                                      document_count > :count'),
                               array(':parent_id' => '181',
                                     ':count'     => '10'));

// loop through the collection
foreach ($folders as $folder) {
    print $folder->name;
}


// get a specific element in the collection<
$specificFolder = $folders[3];
```

Note: The third argument to `find()` is an array of bind variables. It is extremely important to **always bind your variables** to avoid SQL injection attacks.

The result will be a Mad_Model_Collection object which will be conveniently accessible with array-like syntax. This means you can do a `foreach()` over it or access specific elements. If we were to run the same find using `first` instead of `all`, the result would be a single Folder object.

One thing you'll notice about the example above is that we're not trying to avoid SQL. The `conditions` argument as well as many of the other options of `find()` are indeed just SQL. The aim is not to completely replace SQL with an object model but rather to embrace SQL while reducing the duplication involved in writing it.

The options available as the second argument to `find()` are as follows:

- `select`: retrieve specific columns
- `from`: specify FROM tables
- `conditions`: set SQL WHERE conditions
- `order`: set result ordering
- `group`: set result grouping
- `offset`: offset of the result set
- `limit`: limit of the result set
- `include`: eager load associated models

TODO finish section

1.2.7 Associations

The real fun in Mad_Model comes with the associations. Mad_Model allows you to tie model objects together through database foreign-key relationships.

Once we have the correct relationships declared in the `_initialize` method of the model, we can refer directly to related objects of that model. If we were to say that "Folder has many Documents", we could then reference the documents within a folder model through the relationship:

```
// print the name of each document within the folder.
$folder = Folder::find(123);
foreach ($folder->documents as $document) {
    print $document->name;
}
```

There are four different relationships that can be defined between models:

- `belongsTo`: specify a one-to-one association
- `hasOne`: specify a one-to-one association
- `hasMany`: specify a one-to-many association
- `hasAndBelongsToMany`: specify a many-to-many association

In all the relationship methods, the first argument is the name of the association to be added. By default, you will want to make this the Name of the associated class. For example, a Document "belongsTo" a Folder:

```
class Document extends Mad_Model_Base
{
    public function _initialize()
    {
        $this->belongsTo('Folder')
    }
}
```

The plurality of the class name changes with one-to-many and many-to-many relationships so that it reads in a more natural way. Notice how a Document belongsTo Folder, while a Folder hasMany Documents:

```
class Folder extends Mad_Model_Base
{
    public function _initialize()
    {
        $this->hasMany('Documents')
    }
}
```

While this makes our associations nice and easy to read, the name of the association is not tied down to the name of the model. This comes in handy if you need multiple relationships to the same model.

The second argument in all relationship definitions is an array of options to configure the relationship. If you create a custom name for an association (not based directly on the name of the associated model), you will have to specify which model class it refers to using the `className` option:

```
class Folder extends Mad_Model_Base
{
    public function _initialize()
    {
        $this->hasMany('Docs', array('className'  => 'Documents'));
    }
}
```

We can now refer to this association as `docs` instead of ``documents`:

```
$folder = Folder::find(123);
foreach ($folder->docs as $doc) {
    print $doc->name;
}
```

Each association has specific options, as well as specific properties/methods that are dynamically added when the association is declared.

Belongs-To

The `belongsTo()` method allows us to specify a `one-to-one` relationship with another model. This declaration must be made in the model that contains the foreign key.

Options:

- `className`: specify the model class of the associated object
- `foreignKey`: specify the foreign key column name used in the relationship
- `include`: eager loaded associations to include when this association is called

In this example, Folder belongsTo Document:

```
class Document extends Mad_Model_Base
{
    public function _initialize()
    {
        $this->belongsTo('Folder')
    }
}
```

We can now use the relationship referring to the associated object as `folder`:

```
$doc = Document::find(123);
print $doc->folder->name;
```

Properties/methods added with `belongsTo`:

- `{assocName}`: access associated object
- `{assocName} =`: assign associated object
- `build{AssocName}`: assign associated object by building a new one (associated object doesn't save)
- `create{AssocName}`: assign associated object by creating a new one (saves associated object)

Access the associated object:

```
$folder = $document->folder;
```

Assign the associated object and save it:

```
$document->folder = Folder::find(123);
$document->save();
```

Build a new object to use in the association and save it:

```
$folder = $document->buildFolder(array('name' => 'New Folder'));
$document->save();

// build new object to use as association & save new association.
```

```
// This option will automatically save the associated object, but !not!
// the actual association with the current object until you use save().
$folder = $document->createFolder(array('name' => 'New Folder'));
$document->save();
```

Has-One

The `hasOne()` method also allows us to specify a `one-to-one` relationship with another model. This declaration is made in the model that contains the primary key.

Options:

- `className`: specify the model class of the associated object
- `foreignKey`: specify the foreign key column name used in the relationship
- `include`: eager loaded associations to include when this association is called

In this example, User hasOne AvatarImage:

```
class User extends Mad_Model_Base
{
    public function _initialize()
    {
        $this->hasOne('AvatarImage')
    }
}
```

We can now use the relationship referring to the associated object as `avatarImage`:

```
$user = User::find(123);
print $user->avatarImage->name;
```

Properties/methods added with `hasOne`:

- `{assocName}`: access associated object
- `{assocName} =`: assign associated object
- `build{AssocName}`: assign associated object by building a new one (associated object doesn't save)
- `create{AssocName}`: assign associated object by creating a new one (saves associated object)

Access associated object:

```
$avatarImage = $user->avatarImage;
```

Assign associated object and save new association:

```
$user->avatarImage = new AvatarImage(array('name' => 'profile.gif'));
$user->save();
```

Build new object to use as association & save new object/association:

```
$user->buildAvatarImage(array('name' => 'profile.gif'));
$user->save();

// build new object to use as association & save new association.
// This option will automatically save the associated object, but !not!
// the actual association with the current object until you use save().
$user->createAvatarImage(array('name' => 'privileged.gif'));
$user->save();
```

Has-Many

The `hasMany()` method allows us to specify a `one-to-many` relationship with another model. This declaration is made in the model that contains the primary key.

Options:

- `className`: specify the model class of the associated object
- `foreignKey`: specify the foreign key column name used in the relationship
- `conditions`: conditions that the association must meet (WHERE conditions). These must be prefixed with table name.
- `order`: ordering of the results to bring back (ORDER BY statement). These must be prefixed with table name.

In this example, Folder hasMany Documents:

```
class Folder extends Mad_Model_Base
{
    public function _initialize()
    {
        $this->hasMany('Documents')
    }
}
```

We can now use the relationship referring to the associated objects as documents:

```
// use the relationship
$folder = Folder::find(123);
foreach ($folder->documents as $document) {
    print $document->name;
}
```

Properties/methods added with `hasMany`:

- `{assocName}s`: access collection of associated objects
- `{assocName}s =`: assign collection of associated objects
- `{assocName}Ids`: access array of associated object's primary keys
- `{assocName}Ids =`: assign array of associated primary keys
- `{assocName}Count`: count associated objects
- `add{AssocName}`: add an object to the associated objects
- `replace{AssocName}s`: replace associated objects with new assignment of objects
- `delete{AssocName}s`: delete specific associated objects
- `clear{AssocName}s`: clear all associated objects
- `find{AssocName}s`: find subset of associated objects
- `build{AssocName}`: add associated object by building a new one (associated object doesn't save)
- `create{AssocName}`: add associated object by creating a new one (saves associated object)

Access collection of associated objects:

```
$documents = $folder->documents;
```

Assign array of associated objects and save associations:

```
$folder->documents = array(Document::find(123), Document::find(234));
$folder->save();
```

Access array of associated object's primary keys:

```
$documentIds = $folder->documentIds;
```

Set associated objects by primary keys:

```
$folder->documentIds = array(123, 234);
$folder->save();
```

Get the count of associated objects:

```
$docCount = $folder->documentCount;
```

Add an associated object to the collection and save it:

```
$folder->addDocument(Document::find(123));
$folder->save();
```

Replace the associated collection with the given list. Will only perform update/inserts when necessary:

```
$folder->replaceDocuments(array(Document::find(123), Document::find(234)));
$folder->replaceDocuments(array(123, 234));
$folder->save();
```

Delete specific associated objects from the collection:

```
$folder->deleteDocuments(array(Document::find(123), Document::find(234)));
$folder->deleteDocuments(array(123, 234));
$folder->save();
```

Clear all associated objects:

```
$folder->clearDocuments();
$folder->save();
```

Search for a subset of documents within the associated collection:

```
$docs = $folder->findDocuments('all', array('conditions' => 'document_type_id = :type'),
                                      array(':type' => 1));
```

Build new object to add to association collection & save new object/association:

```
$document = $folder->buildDocument(array('name' => 'New Document'));
$document->save();

// build new object to add to association collection & save new association.
// This option will automatically save the associated object, but !not!
// the actual association with the current object until you use save().
$document = $folder->createDocument(array('name' => 'New Document'));
$document->save();
```

Has-Many-Through

The `hasMany(Objects, array('through' => 'JoinTable'))` method uses the `hasMany()` method with an additional `through` option to create a `many-to-many` relationship with an join model. This is the preferred approach to creating many-to-many relationship, and should be used instead of the *Has-And-Belongs-To-Many* association whenever possible. This declaration is made in both models in the relationship.

Options are the same as `hasMany` but add:

- `through`: The join model used in the association

The join table in this type of association is a model in itself, and should have a primary key. We make association declarations in all three models involved. The join model should have `belongsTo` declarations that refer back to the base models:

```
class Tag extends Mad_Model_Base
{
    public function _initialize()
    {
        $this->hasMany('Documents', array('through' => 'Taggings'));
    }
}

class Document extends Mad_Model_Base
{
    public function _initialize()
    {
        $this->hasMany('Tags', array('through' => 'Taggings'));
    }
}

class Tagging extends Mad_Model_Base
{
    public function _initialize()
    {
        $this->belongsTo('Tag');
        $this->belongsTo('Document');
    }
}
```

This sets up the association in both directions. We can now use the relationship referring to Tag's associated objects as `documents`, and Document's associated objects as or `tags`:

```
$tag = Tag::find(123);
foreach ($tag->documents as $doc) {
  print $doc->name;
}

$doc = Document::find(123);
foreach ($doc->tags as $tag) {
  print $tag->name;
}
```

The properties added with this association are the same as those added with a normal `hasMany` association.

Has-And-Belongs-To-Many

The `hasAndBelongsToMany()` method allows us to specify a `many-to-many` relationship with another model using a join table. This declaration is made in both models in the relationship.

Options:

- `className`: specify the model class of the associated object
- `foreignKey`: specify the foreign key column name used in the relationship
- `joinTable`: specify a join table to create the association

- `associationForeignKey`: specify a foreign key column for the join table used in the relationship
- `conditions`: conditions that the association must meet (WHERE conditions). These must be prefixed with table name.
- `order`: ordering of the results to bring back (ORDER BY statement). These must be prefixed with table name.

Example:

```
class Tag extends Mad_Model_Base
{
    public function _initialize()
    {
        $this->hasAndBelongsToMany('Documents')
    }
}

class Document extends Mad_Model_Base
{
    public function _initialize()
    {
        $this->hasAndBelongsToMany('Tags')
    }
}
```

This sets up the association in both directions. We can now use the relationship referring to Tag's associated objects as `documents`, and Document's associated objects as or `tags`:

```
$tag = Tag::find(123);
foreach ($tag->documents as $doc) {
    print $doc->name;
}

$doc = Document::find(123);
foreach ($doc->tags as $tag) {
    print $tag->name;
}
```

Properties/methods added with `hasAndBelongsToMany`:

```
- ``{assocName}s``: access array of associated objects
- ``{assocName}s =``: assign array of associated objects
- ``{assocName}Ids``: access array of associated object's primary keys
- ``{assocName}Ids =``: assign array of associated primary keys
- ``{assocName}Count``: count associated objects
- ``add{assocName}``: add an object to the associated objects
- ``replace{AssocName}s``: replace associated objects with new assignment of objects
- ``delete{AssocName}s``: delete specific associated objects
- ``clear{AssocName}s``: clear all associated objects
- ``find{AssocName}s``: find subset of associated objects
```

Access collection of associated objects:

```
$documents = $tag->documents;
```

Assign array of associated objects and save associations:

```
$tag->documents = array(Document::find(123), Document::find(234));
$tag->save();
```

Access array of associated object's primary keys:

```
$documentIds = $tag->documentIds;
```

Set associated objects by primary keys:

```
$tag->documentIds = array(123, 234);
$tag->save();
```

Get the count of associated objects:

```
$docCount = $tag->documentCount;
```

Add an associated object to the collection and save association:

```
$tag->addDocument(Document::find(123));
$tag->save();
```

Replace the associated collection with the given list:

```
// only performs update/inserts when necessary
$tag->replaceDocuments(array(Document::find(123), Document::find(234)));
$tag->replaceDocuments(array(123, 234));
$tag->save();
```

Delete specific associated objects from the collection:

```
$tag->deleteDocuments(array(Document::find(123), Document::find(234)));
$tag->deleteDocuments(array(123, 234));
$tag->save();
```

Clear all associated objects:

```
$tag->clearDocuments();
$tag->save();
```

Search for a subset of documents within the associated collection:

```
$docs = $tag->findDocuments('all', array('conditions' => 'document_type_id = :type'),
                                   array(':type' => 1));
```

1.2.8 Validations

When you are using the Model to insert or modify data in the database, most of the time you will need to validate data. The framework has a standard way to do this so that you can easily check the data given by a user and return a user-friendly message of any changes that need to be made to save the data.

Validation are added to a model using validation in the `_initialize()` method. There are six types of validations supported:

- `validatesFormatOf`: validate format of attribute values
- `validatesInclusionOf`: validate that the value falls within a list of acceptable values
- `validatesLengthOf`: validate length of attribute values
- `validatesNumericalityOf`: validate that attribute values are numeric
- `validatesPresenceOf`: validate existence of value for attribute values
- `validatesUniquenessOf`: validate uniqueness of attribute value

Validation Types

Format

`validatesFormatOf` validates that the value is alpha, digit, alnum, or that the value matches a given regexp pattern:

```
protected function _initialize()
{
    $this->validatesFormatOf('date_value', array('with' => '/\d{4}-\d{2}-\d{2}/'),
                             'message' => 'has to be formatted (YYYY-MM-DD)');

    $this->validatesFormatOf('number_value', array('on'   => 'update',
                                                   'with' => '[digit]'));
}
```

Options:

- `on`: validate on either save/insert/update (defaults to `save`)
- `with`: The ctype/regex to validate against - `[alpha]`, `[digit]`, `[alnum]`, or `/regex/`
- `message`: Custom error message (default is: `is invalid`)

Inclusion

`validatesInclusionOf` validates that the value falls within an array of acceptable values:

```
protected function _initialize()
{
    $this->validatesInclusionOf('answer', array('in' => array('yes', 'no')));
}
```

Options:

- `in`: validate that the submitted value falls within this array of values
- `on`: validate on either save/insert/update (defaults to `save`)
- `allowNull`: Consider null values valid (defaults to `false`)
- `strict`: Enforce identity when comparing values
- `message`: Custom error message (default is: `is not included in the list`)

Length

`validatesLengthOf` validates that the string length of the value is above, below, exactly matches, or within a range of sizes:

```
protected function _initialize()
{
    $this->validatesLengthOf(array('name', 'description'),
                             array('maximum' => 3000));

    $this->validatesLengthOf('description', array('minimum'  => 10,
                             'tooShort' => 'must have a better description'));
}
```

Options:

- `on`: Validate on either save/insert/update (defaults to `save`)
- `minimum`: Value may not be less than this int
- `maximum`: Value may not be greater than this int
- `is`: Value must be specific length
- `within`: The length of value must be in range: eg. `array(3, 5)`
- `allowNull`: Consider null values valid (defaults to `false`)
- `tooLong`: Message when `maximum` is violated (default: `is too long (maximum is %d characters)`)
- `tooShort`: Message when `minimum` is violated (default: `is too short (minimum is %d characters)`)
- `wrongLength`: Message when `is` is invalid. (default: `is the wrong length (should be %d characters)`)

Numeric

`validatesNumericalityOf` validates that the value is numeric, and can optionally allow decimals in the number:

```
protected function _initialize()
{
    $this->validatesNumericalityOf('number_value');

    $this->validatesNumericalityOf('age', array('allowNull' => true));
}
```

Options:

- `on`: validate on either save/insert/update (defaults to `save`)
- `onlyInteger`: Don't allow floats. (defaults to `true`)
- `allowNull`: Consider null values valid (defaults to `false`)
- `message`: Custom error message (default: `is not a number`)

Presence

`validatesPresenceOf` validates that a value is not blank (`null` or an empty string):

```
protected function _initialize()
{
    $this->validatesPresenceOf('name');
}
```

Options:

- `on`: validate on either save/insert/update (defaults to `save`)
- `message`: Custom error message (default: `can't be blank`)

Uniqueness

`validatesUniquenessOf` validates that the value doesn't already exist in the database. It can also scope this uniqueness only validate when in combination with another column's value:

```
protected function _initialize()
{
    $this->validatesUniquenessOf('name', array('scope' => 'parent_id'));
}
```

Options:

- `on`: validate on either save/insert/update (defaults to `save`)
- `scope`: An attribute by which to limit the scope of the uniqueness
- `message`: Custom error message (default is: `has already been taken`)

Note: This validation works by performing a SQL `SELECT` to check for the value before saving with `INSERT` or `UPDATE`. Since many instances of your application may be accessing the database concurrently, a race condition exists where another instance may insert a duplicate value. If no duplicate values can be tolerated, a unique index must also be created in the database.

Validation Methods

There are three different methods for validating data:

- `validate`: executed before all updates/inserts
- `validateOnCreate`: executed before all inserts
- `validateOnUpdate`: executed before all updates

When you add one or more of the above methods to your model, it will automatically be registered to execute before data is saved. Adding errors from within these methods is done via the `errors->add()` or `errors->addToBase()` methods:

```
Class Folder extends Mad_Model_Base
{
    /**
     * This method will execute before any update/insert operation
     * it makes sure that the description is not empty, and that the name
     * isn't changed.
     */
    public function validate()
    {
        // arguments of add() are the attribute name and message
        if (empty($this->description)) {
            $this->errors->add("description", "cannot be blank");
        }

        // we can also add errors not associated with a attributes
        if (empty($this->name)) {
            $this->errors->addToBase('Fix the name!');
        }
    }
}
```

Validation Errors

When a validation error is encountered during a save operation, a list of errors is added to the model in the object's `errors` property. The `save()` method will return false when errors are encountered. The `errors` property on the object is actually an instance of `Mad_Model_Errors`, which is an iterable list of errors. To get an array with the full error messages encountered, we will use the `$folder->errors->fullMessages()` method:

```
$folder = Folder::find(123);
$folder->description = '';
if (!$folder->save()) {
    $errors = $folder->errors->fullMessages();
    foreach ($errors as $error) {
      print "$error\n";
    }
}
```

Alternately, we can use exception handling to catch validation errors. This only works when we use the `saveEx()` method to save our object. It is preferred to not use exception handling when accessing errors. The `errors` attribute mentioned above is more useful when we are using form helpers to do the work of displaying errors:

```
try {
    $folder = Folder::find(123);
    $folder->description = '';
    $folder->saveEx();

} catch (Mad_Model_Exception_Validation $e) {
    foreach ($e->getMessages() as $message) {
        print $message;
    }
}
```

1.2.9 Callbacks

Mad_Model has ways of monitoring and intercepting the execution inserts, updates, and deletes via the standard Model methods. We can write code that gets invoked at any significant event in the life cycle of a model object:

- `beforeValidation`: executed before validation
- `afterValidation`: executed after validation
- `beforeSave`: executed before inserts/updates
- `afterSave`: executed after inserts/updates
- `beforeCreate`: executed before inserts
- `afterCreate`: executed after inserts
- `beforeUpdate`: executed before updates
- `afterUpdate`: executed after updates
- `beforeDestroy`: executed before deletes
- `afterDestroy`: executed after deletes

A common use of callbacks is to perform pre-validation formatting of data:

```
public function User extends Mad_Model_Base
{
    public function beforeValidation()
```

```
    {
        if (!strstr($this->url, '://')) {
            $this->url = "http://" . $this->url;
        }
    }
}
```

1.3 Controllers

1.3.1 Overview

Controllers are the glue to our application. The controller system:

- Accepts the incoming HTTP request made by the browser
- Processes the URL using routing to determine which controller/action to use
- Dispatches the request to the correct controller/action
- Sends an HTTP response back to the browser with the result

Instead of each page of the application being a separate PHP file in the document root, we send all requests to a single PHP file. This file handles instantiating a controller to process the request based on the URL string.

This is a flexible way of handling requests because the URL is no longer tied directly to a PHP file, and can be changed at any time to suit our needs.

1.3.2 Request Routing

The first step to understanding controllers is learning how the framework handles a URL such as `example.com/customers/show/123` to determine which controller and action will process the request.

The process of mapping URLs to controllers and actions is performed internally by Horde_Routes. This integration is transparent to you. When you get into advanced uses, you may find the Horde_Routes documentation helpful for more information about routing.

Configuration

If you open up the routes configuration file `config/routes.php`, you will see a list of declarations such as:

```
$routes->connect(':controller/:action/:id');
```

Each one of these declarations specifies a route connecting a URL to a controller and action. The string given acts as a pattern to match against incoming URLs. The above route would match any incoming URL with 3 parts. The URL `customers/show/123` matches the route example:

When the route matches, it gives us the following parameters:

```
$params = array('controller' => 'customers',
                'action'     => 'show',
                'id'         => '1');
```

Using this information, the framework will execute the `show()` action in `CustomersController`. It will also make the `id` parameter available with the value of `1`.

Since our URL convention uses underscores, the application will automatically convert URL-style controller/action strings to their PHP counterparts. It will translate a URL such as `/fax_jobs/start_pending`:

- `fax_jobs` => `FaxJobsController`
- `start_pending` => `startPending()`

The application will try to match each route in the order they are defined in the `config/routes.php` file, from the top to the bottom. It will stop once it finds a matching route, and will throw a `Mad_Controller_Exception` if the URL does not match any route.

Route Components

The patterns accepted in the route string are composed of route components. Each component is separated by a forward-slash (`/`), and is matched to one or more URL components. The components can be 1 of 3 variations:

- `:name`
- `*name`
- `name`

`:name` sets the parameter `name` to the corresponding value found in the URL:

```
$this->connect(":name1/:name2");

// Match URL: "/foo/bar"
$params = array('name1' => 'foo',
                'name2' => 'bar');
```

`*name` will match all remaining components in the URL. It will set the parameter `name` to the string that makes up the remaining components. Because this pattern consumes the remainder of the URL, it must appear at the end of the pattern:

```
$this->connect(":name1/*name2");

// Match URL: "/foo/bar/baz"
$params = array('name1' => 'foo',
                'name2' => 'bar/baz');
```

`name` will match the route exactly to the matching text in the URL. The pattern `explore/:action/:id`, would only match a URL that starts with the string `explore`:

```
$this->connect("foo/:name1");

// Does NOT match: /bar/baz
// Does match: /foo/bar
$params = array('name1' => 'bar');
```

Route Options

The second argument to `$route->connect()` is an array of options. The options will typically set either:

- The default value of a component
- A requirement the component must pass

The following options will ensure that `id` is a digit (numeric) in order for this route to match. It will also set the parameter value of `id = null` if no `id` is given in the URL:

```
$this->connect(':controller/:action/:id', array(
                            'defaults'     => array('id' => null),
                            'requirements' => array('id' => '[0-9]+')));
```

More advanced options are also available, please consult the Horde_Routes documentation for information on these.

Route Defaults

You can give any component a default value that will get assigned to it if that component is empty in the URL. This can be done in one of two ways:

Set default action as `index`, the default id as `null`:

```
$this->connect(':controller/:action/:id', array('action' => 'index',
                                                'id'     => null));
```

Same defaults as above, using `defaults` array:

```
$this->connect(':controller/:action/:id', array(
                                'defaults' => array('action' => 'index',
                                                    'id'     => null)));
```

Both of these would match the URL `/explore` because they specify a default value for the empty action and id:

```
$params = array('controller' => 'explore',
                'action'     => 'index',
                'id'         => null);
```

In the above examples, both of these routes are identical in how they behave. The first example shows the shorthand of adding a default value. Which method you choose to use depends on what is required of the route, and what is the most readable format to maintain.

Route Requirements

If you give `requirement` for a route component, that component in the URL being matched must satisfy the requirement for the route to match. A requirement is a Perl-compatible regular expression without the surrounding delimiters:

```
/*
 * Does NOT match:
 *   "/customers/destroy/123" - action must be either index/search
 *   "/customers/show/abc"    - id must be numeric
 *
 * Does match:
 *   "/explore/search/123"
 */
$this->connect(':controller/:action/:id', array(
                     'requirements' => array('action' => 'index|show',
                                             'id'     => '[0-9]+')));
```

Notice above that the requirements do not have the delimiters that would be used with a PHP function like `preg_match()`. For example, where `preg_match` would use `/[0-9]+/`, the equivalent requirement is simply `[0-9]+`.

Common Pitfalls

One thing to always remember when writing a new route is that every route must set a 'controller' param, and an 'action' param. Without these parameters, the application has no clue where to send the request, and will fail every time.

The route is not required to have the controller/action parameters as actual components:

```
// this is perfectly acceptable. We set controller/action as defaults
$this->connect('search', array('controller' => 'search',
                               'action'     => 'display'));
```

Default Route

While we can add all the custom route configuation we need, most of the time we have no need to. We have a simple default route that covers 90% of the URLs we will need to use:

```
// the implicit default
$this->connect(':controller/:action/:id', array('id' => '[0-9]+'));
```

There is one thing special about the `action` and `id` params used in any route. They come will automatic defaults of `'action' => 'index', 'id' => null`.

1.3.3 Generating Stubs

Now that we know how URLs are mapped to controllers, we can start creating our own controller classes.

The framework provides a tool to generate stub files for new controller, and related classes. This is done using the `script/generate` script. This script should be run from the root directory where the project is located, which is also known as the PHP constant `MAD_ROOT`:

```
$> cd project_name

$> php ./script/generate controller SearchController index search
```

Note: Use `./script/generate` with no arguments for help. The example above generates a new controller called `SearchesController` with two actions: `index` and `search`.

This will create the following file stubs which include the controller class, template files for the actions, the functional test stub file, and a helper class file:

- `/app/views/Search/`
- `/app/views/Search/index.html`
- `/app/views/Search/search.html`
- `/app/controllers/SearchController.php`
- `/app/helpers/SearchHelper.php`
- `/tests/functional/SearchControllerTest.php`

1.3.4 Action Methods

When a request is being processed by a controller, it will look for a public method with the name of the action specified through the routes. This means that any public method in your controller can be executed as an action.

If you have methods in your controller that are not actions, you should make them `protected` or `private`. The controller will not treat these methods as actions:

```
class DocumentsController extends ApplicationController
{
    /**
     * This method CAN be executed as an action
     */
    public function show()
    {
    }

    /**
     * This method CANNOT be executed as an action
     */
    protected function _findDocument()
    {
    }
}
```

Note: PHP has a limitation where occasionally a name you would like for an action method will conflict with a PHP construct. For example, PHP will not allow a method named `new()`, but this is would be useful as an action name. When these conflicts arise, you my append `Action()` to the name – `new()` would become `newAction()`. Only do this when necessary.

1.3.5 Sending Responses

When an action is invoked in response to a request, the action needs to generate a response back to the user. The most common ways of doing this are to:

- Render a template/view
- Render text
- Send a File/Data
- Redirect the user

Rendering Templates

The default operation an action will perform (if not told otherwise) is to render a template. The controller will look for a template in the views directory that has the same name as the action:

```
class DocumentsController extends ApplicationController
{
    public function show()
    {
    }
}
```

For flexibility, we can also specify any template we want it to render using `render()`:

```
class DocumentsController extends ApplicationController
{
    public function update()
    {
        $this->render(array('template' => 'show'));
```

```
    }
}
```

Rendering Text

Action methods can render text directly without using a template by using the `renderText()` method. This is mostly used when sending Javascript data back during AJAX requests:

```
class DocumentsController extends ApplicationController
{
    public function remoteUpdate()
    {
        $this->render(array('text' => 'var saved = true;'));
    }
}
```

We can also tell an action to not render any data at all using `render()`:

```
class DocumentsController extends ApplicationController
{
    public function doNothing()
    {
        $this->render(array('nothing' => true));
    }
}
```

Sending Files/Data

When you want to send from the filesystem or text as a binary file, you can use the `sendFile()` and `sendText()` methods. These methods will allow you to force a dialog box on the user to download the resource:

- `sendFile()` sends the contents of a file to the user
- `sendData()` sends a string containing binary data to the client. Typically the browser will use a combination of content-type and disposition, both set in th options, to determine what to do with this data.

Both `sendFile` and `sendData` take an array of options as a second argument:

- `filename`: A suggestion to the browser of default filename to use when saving.
- `type`: the content type, defaulting to 'application/octet-stream'
- `disposition`: Suggest to the browser that the file should be displayed inline (option `inline`) or downloaded and saved (option `attachment`, the default)

Send a JPEG and display it inline:

```
class DownloadsController extends ApplicationController
{
    public function sendJPG()
    {
        $this->sendFile('/path/to/filename.jpg', array(
                                    'type'        => 'image/jpeg',
                                    'disposition' => 'inline'));
    }
}
```

Send a string containing CSV text as an attachment:

```
class DownloadsController extends ApplicationController
{
    public function sendCSV()
    {
        $csvText = $this->_getCsvText();
        $this->sendData($csvText, array('filename'    => 'ChannelReport.csv',
                                        'type'        => 'application/ms-excel',
                                        'disposition' => 'attachment'));
    }
}
```

Redirects

An action always performs one of two tasks: it either renders or it redirects.

The `redirectTo()` method is used to perform all redirects. A redirect will typically be another action name but can also be a URL:

```
class CustomersController extends ApplicationController
{
    public function edit()
    {
        // save data here

        $this->redirectTo(array('action' => 'index'));
    }
}
```

In the example above, the `edit()` action saves the data and then redirects back to the `index()` action. Since both of these actions are in the same controller, the controller name is implied.

To redirect to an action in another controller, set the controller name in the array like `array('controller' => 'documents', 'action' => 'index')`.

To redirect to another URL, use a string instead of an array. This can be an absolute URL such as `http://example.com` or a relative one such as `/foo/bar`. Only use these when necessary. The best practice is to specify redirects in terms of controllers and actions, not as URL strings.

1.3.6 Controller Environment

Initialize

After each controller is instantiated, it will execute the code in the `_initialize()` method. This allows us to perform code in a single place for all actions on a given controller:

```
class ExploreController extends ApplicationController
{
    protected $_foo;

    /**
     * Run this code before all action methods
     */
    protected function _initialize()
    {
        $this->_foo = 'bar';
```

```
    }
}
```

Request

All the data that was sent in the HTTP request made by the browser to our application is stored in our HttpRequest object. This object is available to the controller using the `$this->_request` property. Here are some of the more commonly used methods of the request object:

- `getUri`: Get the requested URI
- `getMethod`: Get the request method
- `getRemoteIp`: Get the IP address of the user
- `isAjax`: Check if this is an Ajax request?
- `getServer`: Get a `$_SERVER` variable
- `getEnv`: Get an `$_ENV` variable

Get the user's ip address from the request:

```
$ipAddress = $this->_request->getRemoteIp();
```

GET/POST/Params/Cookie/Session information is also available through the request, but our convention for accessing these is through the methods explained in *Accessing Data*.

Response

The goal of a controller is to generate a response to send back to the browser. Most of the time this is done behind the scenes using `render()`, `redirectTo()`, and `sendFile()` methods.

The response object is available for modification directly in the controller using the `$this->_response` property:

```
$this->_response->setHeader('X-Robots-Tag: noindex, nofollow');
```

1.3.7 Accessing Data

Route Params

Inside controllers, we can access the pieces of URLs that were configured in *Request Routing*:

```
$this->connect(":controller/:action/:id");
```

When we match against this route against URL such as `/documents/show/123`, The data in the `id` portion of the url (`123`) will be accessible through the controller action using the `params` object, which behaves similar to an array:

```
class DocumentsController extends ApplicationController
{
    public function show()
    {
        $id = $this->params['id'];

        $id = $this->params->get('id', 0); // default to 0
    }
}
```

When accessing params inside controllers, the framework exposes objects behave similarly to an array but overcome some of their headaches. The most useful aspect of this is that when a key does not exist, `$this->params['id']` will simply return NULL and no PHP notice will be raised.

When NULL is not the best default value, you can give any default value by using the `get()` method on this object such as `$this->params->get('id', 1)`. If the 'id' key is not present or the value at 'id' is NULL, it will be defaulted to 1.

Take a moment to understand the pattern above because you will use it frequently. The `params`, `cookie`, `flash`, and `session` objects all work this way.

GET and POST

Data that you would normally access from `$_GET` and `$_POST` is made available through the `params` object.

Similar to `$_GET['sort_by']`:

```
$id = $this->params['sort_by'];
```

Similar to `$_POST['sort_by']`:

```
$id = $this->params['sort_by'];
```

Similar to `$_POST['sort_by']`, default to `asc` if not set:

```
$id = $this->params->get('sort_by', 'asc');
```

You will access route params as well as GET and POST params all from the `params` object. You will always get the data this way – you will never use these superglobals. To remove the temptation, the superglobals are actually erased.

Cookies

Cookie data that you would normally access through the `$_COOKIE` superglobal is made available by the `cookie` object. Be very careful that any data stored this way is not sensitive data, and remember that it is not trusted input when read back into the application.

Remember our folder id:

```
$this->cookie['folder_id'] = 123;
```

Retrieve the folder id, or null if none:

```
$folderId = $this->cookie['folder_id'];
```

Retrieve the folder id, or 123 if none:

```
$folderId = $this->cookie->get('folder_id', 123);
```

Session

Session data that you would normally access through the `$_SESSION` superglobal is made available by the `session` object. You can get and set values in the session.

Remember our folder id:

```
$this->session['folder_id'] = 123;
```

Retrieve the folder id, or null if none:

```
$folderId = $this->session['folder_id'];
```

Retrieve the folder id, or 123 if none:

```
$folderId = $this->session->get('folder_id', 123);
```

Flash

Flash allows us a way to communicate between different actions in a controller across HTTP requests. It is a special value in the session that is available for the next request only.

This is most useful in situations where the user has performed a POST request to modify some data, and you want to display a message to the user about the errors/success of the operation on the next request. For example:

User sends request to save changes, setting flash on success:

```
public function save()
{
    try {
        Document::updateAttribute('name', $this->params['name']);
        $this->flash['sucess'] = "Saved changes successfully.";

        $this->redirectTo(array('action' => 'show'));
        return;

    } catch (Mad_Model_Exception_Validation $e) {
        // handle errors
    }

}
```

The next request uses the flash to display a message:

```
public function show()
{
    if ($msg = $this->flash['success']) {
        // set variables for view to display the message
    }
}
```

1.3.8 Filters

Filters allow us to write code that is executed before or after the action method that is requested. We define these by making a declaration of the filter in the `_initialize` method of the controller. There are two different types of filters available in the framework:

- `beforeFilter`
- `afterFilter`

Before Filter

Before filters get executed before the action in the current request. This allows us an easy way to add custom code that must execute before every action for a controller. The second argument to `beforeFilter` is an array of options:

Before filters get executed before the action in the current request. This allows us an easy way to add custom code that must execute before every action for a controller. The second argument to `beforeFilter` is an array of options:

- `only`: Only execute the filter before these methods
- `except`: Execute the filter before all methods except these

Execute `_checkAccess()` before all actions in this controller:

```
protected function _initialize()
{
    $this->beforeFilter('_checkAccess');
}
```

Execute `_checkAccess()` before all actions except `index` and `search`:

```
protected function _initialize()
{
    $this->beforeFilter('_checkAccess', array('except' =>
                                        array('index', 'search')));
}
```

After Filter

After filters get executed after the action in the current request. This allows us an easy way to add custom code that must execute after every action for a controller. The second argument to `afterFilter` is an array of options:

- `only`: Only execute the filter after these methods
- ‘‘except‘: Execute the filter after all methods except these

Execute `_logUser()` after *download‘* and `print` actions:

```
protected function _initialize()
{
    $this->afterFilter('_logUser', array('only' =>
                                    array('download', 'print')));
}
```

1.4 Views

1.4.1 Overview

In the MVC pattern, presentation logic is strictly separated from application logic. In the framework, views are templates for presentation that are are written in HTML with embedded PHP helpers.

1.4.2 Templates

Templates are stored in `app/views`. Each controller has its own directory for template files based on the name of the controller (eg. `UsersController` => `app/views/users`). There is also a `app/views/shared` directory to store templates that are shared between different controllers.

Templates are typically HTML with bits of embedded PHP code. The PHP code can only be used for presentation logic. For increased readability we use PHP's Alternative Syntax for control structures. This applies to all control structures including `if`, `else`, `elseif`, `while`, `for`, and `switch`:

```
<? if (! $this->user): ?>
  <div>No User</div>

<? else: ?>
  <div><?= Welcome, $this->h( $this->user->name ) ?></div>

<? endif ?>
```

Instance Variables

An action and its corresponding view share instance variables. There is no need to explicitly assign variables to a view object:

```
class UsersController
{
    public function show()
    {
        $this->user = User::find( $this->params['id'] );
    }
}
```

In the example aboce, `$this->user` is available in both the controller and its view.

Accessing instance variables of the view that do not exist will not cause a PHP notice as they normally would. Instead, they will simply return NULL.

Short Tag Emulation

PHP includes a feature called short_open_tag that allows you to write less verbose and more readable templates:

```
<?php if ($foo): ?>           long tags  (wrong)
<? if ($foo): ?>              short tags (right)

<?php echo $this->h($foo) ?>  long tags  (wrong)
<?= $this->h($foo) ?>         short tags (right)
```

The short tags are much nicer but are sometimes avoided for portability. In environments where the short_open_tag feature has been disabled, they can't normally be used.

The framework includes a special feature that overcomes this limitation. Short tags can be used all the time, regardless of the state of `short_open_tag` in the PHP environment. Short tags are preferred and should always be used.

Array Shorthand

The view has many helpers available that provide powerful features such as form generation. You will use these frequently. Most of these helpers have options that are specified as an associative array:

```
<? $form = $this->formFor('message', array('url' => array('action' => 'create'))) ?>
      <?= $form->textField('subject', array('class' => 'text')) ?>
      <?= $form->textArea('content', array('class' => 'text', 'rows' => '5')) ?>
<? $form->end() ?>
```

Since we use these options so often, views provide a special feature that allows associative arrays to be written as `[]` instead of `array()`.

Here's the example from above but cleaned up with the `[]` notation:

```
<? $form = $this->formFor('message', ['url' => ['action' => 'create']]) ?>
      <?= $form->textField('subject', ['class' => 'text']) ?>
      <?= $form->textArea('content', ['class' => 'text', 'rows' => '5']) ?>
<? $form->end() ?>
```

Using this notation for associative arrays makes the views easier to type and read. Always use this when specifying options for helpers.

Escaping Output

It is your responsibility to always escape your output:

```
WRONG:
<?= $this->user->name ?>

Right:
<?= $this->h( $this->user->name ) ?>
```

Warning: Never send data to the output without escaping it!

1.4.3 Helpers

Views separate the presentation from the controllers and models. Views are allowed to have logic, provided that the logic is only for presentation purposes. This presentation logic is small bits of PHP code embedded in the HTML.

Bits of presentation logic code can be extracted into helper methods. Once extracted, a helper method can be called in the view in place of the former code block. Extracting presentation logic into helpers is a best practice and helps views clean and DRY.

Helpers are simply methods of a class. The framework mixes the helpers into the view behind the scenes, and makes the appear as methods inside the view. An example of a helper class with a single `highlight()` helper follows:

```
class UsersHelper extends ApplicationHelper
{
    /**
     * Highlight a phrase within the given text
     * @param   string  $text
     * @param   string  $phrase
     * @return  string
     */
    public function highlight($text, $phrase)
    {
        $escaped = $this->h($text);
        $highlighted = "<strong class=\"highlight\">$escaped</strong>";

        if (empty($phrase) || empty($text)) {
            return $text;
        }
        return preg_replace("/($phrase)/", $highlighted, $text);
    }
}
```

Using the helper in a view:

```
<div><?= $this->highlight($this->var, 'bob') ?><div>
```

It is acceptable to put HTML into helper class methods because they exist to assist with presentation. However, it is not acceptable to put `print`/`echo` statements within a helper class. Helper methods always return a value that is displayed in the view like `<?= $this->highlight($text) ?>`.

The name of the helper class above is `UsersHelper`, so we know by convention that these methods will be available to the views of UsersController.

Organization

As shown above, helpers are methods that are organized into classes. A view typically has access to helpers from many sources:

- `UsersHelper`: Each controller has corresponding views that are specific to that controller. All views of the same controller share helpers that are not shared with views of other controllers. The views of `UsersController` share `UsersHelper`.
- `ApplicationHelper`: By default, the framework will create an `ApplicationHelper` which other helper classes extend. For example, `UsersHelper extends ApplicationHelper`. Due to this inheritence, all views of all controllers can access the helpers in `ApplicationHelper`.
- `Built-In Helpers`: The framework has a number of built-in helpers for tasks such as formatting numbers and dates, building forms, generating hyperlinks, and more. All of the built-in helpers of the framework are always available to all views.

The framework will instantiate helper classes automatically and then mix them together through overloading. Inside a view, helper methods from all of the sources above be called by simply using `<?= $this->helperMethod() ?>`.

Built-in Helpers

Many convenient helper classes are available within the framework itself, and the list of these will grow as time goes on. These are located in the `vendor/Mad/View/Helper/` directory and are always available all the time.

The helpers provided by the Mad_View component are very close to the helpers provided by Ruby on Rails 1.2.

1.4.4 Layouts

When building our application, most of the time you'll have a common layout between different pages. The layout being the menu, header, and navigation. The framework has a way to share this code between different actions so that we only need one shared layout between similar pages.

Layout templates are stored in the `app/views/layouts` directory.

Using Layouts

Let's take a look at an example layout template:

```
<!-- in /app/views/layouts/myLayout.html -->
<html>
  <head>
    <title><?= $this->h( $this->pageTitle ) ?></title>
  </head>
  <body>
```

```
    <?= $this->contentForLayout ?>
  </body>
</html>
```

You'll notice the variable `<?= $this->contentForLayout ?>` in the template. This is a special variable that tells us where our action template will be rendered within the layout code.

We can use this layout by using the controller's `setLayout()` method in `_initialize`:

```
class UsersController
{
    protected function _initialize()
    {
        // add this layout for all actions
        $this->setLayout('myLayout');
    }

    protected function helloWorld()
    {
        $this->pageTitle = "Hello From Users!";
    }
}
```

You'll notice we can also set variables in our actions to be available in the layout template. Now if we were to add a template for our `helloWorld` action:

```
<!-- in /app/views/users/helloWorld.html -->
<h1>Hello World</h1>
```

When the helloWorld action renders, it will first render the action template (`app/views/users/helloWorld.html`). Then it will replace our `$this->contentForLayout` layout variable layout with this content to produce our final result:

```
<html>
  <head>
    <title>Hello From Users!</title>
  </head>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
```

Disabling Layouts

There are times when you won't want every action in the controller to use our layout (especially when using Ajax). It is easy enough to disable layout for a specific action by adding `$this->useLayout(false)` to that method:

```
class UsersController
{
    protected function _initialize()
    {
        // set this layout for all actions
        $this->setLayout('myLayout');
    }

    public function showSpecial()
    {
        // don't use layout here
```

```
        $this->useLayout(false);

        $this->render(array('text' => '...'));
    }
}
```

1.4.5 Partials

Partial templates are snippets of HTML code that are reusable between different actions. They make it easy to make our templates as DRY as possible. Partial templates are named with a leading prefix to differentiate them from our normal templates. An example would be: `app/views/users/_user.html`.

Render Partial

Rendering a partial template within another template is done using the helper method: `render(['partial' => '...'])`. The leading underscore and extension is omitted when referring to the partial file:

```
<div>
<? foreach ($this->users as $user): ?>
    <?= $this->render(['partial' => 'user']); ?>
<? endforeach ?>
</div>
```

Any instance variables available in the main template are also available in partial templates.

1.5 Support Objects

The framework provides various utility classes to take care of common operations such as logging, configuration, and debugging. These can be found in `/vendor/Mad/Support`.

1.5.1 Logging

The framework exposes a logger object to the application. The logger writes to different logs depending on the environment in which the application is run:

- Development log data is sent to `/log/development.log`
- Testing log data is sent to `/log/test.log`
- Production log data is sent to `/log/production.log`

The logger is a preconfigured instance of Horde_Log. See its documentation for details on its standard log levels, usage, etc.

You can access the logger instance from a controller through the `_logger` property:

```
class CustomersController extends ApplicationController
{
    public function index()
    {
        $this->_logger->info('Informational message');

        $this->_logger->debug('Debug message');
```

```
    }
}
```

1.5.2 Base Object

When working with objects in PHP, and our framework models in particular, we want to have consistent interfaces to our objects and promote encapsulation.

PHP provides some interesting challenges to this goal because it provides properties, methods, and ways of handling missing properties and methods. How to use all of these features is left to the developer, and many PHP programs have inconsistent interfaces as a result.

Models in the framework expose their attributes as PHP properties. If the database table `users` has a column `name`, then the model object exposes it as `$user->name`. This is always predictable and models should not have attributes other ways like `$user->getName()`, `$user->name()`, etc. for consistency and clarity.

Models extend `Mad_Model_Base`, which itself extends `Mad_Support_Object`. This base object provides features for creating objects with uniform attribute access. You can use these features for adding virtual attributes to models or for creating your own objects.

Here is a simple example of creating an object called User, which extends the base object but is does not have a corresponding database table:

```
class User extends Mad_Support_Object {
  protected $_name;
  protected $_phone;

  public function __construct() {
    $this->attrAccessor('name', 'phone');
  }
}

$user = new User;
$user->name = 'Fred';
echo $user->name; // Fred
```

The method `attrAccessor()` creates an attribute on the instance that can be both read and written. Two other variations exist: `attrReader()` creates attributes that can only be read, and `attrWriter()` creates attributes that can only be written.

The advantages of the above example are not clear until the implementation of User needs to change. The attributes promote encapsulation by allowing the implementation to change while the interface remains the same:

```
class User extends Mad_Support_Object {
  protected $_name;
  protected $_phone;

  public function __construct() {
    $this->attrAccessor('name', 'phone');
  }

  public function getName()
  {
    return isset($this->_name) ? $this->_name : 'Anonymous';
  }
}
```

```
$user = new User;
echo $user->name; // "Anonymous"
```

In the example above, calling `$user->name` now consults the new `getName()` method. The implementation has changed but the interface remains the same (`$user->name`).

Mad_Support_Object works by using the missing property handlers `__get()` and `__set()`. When one of our virtual attributes is accessed, it first scans for a method like `getName()` or `setName()` on the object. If not found, it looks for a protected property like `_name`. If that is also not found, it searches for a method called `_get()` or `_set()`. Finally, an exception is thrown if nothing is available to satisfy the conditions.

You can use Mad_Support_Object to create you own custom objects as shown above, or to add new virtual attributes to objects that inherit from Mad_Model_Base. For more details on the implementation, see the inline source documentation for Mad_Support_Object.

1.5.3 Array Object

The Standard PHP Library (SPL) provides ArrayObject, which facilitates building objects with an array-like interface. In PHP and the framework in particular, we often work with associative arrays. `Mad_Support_ArrayObject` extends SPL's ArrayObject to provide additional conveniences for working with associative arrays.

To use it, instantiate a new Mad_Support_ArrayObject:

```
// new array
$a = new Mad_Support_ArrayObject();
$a['foo'] = 'bar';
echo $a['foo']; // bar

// existing array
$existing = array('foo' => 'bar');
$a = new Mad_Support_ArrayObject($existing);
echo $a['foo']; // bar
```

One advantage of using Mad_Support_ArrayObject is its handling of nonexistant keys in the array. In PHP, when an array contains a value that does not exist, a notice is raised. This frequently leads to messy `isset()` checks with the ternary operator. These are tedious, error-prone, and hard to read. Instead, Mad_Support_Object just returns NULL when a key does not exist:

```
// php array
$a = array();
var_dump($a['foo']);  // NULL, but PHP notice raised

// array object
$a = new Mad_Support_ArrayObject();
var_dump($a['foo']);  // NULL, and no PHP notice raised
```

A similar issue has to do with default values. When a key does not exist or the value at that key is NULL, we often want a default value that is not NULL. This is done with the `get()` method:

```
// php array
$a = array();
$foo = isset($a['foo']) ? $a['foo'] : 42;

// array object
$a = new Mad_Support_ArrayObject();
$foo = $a->get('foo', 42);
```

The feature shown very useful when used with GET and POST parameters, or any array where the keys and values are unreliable. In fact, the `$this->params` object accessible in controllers is built using Mad_Support_ArrayObject.

Another useful utility method is `update`, which will update the array object with the contents of another array in the same way as `array_merge()`:

```
// php array
$a = array('foo' => 'bar');
$b = array('baz' => 'qux');
$a = array_merge($a, $b);
var_dump($a); // array('foo' => 'bar', 'baz' => 'qux')

// array object
$a = new Mad_Support_ArrayObject(array('foo' => 'bar'));
$a->update(array('baz' => 'qux'));
var_dump($a); // array('foo' => 'bar', 'baz' => 'qux')
```

There are other useful methods available in Mad_Support_ArrayObject for getting the keys and values, popping off values, clearing the array, and more.

1.5.4 Extension Proxy

One of the problems that hampers the testability of PHP code is the coupling created by accessing all of the PHP global functions. This happens often because a large number of useful extensions are accessed only through global functions. Consider the following code snippet:

```
$res = ldap_connect($host, $port);
if (! $res) {
  // error logging
  return false;
}
```

There are two code paths shown above: the connection succeeding, and it failing. Both of them are very difficult to test because of the coupling to the global function `ldap_connect()` provided by the LDAP extension.

To make it succeed, you'd need an LDAP server. Causing it to fail is easier but the could take a very long time until the connection timeout occurs. Also, the code can't be tested at all without the LDAP extension. All of these problems are unacceptable.

The solution is to use to the extension through an object instead of calling the extension function directly. Since most PHP extensions prefix all of their functions with the name followed by an underscore, it's easy to wrap them. This is what Mad_Support_ExtensionProxy provides.

Our connection example becomes:

```
$ldap = new Mad_Support_ExtensionProxy('ldap');

$res = $ldap->connect($host, $port);
if (! $res) {
  // error logging
  return false;
}
```

The difference in usage is trivial but this version is easily testable. It now depends only on an `$ldap` instance, which the class needing LDAP can receive in its constructor. To test, now just pass a mock object for `$ldap`.

1.6 Unit Testing

1.6.1 Overview

A good automated testing strategy is crucial to keeping our applications maintainable. The framework provides a testing structure suite to get you on the right track setting up tests. Test files are stored under the `test/` directory.

The framework refers to two different groups of tests. Unit Tests for our models, and Functional Tests for our controller and views. As you create stubs for your models and controllers using `script/generate`, the framework will automatically create corresponding test stubs.

1.6.2 Running Tests

Installing PHPUnit

Our testing framework is built on PHPUnit and this must be installed on your system. To check if PHPUnit is installed properly, run the `phpunit --version` command:

```
$> phpunit --version

PHPUnit 3.2.0beta9 by Sebastian Bergmann.
```

If the output does not look similar to the above and indicate version 3.2 or greater, please see the installation instructions in the PHPUnit Manual.

Running a Single Test

You can run any single test file in the `test/unit/` or `test/functional/` directories:

```
$> cd /your-application/test/unit
$> phpunit DocumentTest
```

The above will run only the tests that have been written for the Document model. It may take a few seconds and should display output that is similar to:

```
PHPUnit 3.2.0beta9 by Sebastian Bergmann.
..........

Time: 16.990113019943
OK (10 Tests)

$>
```

Each dot represents a passed test. If there are any Failures or errors, they will be represented by `F` or `E` respectively with a message describing what went wrong:

```
PHPUnit 3.2.0beta9 by Sebastian Bergmann.
F.........

Time: 11.186962842941
There was 1 failure:
1) testFindByPk(DocumentTest)
expected <1234> but was: <0>
../your-application/test/unit/DocumentTest.php:47
../vender/Mad/Test/Unit.php:125
```

```
FAILURES!!!
Tests run: 10, Failures: 1, Errors: 0, Incomplete Tests: 0.

$>
```

We are given a wealth of information to fix what went wrong including the name of the test that failed, the value we expected in our test assertion, and the backtrace of the error.

Running All Tests

All of the tests for your application are located in the `test/` directory under you application's root directory. To run all of the tests, change to the `test/` directory and run `AllTests.php`:

```
$> cd /your-application/test
$> phpunit AllTests.php
```

Running a Test Group

Tests are organized into two groups: unit tests and functional tests. When you run `AllTests.php` as shown above, it will run all tests in both groups.

It is often convenient to run all of the tests of a particular group, e.g. just the unit tests or just the functional tests. You can do this with the `--group` switch for PHPUnit:

```
$> cd /your-application/test
$> phpunit --group unit AllTests
```

1.6.3 Populating the Database

A fixture is the system's state before the test is performed. We set this up by loading sample data into the database for the test to perform. This sample data is loaded before the test and then cleared out at the end of the test. This ensures that no test will have an effect on the operation of a different test.

YAML Overview

YAML (Rhymes with "camel") is our preferred method for loading test data. The YAML fixture files can be found in `test/fixtures` and should end with a `.yml` extension.

YAML is great because it is very readable. YAML is formatted using a few simple guidelines. Note that spacing is important (2 space indents, no tabs).

A comment:

```
# a comment in YAML format
```

A Mapping is a key associated with a value:

```
String:  Any string of characters
Int:     12
Boolean: true
Null:    NULL
Float:   3.43
```

A Sequence is a list of items:

```
- item 1
- item 2
- item 3
```

An Inline Sequence is a shortcut for writing a sequence when your values consist of only one word:

```
[item1, item2, item3]
```

A Mapping of a Sequence:

```
teardown:
  - do this first
  - do this second
  - do this third
```

A Mapping of an Inline Sequence:

```
requires: [folders, documents]
```

A Mapping of a Mapping:

```
public:
  id:          1
  parent_id:   0
  name:        Documents
  path:       ./Documents
```

Fixture Files

Our fixtures are by convention named after the table in which they are inserted. So in the simplest form, inserting two records into the folders table would look like the following and would be in `test/fixtures/folders.yml`:

```
public:
  id:        1
  parent_id: 0
  name:      Documents
  path:      ./Documents

private:
  id:         2
  parent_id:  1
  name:       Private
  path:       ./Documents/Private
```

This fixture would load two records into the database, but also make the yml records accessible within the test itself by their respective names (public/private). It is important to name each record with a unique name.

1.7 Model Testing

1.7.1 Overview

Model tests are stored in `test/unit`. Once we have some fixtures written, testing the model functionality is straightforward. We specify what data to load, and make sure that our model functionality returns the correct data when it is executed.

Each unit test class is named after the model class with a "Test" suffix and extends `Mad_Test_Unit`. Most of these details are already handled when the stubs are generated for the model file. Tests classes have two special methods that are optional. `setUp()` will execute before each test, and `tearDown()` will execute after each test finishes:

```
class FolderTest extends Mad_Test_Unit
{
    public function setUp()
    {
        // this code is run before each test
    }

    public function tearDown()
    {
        // this code is run after each test
    }
}
```

Any method within the class that begins with the prefix `test` will be considered a test and will be executed when PHPUnit runs. Most of the time you will write at least one test per public method in your model. More often than not you will write more:

```
class FolderTest extends Mad_Test_Unit
{
    // this is considered a test
    public function testGetParentFolders()
    {
    }

    // this is not a test (not prefixed with 'test')
    public function doSomething()
    {
    }
}
```

Most tests will be performed with the use of PHPUnit's assertion methods. When the test executes, PHPUnit ensures that each assertion performs correctly in order for the test to pass. The three most common assertions are `assertTrue`, `assertFalse`, and `assertEquals`:

```
class FolderTest extends Mad_Test_Unit
{
    public function testGetParentFolders()
    {
        $this->assertTrue($folder instanceof Folder);

        $this->assertFalse(empty($folder->name));

        $this->assertEquals('The Description', $folder->description)
    }
}
```

1.7.2 Loading Fixtures

Loading fixture data in a unit test is simple. It can be put in the setUp() method which gets run before each test:

```
public function setUp()
{
    $this->fixtures('folders');
```

```
    $this->fixtures('documents', 'document_types');
}
```

It can also be loaded individually per test:

```
public function testSomeFunctionality()
{
    $this->fixtures('folders');
}
```

1.7.3 Data by Name

When a fixture gets loaded, we also get access to the records in the fixture file by name. This allows us to write tests that are more resilient to changes within the fixture. Each name has an associative array of values from the fixture.

Fixture data in `folders.yml`:

```
public:
  id:        123
  parent_id: 0
  name:      1. Documents
  path:      ./1. Documents
```

Access the fixture record by name:

```
public function testSomeFunctionality()
{
    $this->fixtures('folders');

    // a Folder object is instantiated with the fixture data
    $folder = $this->folders('public');

    // assert the name is correct
    $this->assertEquals('1. Documents', $folder->name);
}
```

1.7.4 Database Access

Most of the tests written to test Model functionality will not need to access the database directly, but will instead verify data from a fixture file. If you need to access the database directly you can do so by using `$this->_conn`:

```
public function testQuerySomeData()
{
    $results = $this->_conn->select("SELECT * FROM folders");
    foreach ($results as $row) {
        print $row['name'];
    }
}
```

1.7.5 Test Helpers

If we have custom assertion or test helper methods, we can share them using the `MadTestHelper` class. Adding public methods to `test/MadTestHelper.php` will make them accessible from all of our unit and functional tests:

```
class MadTestHelper
{
    public function clearUploads()
    {
        Mad_Support_FileUtils::rm_rf(UPLOAD_DIR);
    }
}

class DocumentTest extends Mad_Test_Unit
{
  public function setUp()
  {
      $this->clearUploads();
  }
}
```

1.7.6 Assertions

These assertions are added by Mad_Test_Unit to make testing models more convenient.

AssertDifference

This assertion uses a block style syntax to make sure that a expression given yields different results after a block of code has executed. The first argument to this assertion is an expression. The second is an integer that represents the difference between the expression's result before and after the block is finished. The block is anything that comes before we call the `end()` method:

```
// assert count is +1 after create() finishes
$diff = $this->assertDifference('User::count()', 1);
    User::create(array('username' => 'lebowski'));
$diff->end();
```

The difference expected in this case is that `User::count()` will increase by 1 after the creation is finished. This is an optional argument, and will default to 1 when not specified. We can alternately use a negative number if the net count would have decreased during the block:

```
// assert count is -1 after delete() finishes
$diff = $this->assertDifference('User::count()', -1);
    User::delete(1);
$diff->end();
```

AssertNoDifference

This assertion works very similarly to its counterpart, but asserts that no change takes place when the expression is evaluated before and after the block executes:

```
// make sure that the count is the same after create() finishes
$diff = $this->assertNoDifference('User::count()');
    User::create(array('username' => ''));
$diff->end();
```

1.8 Functional Testing

Now that we have our models tested, we can go to some higher level tests. We call the code that tests our controllers functional tests, and they are stored in `tests/functional`.

Functional tests follow all the same conventions that unit tests do and can load fixtures using the `fixtures()` method. Functional tests subclass Mad_Test_Functional, which itself subclasses Md_Test_Unit:

```
class DocumentsControllerTest extends Mad_Test_Functional
{
    public function testIndex()
    {
    }
}
```

Functional tests have many features that we can use to test that our controllers and views are working properly. We want to test aspects like:

- Was the request routed to the correct controller?
- Was the web request successful?
- Were we redirected to the right page?
- Were the correct cookies/session data set?
- Was the correct template(s) rendered?
- Was the template rendered correctly?

The Functional tests can send a test request to the application and receive the response:

This requires our tests to have both the request and response objects defined in our `setUp()` method. This is already generated within the stub file when creating new controllers with `script/generate`:

```
class DocumentsControllerTest extends Mad_Test_Functional
{
    public function setUp()
    {
        $this->request  = new Mad_Controller_Request_Mock();
        $this->response = new Mad_Controller_Response_Mock();
    }
}
```

1.8.1 Performing Requests

Requests can be simulated using few different methods.

`get()` performs a simulated HTTP GET request to an action of the controller:

```
public function testIndexPage2()
{
    $this->get('index', array('page' => '2'));
}
```

In the example above, a GET request is made to the `index` action of the controller being tested. The params are also set for page 2.

`post()` performs a simulated HTTP POST request to an action of the controller:

```
public function testEditSavesDocument()
{
    $this->post('edit', array('id' => '123'));
}
```

In the example above, a POST request is made to the `edit` action of the controller being tested. The params are also set for ID 2.

`followRedirect()` will do an HTTP GET on the redirect location in a 300 level response:

```
public function testSomething()
{
  // index action performs redirectTo(array('action' => 'binder'));
  $this->get('index');

  // this will perform another GET to follow the redirect
  $this->followRedirect();
}
```

`recognize()` doesn't actually perform a request, but will evaluate the routing and instantiate the correct controller for the reqeuest. This is a good way to test out that routing data is working correctly:

```
public function testSomething()
{
    $this->recognize('/explore/folder/123');
}
```

Before a request is made, you can modify the request data to your heart's content. Some things that will probably commonly be changed are: Cookies, Session Data, Flash Data, etc. This can help set the environment for your test:

```
public function testSomething()
{
    $this->request->setCookie('FOLDERID', '123');
    $this->request->setRemoteIp('172.17.118.5');
    $this->request->setIsAjax(true);

    $this->get('/explore/folder');
}
```

1.8.2 Available Data

After a request has been performed, we have access to a valid response object. Most of the time assertions done on the response will be done through our custom assertion methods mentioned below, but one handy option you have for debugging your tests is to print out the response body:

```
public function testIndexBody()
{
    $this->get('index');

    echo $this->response;
}
```

We also have access to some new data that we can use to assert that the correct actions have been executed during the request.

`getAssigns()` will get us any template variable that was set during the action:

```
public function testIndexAssignsPageId()
{
    $this->get('index');

    $pageId = $this->getAssigns('PAGE_ID');
}
```

`getCookie()` will get us any cookie data set during the action:

```
public function testShowSetsDocumentIdCookie()
{
    $this->get('show', array('id' => 3));

    $binderId = $this->getCookie('BINDER_ID');
}
```

`getSession()` will get us any session data set during the action:

```
public function testShowSetsSelectedFolderInSession()
{
    $this->get('show');

    $folder = $this->getCookie('FOLDER_ID');
}
```

`getFlash()` will get us any flash data set during the action:

```
public function testLoginFlashesSuccessMessage()
{
    $this->get('login');

    $msg = $this->getFlash('SUCCESS_MSG');
}
```

1.8.3 Assertions

There are many assertions available to the functional tests. These will help evaluate that the request/response gets executed correctly.

AssertRouting

`assertRouting()`: asserts that the URL given to `recognize()` set the given params correctly:

```
public function testSomething()
{
    $this->recognize('/explore/binder/123');

    // assert that the params['id'] was set correctly from the url
    $this->assertRouting('id' => '123');
}
```

AssertNoRouting

`assertNoRouting()`: does the exact opposite of `assertRouting()`. It makes sure that the URL given to `recognize()` was not routed correctly:

```
public function testSomething()
{
    $this->recognize('/explore/binder/asdf');

    $this->assertNoRouting();
}
```

AssertAction

`assertAction()`: asserts that the given action/controller were executed during the request:

```
public function testSomething()
{
    $this->recognize('index');

    $this->assertAction('index', 'DocumentsController');
}
```

AssertAssigns

`assertAssigns()`: asserts that the given variable was assigned to the given value:

```
public function testSomething()
{
    $this->get('binder', array('id' => 123));

    $this->assertAssigns('PAGE_ID', 'binderExplore');
}
```

AssertAssignsCookie

`assertAssignsCookie()`: asserts that the given cookie was assigned to the given value:

```
public function testSomething()
{
    $this->get('binder', array('id' => 123));

    $this->assertAssignsCookie('COOKIE_NAME', 'cookie value');
}
```

AssertAssignsSession

`assertAssignsSession()`: asserts that the given session was assigned to the given value:

```
public function testSomething()
{
    $this->get('binder', array('id' => '123');

    $this->assertAssignsSession('SESSION_NAME', 'session value');
}
```

AssertAssignsSession

`assertAssignsFlash()`: asserts that the given flash was assigned to the given value:

```
public function testSomething()
{
    $this->get('binder', array('id' => '123');

    $this->assertAssignsFlash('FLASH_NAME', 'flash value');
}
```

AssertResponse

`assertResponse()`: asserts that the response was successful, redirected, missing, an error, or a specific HTTP code:

```
public function testSomething()
{
    $this->get('binder', array('id' => '123');
    $this->assertResponse('success');

    $this->get('index');
    $this->assertResponse('redirect'); // 302

    $this->get('missing_action');
    $this->assertResponse('missing'); // 404

    $this->get('moved_action');
    $this->assertResponse(301); // 301
}
```

AssertRedirectedTo

`assertRedirectedTo()`: assert that the response is a redirect to the given URL:

```
public function testSomething()
{
    $this->get('index');

    $this->assertRedirectedTo(array('action' => 'binder'));
}
```

AssertResponseContains

`assertResponseContains()`: assert that the given string/regexp is contained in the response body:

```
public function testSomething()
{
    $this->get('index');

    $this->assertResponseContains('Documents');

    $this->assertResponseContains('/[0-9]{1,} Documents/');
}
```

Another assertion, `assertResponseDoesNotContain()`, asserts that the response does not contain the content.

AssertSelect

`assertSelect()`: assert that we find an HTML tag that matches the given CSS selector and options. This assertion is probably the one you'll use the most during your testing.

The syntax of assertSelect is very simple if you know CSS selector syntax:

- `div` : an element of type div
- `div.warning` : an element of type div whose class is "warning"
- `div#myid` : an element of type div whose ID equal to "myid"
- `div[foo="bar"]` : an element of type div whose "foo" attribute value is exactly equal to "bar"
- `div[foo~="bar"]` : an element of type div whose "foo" attribute value is a list of space-separated values, one of which is exactly equal to "bar"
- `div[foo*="bar"]` : an element of type div whose "foo" attribute value contains the substring "bar"
- `div span` : an span element descendant of a div element
- `div > span` : a span element which is a direct child of a div element

  We can also do combinations of these options such as:

- `div#folder.open a.class_name`
- `a[href="http://example.com"][title="example"].selected.big > span`

The second argument determines what we're matching in the content or number of tags. It can be one 4 options:

- `content` : match the content of the tag
- `true/false` : match if the tag exists/doesn't exist
- `number` : match a specific number of elements
- `range` : to match a range of elements, we can use an array with the options '>' and '<'

There is an element with the id "binder_1" with the content "Test Foo":

```
$this->assertSelect('#binder_1', "Test Foo");
```

There is not an element with the id "binder_1" and the content "Test Foo":

```
$this->assertSelect('#binder_1', "Test Foo", false);
```

The "#binder_foo" id exists:

```
$this->assertSelect('#binder_foo');
$this->assertSelect('#binder_foo', true);
```

The "#binder_foo" id DOES NOT exist:

```
$this->assertSelect('#binder_foo', false);
```

There are 10 div elements with the class folder:

```
$this->assertSelect('div.folder', 10);
```

There are more than 2, less than 10 li elements:

```
$this->assertSelect('ul > li', array('>' => 2, '<' => 10));
```