Chapter 1

EXTENDING YII

The Yii framework is fairly complete in its own right, and through the extensions made available by others, you might be able to go a long time before you hit a development wall. But should that day come, Yii is easily extendible by anyone with decent Object-Oriented Programming skills and comfort with the framework.

Extensions in Yii are simply that: the creation of additional functionality not inherently built into the framework. In fact, the application template created by the `yiic` command already defines a couple of extensions, such as the `Controller` and `UserIdentity` classes (found within **protected/components**).

The term "extension" can be used in a couple of ways. First, there's the type of extension just mentioned and created by the framework for you. Or, you might create your own classes in a similar vein. This category of extensions are true extensions in that they add functionality beyond that defined in the framework itself, and are commonly created by more advanced Yii developers.

A second, more specific type of extension also extends the core functionality but are *intended to be shared with third parties*. In other words, instead of writing a chunk of code that will help you out on one or two projects, a Yii extension is also a way to share the extremely useful code that you've come up with.

In this chapter, I'll explain everything you need to know to create, use, and even deploy (for the use of others) your own extensions.

FUNDAMENTAL CONCEPTS

Before getting into the particulars of creating an extension, let's look at the fundamental concepts when it comes to the subject:

- Guidelines
- Structure

- Publishing assets
- Types of extensions

Guidelines

The purpose of an extension is not just to add needed functionality (including changing existing functionality), but to do so in a manner that's highly and easily reusable. Moreover, a complete extension is easily reusable not just by you, but by other Yii developers as well. Towards that end, there are many guidelines in place for writing extensions. Note that these are not absolute rules, just good and logical recommendations:

- Give your extension a unique but meaningful name
- If your name could possibly conflict with another extension name, use a unique but meaningful prefix (as a tip, use the same prefix for every extension you create)
- Write the extension to be self-contained, managing its own external dependencies (i.e., the user should not have to download additional packages to use your extension, as a general rule)
- All extension files should go within an extension directory that uses the same name as the extension itself (but in all lowercase letters)
- Extension classes should be named with a prefix to avoid collusion issues with other classes (again, your own unique prefix would be good here)
- Include good, reliable installation and configuration instructions
- Provide good documentation for usage, FAQ, and so forth
- Use meaningful version numbers and provide a CHANGELOG file when you update the extension
- Include a license
- Do your best to maintain the extension

> *{NOTE}* The Yii community is wonderfully generous. Even if you worry that your work might be amateurish, or too limited in scope, I would highly recommend you consider sharing your extensions with others.

And, of course, your extension should be written under the best practices, with respect to the programming in general and how code is written in Yii.

> *{TIP}* For help in picking the license that's most appropriate for you and the extension itself, see an online resource such as Choose a License.

Structure

Per the extension guidelines, all extension files should go within an extension directory that uses the same name as the extension itself, but in lowercase letters.

So if you're creating the *ArgyleBargle* extension, all of its files would go in the **argylebargle** directory.

Within the extension directory, you'd place your main class in the root folder. In this hypothetical example, that might be the file **ArgyleBargle.php**.

If **ArgyleBargle** is a fairly common term (e.g., you're making a jQuery or database-related extension), you'd probably want to prefix the extension directory and class to distinguish it. I might use **luargylebargle** and **LUArgyleBargle.php**.

Some extensions require only a single class file, meaning that the extension directory contains only that. Other extension types have their pieces appropriately divided into multiple subdirectories:

- **extDir/assets**
- **extDir/models**
- **extDir/views**

This is often the case when creating modules, which have an application-like structure, or widgets, which have view files.

Regardless of the complexity of the extension, you should also be in the habit of including:

- A README file
- A LICENSE file
- A CHANGELOG file

These files should be in plain text or Markdown format.

Publishing Assets

More complex extensions, particularly widgets and modules, often make use of resources that need to be in a public directory: CSS, JavaScript, or media. But extensions generally go in non-public directories, such as **protected/extensions**. And the browser should be prevented from loading any resources from the extension's directory, as browsers shouldn't have access to **protected** at all. This is an important security feature, that can be taken further by moving the entire **protected** directory out of the web root.

Because an extension might require publicly accessible resources, the Yii framework provides the CAssetManager class. This tool will handle the copying of resources from a non-public directory to the public **assets** directory.

> *{NOTE}* By default the resources will be copied to the **webroot/assets** directory, but this can be changed, if needed.

Assets are copied by invoking the `CAssetManager` class's `publish()` method. The first argument to this method is the path to the file(s) to be copied. Your primary extension class will normally invoke this method in a constructor or initialization method:

```
public function init() {
    $source = dirname(__FILE__) . '/assets/filename.css';
    Yii::app()->assetManager->publish($source);
```

By default, all published assets will go within an **assets** subdirectory, with a simple hashed name like *8e98dfc4*. This name is based upon the basename of the file(s) being copied. If you set the second argument of the `publish()` method to true, a common assets subdirectory can be used by multiple extensions.

```
public function init() {
    $source = dirname(__FILE__) . '/assets/filename.css';
    Yii::app()->assetManager->publish($source, true);
```

> *{WARNING}* The **assets** directly must be writeable by the web server, but that is the default expectation.

If the asset being published is a single file, as in the above, it will be placed within the assets subdirectory (e.g., **assets/8e98dfc4/filename.css**). If the asset being published is a folder of files, then the files will be moved to an assets subdirectory:

```
public function init() {
    $source = dirname(__FILE__) . '/assets/css';
    Yii::app()->assetManager->publish($source);
```

Assuming the extension's **css** directory had both **style.css** and **print.css**, those two files would end up as **assets/8e98dfc4/style.css** and **assets/8e98dfc4/print.css** (not **assets/8e98dfc4/css/style.css**). If you have a nested directory structure that you would like to copy intact (**Figure 19.1**), you'd use this code:

```
public function init() {
    $source = dirname(__FILE__) . '/assets';
    Yii::app()->assetManager->publish($source);
```

When the assets are published, the structure will be maintained (**Figure 19.2**).

The third argument to `publish()` is an indicator of what level of copying is performed. By default, every subdirectory and file is copied. This equates to an argument value of -1. If you use the value 0, only the immediate files in the named

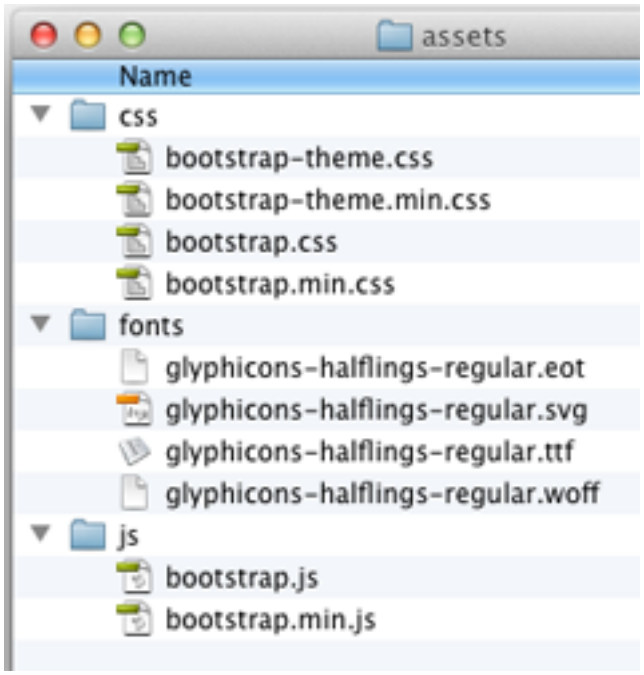

Figure 1.1: *An extension's collected assets.*

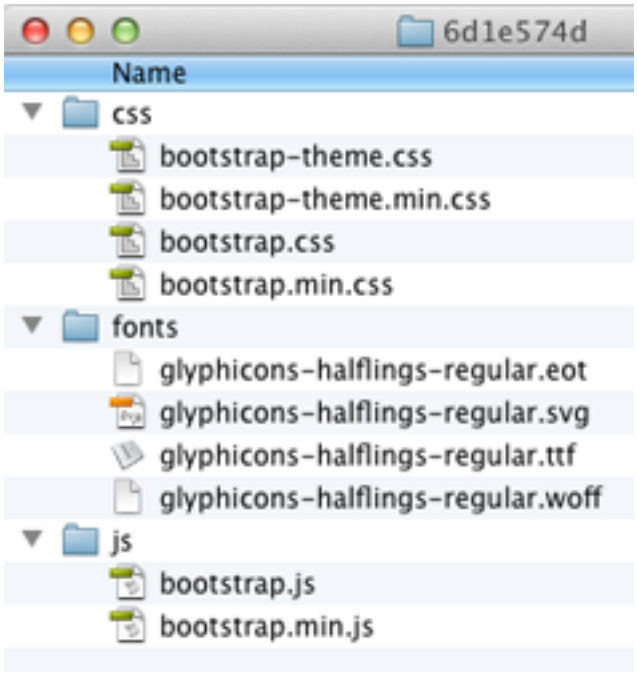

Figure 1.2: *The published assets.*

directory will be copied. If you use an N value, where N is any integer, that number of files and subdirectories will be copied.

The ability to specify how recursive of a copy to perform allows you to copy, for example, the CSS directory without copying the LESS directory you've also included with the extension. Or, as another example, the JavaScript file could be copied without copying the unit tests directory or un-minified JavaScript files.

When publishing a subdirectory with limited recursion, you'll likely need to invoke `publish()` more than once:

```
public function init() {
    $base = dirname(__FILE__) . '/assets/';
    $am = Yii::app()->assetManager;
    $am->publish($base . 'css', false, 0);
    $am->publish($base . 'js', false, 0);
```

The fourth and final argument to `publish()` is a Boolean indicating whether existing files should be overridden. When you're developing a new extension, and changing the files frequently, you'd likely want to set this to true:

```
public function init() {
    $source = dirname(__FILE__) . '/assets';
    Yii::app()->assetManager
        ->publish($source, false, -1, true);
```

However, this setting adversely affects performance, and would not be desired on a live, distributed extension.

With the default values, if the named asset is a file and it already has been published, its modification time will be checked before re-publishing. If the asset is a directory and the fourth argument is false, the asset manager will only check for the existence of the directory (no recursive check will be performed). If the fourth argument is true, every file and directory will be checked.

> *{WARNING}* Never delete a file from an assets subdirectory, as the assets manager doesn't check the detailed contents of a directory when deciding whether or not to republish assets.

When you publish assets, you'll need to know where they ended up in order to create proper references to them after the fact. The `publish()` method will return the new path to the asset. This value can be stored in an internal variable for later use:

```
class MyExt {
    private $_assetsUrl;
    public function init() {
        $source = dirname(__FILE__) .
            '/assets/filename.css';
        $this->_assetsUrl = Yii::app()
            ->assetManager->publish($source);
    }
}
```

Now, other methods within the class can reference `$this->_assetsUrl`. Understand that even though the asset manager won't republish assets if no changes have been made (unless you pass a fourth argument of true), the `publish()` function will still return the used path for the extension's assets.

The final thing to note is that you'll need to tell the application to include your CSS or JavaScript file. This is done using the `registerCssFile()` and `registerScriptFile()` methods. You'd call either after copying the assets:

```
public function init() {

    // Set the sources:
    $source = dirname(__FILE__) . '/assets/';
    $css = $source . 'css/bootstrap.css';
    $js = $source . 'js/bootstrap.js';

    // Publish them:
    $am = Yii::app()->assetManager;
    $cssUrl = $am->publish($css);
    $jsUrl = $am->publish($js);

    // Register them:
    $cs=Yii::app()->clientScript;
    $cs->registerCssFile($cssUrl);
    $cs->registerScriptFile($jsUrl);

}
```

Extension Types

The particulars of any extension will be dependent upon the *type* of extension being created. Fundamentally, it's a question of what role the extension will play in a Yii application, and therefore, from what class the extension's primary class will be derived (if any).

Save for modules, I will next walk through the basic types of extensions, and outline how they're written, before delving into some examples of specific types. Due to the more complex nature of modules, I'll discuss and demonstrate creating a module later in the chapter.

Application Components

An application component extension serves the same purpose as the application components built into the Yii framework, such as the URL manager, the asset manager, the caching component, the session component, and so forth. This is where you implement specific functionality that a website needs that's not particular to any model. Application components are particularly good for making code or data available to every aspect of an application; controllers, models, and views can all readily access application components.

Application components must implement the **IApplicationComponent** interface or extend the **CApplicationComponent** class:

```
class MyAppComponent extends CApplicationComponent {
}
```

Within the class, you can define any attribute or method you want:

```
class MyAppComponent extends CApplicationComponent {
    public $var = false;
    private $_thing = 42;
    public function getThing() {
        return $this->_thing
    }
}
```

Once defined, you need to register the component with the application. This is done in the primary configuration file:

```
# protected/config/main.php
// Other stuff.
'components' => array(
    'myappcomponent' => array(
        'class' => 'ext.myappcomponent.MyAppComponent',
    ),
),
```

> *{TIP}* The path reference `ext` equates to **protected/extensions** (or wherever your extensions directory is).

That code tells the application that the *myappcomponent* component is an instance of the `MyAppComponent` class. (And the code assumes that the **MyAppComponent.php** file is within the **protected/extensions/myappcomponent** directory.)

With the component registered, you can access an instance of that class using `Yii::app()->myappcomponent`:

```
Yii::app()->myappcomponent->var = true;
Yii::app()->myappcomponent->getThing();
```

Application components are often configurable, affecting how they behave. For example, the user component can be told to allow for auto login (i.e., "remember me" functionality):

```
# protected/config/main.php
// Other stuff.
'user'=>array(
    // enable cookie-based authentication
    'allowAutoLogin'=>true,
),
// More stuff.
```

The configuration names and values just correspond to public properties of the underlying class. To make your application configurable, just add public variables such as `$var` in the trivial example above. Then assign a value to that variable in your configuration:

```
# protected/config/main.php
// Other stuff.
'components' => array(
    'myappcomponent' => array(
        'class' => 'ext.myappcomponent.MyAppComponent',
        'var' => true,
    ),
),
```

Do be certain to give your public attributes default values, however, so that a failure to configure the value does not result in errors.

A final consideration for application components is whether there's any initial setup or other work to be performed before the component can be used. For example, does a database connection need to be made? If there is something to be done, define a `init()` method that performs any initialization work.

Widgets

Widgets are a specific, and common, type of extension used to add complex functionality to view files without embedding a lot of code and logic. Yii comes with a number of widgets built-in, as described in Chapter 12, "Working with Widgets." The built-in widgets:

- Present loads of information in grid format
- Represent jQuery UI components
- Present individual records in a nice format
- And more

A widget must extend the `CWidget` class (or an existing child class of `CWidget`). For example, to create your own variation on the Yii `CCaptcha` widget, you could just extend that class and override its methods, or change its default behaviors, to suit your needs.

When creating a brand-new widget, you extend the `CWidget` class and must define at least the `init()` and `run()` methods. The `init()` method will be called when `$this->beginWidget()` is used in a view file to create the widget instance. The `run()` method is called when the view file invokes `$this->endWidget()`:

```
# protected/extensions/mywidget/MyWidget
class MyWidget extends CWidget {
    public function init() {
        // Do whatever to start.
    }
    public function run() {
        // Do whatever to end.
    }
}
```

```
# protected/views/controllerID/view.php
<?php $this->beginWidget('ext.mywidget.MyWidget'); ?>
<?php $this->endWidget(); ?>
```

Some widgets, such as `CActiveForm` expect you to create data between the `beginWidget()` and `endWidget()` method calls (in the case of `CActiveForm`, the data created are form elements). The data placed between those method calls is captured by the widget by starting output buffering within the `init()` method and ending it in `run()`:

```
# protected/extensions/mywidget/MyWidget
class MyWidget extends CWidget {
```

```
    public function init() {
        // Do whatever to start.
        ob_start();
    }
    public function run() {
        // Do whatever to end.
        $data = ob_get_clean();
        // Use $data.
    }
}
```

```
# protected/views/controllerID/view.php
<?php $this->beginWidget('ext.mywidget.MyWidget'); ?>
<p>This data is being captured.</p>
<?php $this->endWidget(); ?>
```

Note that the output buffering captures the data, it does not output it. You'll need to echo the `$data` variable to do that.

If your widget doesn't need to capture any input, it can be created in a view file using just `widget()`:

```
# protected/views/controllerID/view.php
<?php $this->widget('ext.mywidget.MyWidget'); ?>
```

In that case, the `init()` and `run()` methods are still invoked in that order.

> *{NOTE}* You always need `init()` and `run()` methods in your widget class.

You can configure how a widget functions when it's created (in the view file) by passing additional arguments to the `beginWidget()` or `widget()` method:

```
# protected/views/controllerID/view.php
<?php $this->widget('ext.mywidget.MyWidget',
    array('name' => 'value')); ?>
```

The names of the array elements must match the public properties defined in the widget class:

```
# protected/extensions/mywidget/MyWidget
class MyWidget extends CWidget {
    public $var1;
```

```
    public $var2;
    public function init() {
        // Do whatever to start.
    }
    public function run() {
        // Do whatever to end.
    }
}
```

```
# protected/views/controllerID/view.php
<?php $this->widget('ext.mywidget.MyWidget',
    array('var1' = 42, 'var2' = false); ?>
```

In that example, the public `$var1` property in the class will be assigned the value 42, `$var2` will be assigned false.

If a property value must be assigned to use the widget, you would throw an exception in the `init()` method:

```
# protected/extensions/mywidget/MyWidget.php
class MyWidget extends CWidget {
    public $var1;
    public $var2;
    public function init() {
        // Do whatever to start.
        if ($this->$var1 === null) {
            throw new CException('You must provide a `$var1`
                value.');
        }
    }
    public function run() {
        // Do whatever to end.
    }
}
```

You can also reference the public attributes in the `run()` method, of course, including performing validation there, if you'd like.

Most widgets output HTML. For anything but the very simplest output, you should create your own view file (or files) for the widget. The view files would go within the **views** subfolder of your extension's directory, and be given the name **ViewFile.php**, as is the case with any controller's view file. You'd then use the `render()` method within the widget's `run()` method, as you would within a controller.

To put this all together, as a stupid, but comprehendible, example of this, say your *MyWidget* outputs some text within a DIV or a paragraph (which is not a good use of a widget). The view files would be:

```
# protected/extensions/mywidget/views/div.php
<div><?php echo CHtml::encode($output); ?></div>
```

```
# protected/extensions/mywidget/views/p.php
<p><?php echo CHtml::encode($output); ?></p>
```

Use of the widget (in a view file) would be:

```
# protected/views/controllerID/view.php
<?php $this->beginWidget('ext.mywidget.MyWidget',
    array('tag' => 'p')); ?>
This is the output.
<?php $this->endWidget(); ?>
```

Or:

```
# protected/views/controllerID/view.php
<?php $this->beginWidget('ext.mywidget.MyWidget',
    array('tag' => 'div')); ?>
This is the output.
<?php $this->endWidget(); ?>
```

And the widget itself would be defined as:

```
# protected/extensions/mywidget/MyWidget.php
class MyWidget extends CWidget {
    public $tag;
    public function init() {
        ob_start();
    }
    public function run() {
        $output = ob_get_clean();
        if ($this->tag == 'p') {
            $this->render($this->tag,
                array('output' => $output));
        } else {
            $this->render('div',
                array('output' => $output));
        }
    }
}
```

Controllers

The default Yii template, used when creating a new application, already creates a new base controller class, aptly named `Controller`:

```
# protected/components/Controller.php
<?php
class Controller extends CController {
    public $layout='//layouts/column1';
    public $menu=array();
    public $breadcrumbs=array();
}
```

Although very useful, and a reasonable step to take with any of your own applications, the `Controller` class is not a good candidate as a *distributable* extension. In Yii, if you create a controller intended as an extension, it should actually extend the `CExtController` class, not `CController`:

```
# protected/extensions/mycontroller/MyController.php
class MyController extends CExtController {
}
```

The reason for this difference is that the `CController` class will look within the **protected/views/ControllerID** directory by default when it comes to rendering views. Presumably, with a controller extension, the views will be distributed along with the class file. That puts the view files in, for example, **ext/mycontroller/views**, not **protected/views/ControllerID**. But if you base your controller extension on `CExtController`, calls to `render()` will pull from the **views** directory found with the extension class file.

Other than that, a controller extension is written and used like any other.

Actions

Actions are controller methods invoked as part of a request. These are methods defined within the controller, whose names start with *action*. For example, the `actionIndex()` method is normally the default method called for every controller (i.e., when no other action is specified).

It's standard for controller actions to be defined within the controller itself, but they can also be defined independently, thereby creating an *action extension*. As with any extension, the benefit of an action extension is reusability: if you have an action whose logic could be used by multiple controllers without change, you could define that as an extension and then tell each controller about it.

Actions must extend the `CAction` class, or an existing child class of `CAction`. Actions must define a `run()` method, which does the bulk of the work:

```
# protected/extensions/myaction/MyAction.php
class MyAction extends CAction {
    public function run() {
        // Do the work.
    }
}
```

To use the action in any controller, you just need to tell the controller about it. This next bit of code tells the `MyController` class to use the `MyAction` class for the `myaction` request:

```
# protected/controllers/MyController.php
class MyController extends CController {
    public function actions() {
        return array(
            'myaction' => 'ext.myaction.MyAction'
        );
    }
    // Other controller code.
}
```

If the action needs parameters–values passed to the action when it's executed, you can do that by creating arguments in the `run()` method:

```
# protected/extensions/myaction/MyAction.php
class MyAction extends CAction {
    public function run($thing) {
        // Do the work.
    }
}
```

When the `myaction` action is executed (associated with a controller), the value of `$_GET['thing']` will now be passed to the `run()` method.

Filters

Like actions, filters are also used by controllers. Filters perform a task before and/or after an action is executed. For example, the "access control" filter is used to restrict access to the execution of controller actions.

To create a filter extension, you'll need to extend the `CFilter` class (or extend a child of that class). The class should have a `preFilter()` method, which will be executed before the controller action. The class should also have a `postFilter()` method, which will be executed after the controller action. The `preFilter()` method needs

to return a Boolean indicating whether or not the controller action should be allowed to execute. The `postFilter()` method does not need to return anything. Both functions should take one argument, a filter chain.

The filter chain argument allows the filter to be one of many applied to an action. For example, the `actionDelete()` method of a controller may have both the "access control" and the "postOnly" filters applied to it.

Because a filter may be one in a sequence to be executed, you ought to call the `preFilter()` and `postFilter()` methods of the parent class in your filter extension. You can use the returned value as the value to be returned by your methods. Here, then, is the shell of a filter extension:

```
# protected/extensions/myfilter/MyFilter.php
class MyFilter extends CFilter {
    public function preFilter($fc) {
        // Do whatever.
        return parent::preFilter($fc);
    }
    public function postFilter($fc) {
        // Do whatever.
        return parent::postFilter($fc);
    }
}
```

The `filters()` method of the controller is used to apply the filter:

```
# protected/controllers/MyController.php
class MyController extends CController {
    public function filters() {
        return array(
            'accessControl',
            'postOnly + delete',
            'ext.myfilter.MyFilter + index'
        );
    }
    // Other controller code.
}
```

Behaviors

Behavior extensions add specific functionality to common MVC pieces. For example, a behavior might be defined that adds the ability to smoothly handle checkboxes in any model. As another example, the `TimestampBehavior` will automatically set a model's `create_time` attribute to the current moment when the model is first

created. It will also do the same for the `update_time` attribute when the model instance is updated.

The behavior extension must implement the `IBehavior` interface. This is normally accomplished simply by extending the `CBehavior` class. Behaviors intended for specific contexts, such as models in general or Active Record in particular, can extend the more specific `CModelBehavior` or `CActiveRecordBehavior` classes instead:

```
# protected/extensions/mybehavior/MyBehavior.php
class MyBehavior extends CActiveRecordBehavior {
}
```

To attach a behavior to a model, use the model's `behaviors()` method. This method returns an array of behaviors to attach. For each item, provide a behavior name. The name's value is an array. The most important element in this subarray will be the class associated with the added behavior:

```
# protected/models/SomeModel.php
public function behaviors() {
    return array(
        'behaviorName' => array(
            'class' => 'ext.mybehavior.MyBehavior'
        )
    );
}
```

The result is that the model now has access to the methods defined in the behavior as if the methods were defined in the model itself. The main difference being that these methods–the behavior–can be added to *any* model.

As you'll see in a later example, behaviors tied to models often use `beforeSave()` and other event-driven methods to perform steps in conjunction with the life of a model. This is how the `TimestampBehavior` works.

Validators

Validators are used by models to validate the model's properties. There are dozens of validators built into Yii, but you may need your own custom validator, too. Chapter 5, "Working with Models," showed how to define a validator within a model. To make a more reusable validator, you could build it as an extension instead.

A validator extension needs to extend `CValidator`. The class must define the `validateAttribute()` method. This method takes two arguments: a model instance and the attribute to validate. The method doesn't need to return anything, but instead adds an error to the method upon failure to validate:

```
# protected/extensions/myvalidator/MyValidator.php
class MyValidator extends CValidator {
    public function validateAttribute($model, $attr) {
        if (/* $model->$attr is NOT valid */) {
            $model->addError($attribute, 'Error message');
        }
    }
}
```

And that's all there is to it! To have your model use this validator, add it to the list of rules:

```
public function rules() {
    Yii::import('ext.myvalidator.MyValidator');
    return array(
        // Other rules.
        array('attrName', 'ext.myvalidator.MyValidator'),
    );
}
```

To make your validator configurable, define a public attribute for every configuration option:

```
# protected/extensions/myvalidator/MyValidator.php
class MyValidator extends CValidator {
    public $var1;
    public function validateAttribute($model, $attr) {
        if (/* $model->$attr is NOT valid */) {
            $model->addError($attribute, 'Error message');
        }
    }
}
```

Then, when you apply the rule to your model, provide an array of name-value pairs, where each name matches a public attribute:

```
public function rules() {
    Yii::import('ext.myvalidator.MyValidator');
    return array(
        // Other rules.
        array('attrName', 'ext.myvalidator.MyValidator',
            array('var1' => 'value')),
    );
}
```

Console Commands

Console commands are like controller actions meant to be run from a command-line interface. Similarly, you can create a console command as an extension that's distributable and usable in any application.

To create a console command extension, you need to extend the `CConsoleCommand` class. The class's `run()` method will do the actual work. This method can be written to accept one parameter:

```
class MyCommand extends CConsoleCommand {
    public function run($args) {
        // Do whatever.
    }
}
```

The `$args` array will store whatever values were passed as command-line parameters when the command is invoked:

```
yiic mycommand --param1=value
```

To make your command a bit more professional, you can also implement the `getHelp()` command. This method will be invoked when the user types `command-name --help`. The function should return a string, which will be printed:

```
class MyCommand extends CConsoleCommand {
    public function run($args) {
        // Do whatever.
    }
    public function getHelp() {
        return "Usage: mycommand --param"
    }
}
```

To make the command extension available to your application, you need to add it to the `commandMap` property of the console application. This is done in the **console.php** configuration file:

```
# protected/config/console.php
return array(
    'basePath'=>dirname(__FILE__).DIRECTORY_SEPARATOR.'..',
    'name'=>'My Console Application',
    'commandMap' => array(
        'mycommand' => array(
            'class' => 'ext.mycommand.MyCommand'
```

```
        )
    ),
// Other stuff.
```

If you want to be able to configure the console command, add public properties to the class and assign values to those properties in the configuration file.

Other Extension Types

Finally, you may have an extension that does not fit neatly into any of the listed categories. Remember that in Yii you can extend almost anything you need to extend!

As one example, you might like all of your ActiveRecord models to have certain attributes or methods. To do that, you'd create your own model class based upon `CActiveRecord`:

```
# protected/components/MyActiveRecord.php
class MyActiveRecord extends CActiveRecord {
    // Do whatever.
}
```

Then you can have all your ActiveRecord models extend `MyActiveRecord` instead of `CActiveRecord`. The only caveat is that you need to tell Yii about the existence of your class, first.

Yii can (obviously) recognize any class defined within the framework itself. Any other class your application uses must be imported into the framework in order to be usable. There are two ways of doing so.

First, to import a class on a global scale–so that every application element has access to it, use the configuration file:

```
# protected/config/main.php
// Other stuff.
'import'=>array(
    'application.models.*',
    'application.components.*',
),
// More other stuff.
```

That code, from the standard configuration, automatically imports every class found within the **protected/models** and **protected/components** directories.

A global import is practical when a class might be used in multiple places. When that's not the case, import the specific class (or classes) when needed using:

```
Yii::import('path/to/classfile');
```

MORE EXAMPLES

With the fundamental concepts of exceptions explained, and a couple of simple examples in place, it's time to look at some more specific and real-world examples. Traditionally, as a writer, I always come up with my own ideas for every example in a book. However, in this case, I'm going to use a combination of new examples, as well as analysis of existing extensions (with due credit given, of course). I'm using existing extensions for a couple of reasons:

1. Lots of great extensions have already been created, and spending time coming up with second-rate examples just for the sake of being new leaves a lot to be desired.
2. This approach shows how you can learn to write your own extensions by looking at existing extensions, now that you know what to look for.

If, when all is said and done with this book, you'd like another example of a specific extension type, let me know and I'll see what I can do.

Creating Behaviors

Earlier I mentioned the timestamp behavior as a useful concept. This behavior, part of the Zii library and written by Jonah Turnquist, can automatically set the value of model attributes to the current timestamp. The extension is mostly defined like so (some content has been removed for brevity and clarity):

```
<?php
class CTimestampBehavior extends CActiveRecordBehavior {
    public $createAttribute = 'create_time';
    public $updateAttribute = 'update_time';
    public $setUpdateOnCreate = false;
    public $timestampExpression;
    public function beforeSave($event) {
        if ($this->getOwner()->getIsNewRecord() &&
        ($this->createAttribute !== null)) {
            $this->getOwner()->{$this->createAttribute} =
            $this->getTimestampByAttribute($this->createAttribute);
        }
        if ((!$this->getOwner()->getIsNewRecord()
        || $this->setUpdateOnCreate)
        && ($this->updateAttribute !== null)) {
```

```
            $this->getOwner()->{$this->updateAttribute} =
                $this->getTimestampByAttribute($this->updateAttribute);
        }
    }
}
```

As you can see, the class extends `CActiveRecordBehavior`, as the behavior is meant to be applied to models that extend `CActiveRecord`. The class has five public attributes, making these configurable options when using the behavior:

```
public function behaviors() {
    return array(
        'CTimestampBehavior' => array(
            'class' => 'zii.behaviors.CTimestampBehavior',
            'createAttribute' => 'create_time_attribute',
            'updateAttribute' => 'update_time_attribute',
        )
    );
}
```

Being a behavior, the most important code is defined within the `beforeSave()` method. Within that method, the model instance that triggered the behavior is available through `$this->getOwner()`. The code is a bit complicated from there (for maximum flexibility), but it simply updates the appropriate attribute of that model, varying whether this is a new record or not, and whether the "setUpdateOnCreate" option is true or not.

As another example, in Chapter 9, "Working with Forms," I discussed how to work with model attributes associated with checkboxes. In particular, you might have an attribute whose value stored in the database is Y/N, which will need to be properly represented by a form checkbox. The catch being when a checkbox is not checked in the form, the resulting value will be 0 (by default). To address this problem, you could make a behavior extension. I'll intermingle the code and the explanation:

```
<?php

class LUCheckboxBehavior extends CActiveRecordBehavior {
        public $trueValue = 'Y';
        public $falseValue = 'N';
        public $attr = null;
```

The class extends `CActiveRecordBehavior`, of course. Within the class are three public properties, making it configurable. Two properties set the "true" and "false" values as they're stored in the database. The defaults are Y and N. The third

property is the attribute to which the behavior should be applied. The behavior would be used like so:

```
public function behaviors() {
    return array(
        'LUCheckoutBehavior' => array(
            'class' => 'ext.lucheckoutbehavior.LUCheckoutBehavior',
            'trueValue' => 'Yes',
            'falseValue' => 'No',
            'attr' => 'receiveEmails'
        )
    );
}
```

Next, the behavior has a `beforeSave()` method. Its role is to convert the form data from the default 1/0 to those expected by the database:

```
public function beforeSave($event) {
    $model = $this->getOwner();

    if (($this->attr === null) ||
    !in_array($this->attr, get_object_vars($model)) ) {
        throw new CException('You must indicate a valid model
        attribute!');
    }

    if ($model->{$this->attr} == 1) {
        $model->{$this->attr} = $this->trueValue;
    } else {
        $model->{$this->attr} = $this->falseValue;
    }
}
```

First, the method gets a reference to the model instance. Next, the method verifies that an attribute value was provided and that it exists within the model instance. If not, an exception is thrown. Finally, the method converts the values from 1/0 to whatever the user wants.

The behavior also needs to perform a conversion when the record is retrieved (so that the form works properly). That code goes in an `afterFind()` method, and is similar to that in the other method:

```
public function afterFind($event) {
    $model = $this->getOwner();
    if (($this->attr === null) ||
```

```
    !in_array($this->attr, get_object_vars($model)) ) {
        throw new CException('You must indicate a valid model
        attribute!');
    }
    if ($model->{$this->attr} == $this->trueValue) {
        $model->{$this->attr} = 1;
    } else {
        $model->{$this->attr} = 0;
    }
}
```

The only difference in this method is that it checks if the existing values match the "true" and "false" values, and assigns 1 or 0 accordingly.

And there you have a functional behavior extension that's easy to use and quite helpful. Once applied to a model, you'll never have to think about converting between database and checkbox form values again!

> *{NOTE}* If you're cutting and pasting this code, you also need a closing bracket to complete the class.

Creating Widgets

For the next example, let's create a widget, one of the most common extension types around. From a development perspective, creating a widget requires a decent amount of knowledge and effort, although not so much as a full-blown module. Widgets are used to provide more complex logic, normally including HTML, as a component that's separate from your own view files.

Instead of trying to come up with a perfect example, that doesn't yet exist, I've decided upon a somewhat decent example that I can use to help explain the general process of creating a widget. The specific example is a widget-based implementation of the Stripe Checkout form.

In case you're not familiar with Stripe or how its system works, you use a form on your Web site for taking the customer's payment information (i.e., credit card details). This data is sent to Stripe via Ajax, so that it never touches your server. Not having the credit card information on your server relieves almost all of the PCI compliance burden.

Stripe takes the customer's credit card information, stores it on its system, and returns a representative token to your site. The form on your site is then submitted to your server (so your server receives the token and any other form data, but not the payment details). The form handling script on your server then actually processes the charge, using the supplied token to represent the card to be used.

Checkout is Stripe's easy and convenient modal form for taking a customer's payment information and securely sending it to Stripe. You don't need to create the form itself or write any JavaScript. Although you can do those things manually (and commonly will), Checkout is a no-brainer implementation (**Figure 19.3**).

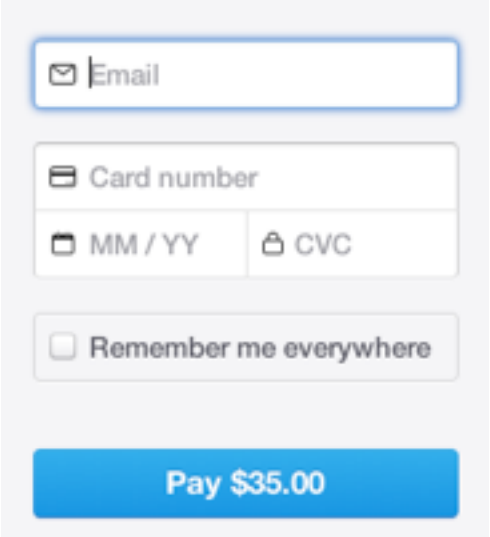

Figure 1.3: *Part of the Checkout modal window*

There are two caveats to this widget example. First, as of the summer of 2013, I work for Stripe (as a Support Engineer). But I work at Stripe because I'm such a huge fan of their product, not the other way around.

The second caveat is that this widget only does half of the task: the part that takes the customer's payment information, passes it to Stripe, receives a token in return, and submits it to your server. To complete the implementation of Stripe Checkout, you would need to:

- Download and install the Stripe PHP library
- Process a charge in your form handling code

That being said, what this widget does do it does just fine, and I think it explains the process of creating widgets well enough. Further, this widget example will make the fuller Stripe example (as a module) a bit easier to understand when that time comes later in the chapter.

Looking at the End Result

To create this widget, I begin by identifying the desired end result. In the case of Stripe Checkout, the end result will be something like:

```
<form action="/charge" method="POST">
  <script
    src="https://checkout.stripe.com/checkout.js" class="stripe-button"
    data-key="pk_test_APr32Tly9WH6K9XfZpJeEKCH"
    data-image="/square-image.png"
    data-name="Demo Site"
    data-description="2 widgets ($20.00)"
    data-amount="2000">
  </script>
</form>
```

That's from the Stripe documentation. The Checkout JavaScript file is hosted on Stripe's servers, so the widget doesn't need to worry about that.

The questions I must now ask are:

- What information must the widget user provide?
- What information should be optionally configurable?

In the above, the form's "action" attribute is mandatory, although it would make sense to default this to the current page. In terms of Stripe, the "key" value (which identifies your account to Stripe) is required. The other "data-*" items are optional, along with a couple more mentioned in the documentation.

The combination of the required and optional information will be the widget's properties, configurable on a widget-by-widget basis.

Defining the Class

Next, it's time to create the widget class. It must extend `CWidget`:

```
<?php
class LUStripeCheckout extends CWidget {
    public $action;
    public $key;
    public $amount;
    public $description;
    public $name;
    public $currency = 'usd';
    public $panelLabel;
    public $billingAddress = false;
    public $shippingAddress = false;
    public $email;
    public $label = 'Pay with Card';
    public $image;
```

As you can see, there are a slew of public properties. A couple of them have default values, primarily the default values used by Stripe Checkout already.

In a real-world (i.e., non-book) class, there would be plenty of documentation included, starting with phpDoc syntax for the class itself and the properties. I would also include an example of the widget's usage in the documentation.

Next, the widget must have two methods: `init()` and `run()`. Here's the first of those:

```
public function init() {
    if ($this->key == null) {
        throw new CException('You must provide your public
        Stripe API key.');
    }
    if (($this->amount == null) ||
    !is_integer($this->amount)) {
        throw new CException('You must provide an amount as
        an integer in cents.');
    }
    if ($this->action == null) {
        $this->action = Yii::app()->request->url;
    }
    if (!Yii::app()->getRequest()->isSecureConnection) {
        throw new CException('This form must be loaded over
        HTTPS.');
    }
}
```

In this widget, the role of the `init()` method is to perform the necessary validation. To start, the method confirms that a key was provided. If not, an exception is thrown (**Figure 19.4**).

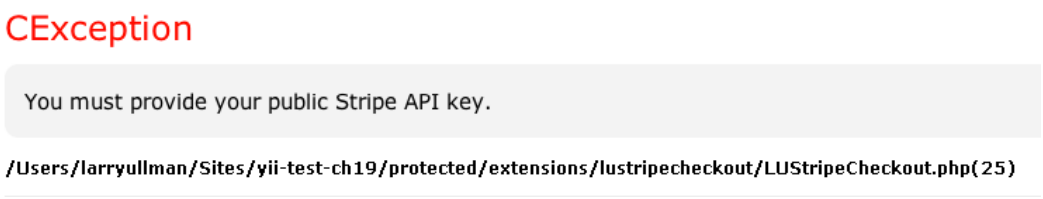

Figure 1.4: *A thrown exception.*

Next, the amount is not required, but ought to be used, so it's validated. (To be clear, the amount is only required on the server-side.) More importantly, Stripe requires that the amount be provided as an integer in cents, so it makes sense (ha!) to enforce that here (**Figure 19.5**).

The action value is required, but instead of throwing an exception, if one is not provided, the current page will be used.

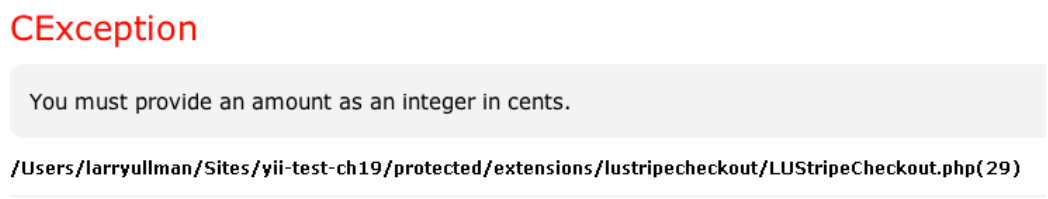

Figure 1.5: *Another exception.*

Finally, the Checkout form must be loaded over HTTPS in order to be PCI compliant. The method makes a last check for that.

The other method, `run()` has nothing to do other than to render the view file:

```
    public function run() {
        $this->render('form');
    }
} // End of widget class.
```

Now it's time to define that view file.

Defining the View File

The view file, named **form.php**, should be placed within the **views** directory of the widget extension's directory. It might look like this:

```
<form action="<?php echo CHtml::encode($this->action); ?>" method="POST">
  <script
    src="https://checkout.stripe.com/checkout.js"
    class="stripe-button"
    data-key="<?php echo CHtml::encode($this->key); ?>"
<?php if (!empty($this->image)) echo ' data-image="' .
    CHtml::encode($this->image) . '"'; ?>
<?php if (!empty($this->name)) echo ' data-name="' .
    CHtml::encode($this->name) . '"'; ?>
<?php if (!empty($this->description)) echo ' data-
    description="' . CHtml::encode($this->description) .
    '"'; ?>
<?php if (!empty($this->amount)) echo ' data-amount="' .
    CHtml::encode($this->amount) . '"'; ?>
<?php if (!empty($this->currency)) echo ' data-currency="' .
    CHtml::encode($this->currency) . '"'; ?>
<?php if (!empty($this->panelLabel)) echo ' data-panel-
    label="' . CHtml::encode($this->panelLabel) . '"'; ?>
<?php if (!empty($this->billingAddress)) echo ' data    billing-address="' .
```

```
    CHtml::encode($this->billingAddress) .
    '"'; ?>
<?php if (!empty($this->shippingAddress)) echo ' data-
    shipping-address="' .
    CHtml::encode($this->shippingAddress) . '"'; ?>
<?php if (!empty($this->email)) echo ' data-email="' .
    CHtml::encode($this->email) . '"'; ?>
<?php if (!empty($this->label)) echo ' data-label="' .
    CHtml::encode($this->label) . '"'; ?>
    ></script>
</form>
```

The two required pieces of information–the action and the key–are printed automatically. Everything else is printed only if its value is not empty (**Figures 19.6** and **19.7**). For security, everything value is run through `CHtml::encode()`.

```
<form action="/yii-test-ch19/index.php/site/test" method="POST">
  <script
    src="https://checkout.stripe.com/checkout.js" class="stripe-button"
    data-key="pk_test_APr32Tly9WH6K9XfZpJeEKCH"
 data-amount="3500" data-currency="usd" data-label="Pay with Card"  ></script>
</form></div><!-- content -->
```

Figure 1.6: *The bare minimum source.*

```
<form action="/yii-test-ch19/index.php/site/test" method="POST">
  <script
    src="https://checkout.stripe.com/checkout.js" class="stripe-button"
    data-key="pk_test_APr32Tly9WH6K9XfZpJeEKCH"
 data-amount="2299"
 data-currency="usd"
 data-billing-address="1"
 data-shipping-address="1"
 data-label="Pay with Card"
    ></script>
</form></div><!-- content -->
```

Figure 1.7: *Different source based upon the provided values.*

Using the Widget

Finally, there's the use of the widget. Assuming you've provided good documentation, using the widget should be very easy. This would go in one of the site's view files:

```
<?php $this->widget('ext.lustripecheckout.LUStripeCheckout',
    array(
    'shippingAddress' => true,
```

```
    'billingAddress' => true,
    'key' => 'pk_test_APr32Tly9WH6K9XfZpJeEKCH',
    'amount' => 2299
)); ?>
```

Any other public property can be set in the configuration. Failure to meet the minimum requirements results in an exception being thrown. **Figure 19.8** shows the result of the above code (after clicking the "Pay with Card" button that's also generated).

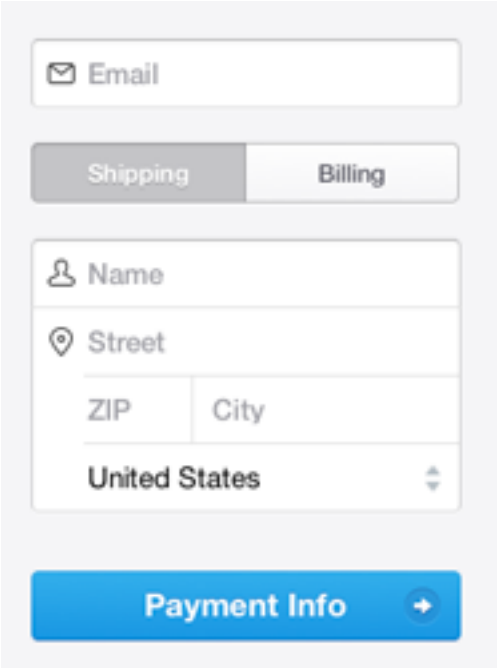

Figure 1.8: *A custom Checkout integration.*

If you'd like to see what the page receives upon successful checkout, fill out the form using sample data, then dump out the value of `$_POST`.

WORKING WITH MODULES In a way, modules are the easiest extension type to understand. A module is an entire Web application, used as a component of a larger application. A module will have the same core pieces as any application–models, views, and controllers, but be particular to one aspect of a site. For example, your site may have:

- A main area
- Support forums
- A shop
- A blog
- An administrative area

In this example, the main area of your site would be the primary Yii application. The other three areas could each be their own module. The result is a fairly complex and sophisticated site that's still easy to use. Moreover, because you'll have created modules for three of these areas, those modules could be turned into reusable extensions for other projects you do.

The only thing you can't do with a module is deploy it on its own. This is really the primary distinction between a module and an application: modules are always subservient to an application.

To best explain how to create and use modules, let's first walk through the basics and then create a specific example as a module.

Module Basics

One of the nice things about creating modules in Yii is that the Gii tool will create the shell of the module for you (just as it does the entire application).

Creating a Module The first thing you must do, before heading to Gii, is make sure you have a **protected/modules** directory that's writable by the Web server. This is where Gii will attempt to create the module.

Once you've created that,

1. Enable Gii, if it's not already.
2. Access Gii for your site in your browser.
3. Click Module Generator.
4. On the resulting page, enter the name of the module to be generated, and click Preview.
5. On the resulting page, confirm the files being created, and click generate.

This is all pretty straightforward. The most common complication will be the lack of an existing **modules** directory, which can result in odd errors.

Give serious thought to your module name prior to following those steps. Modules need unique names, which will be their identifier. To avoid bugs, it's best to name the module in all lowercase letters, and do not use any numbers, punctuation, or other symbols.

Gii will create a new directory, **protected/modules/modulename**. Within it, you'll find:

- **controllers**
- **controllers/DefaultController.php**
- **views**
- **controllers/default**

- **controllers/default/index.php**
- **ModulenameModule.php**

This last item will be the main module class, which will extend `CWebModule`. In many ways, this class is a corollary to the application's `CApplication` class.

Using a Module Once you've defined a module, or added an existing module extension to your site, you just need to tell your site about the module in order to use it. This is done in the "modules" section of the primary configuration file. In fact, you've already seen an example of this with Gii:

```
# protected/config/main.php
'modules'=>array(
    'gii'=>array(
        'class'=>'system.gii.GiiModule',
        'password'=>'password',
        'ipFilters'=>array('127.0.0.1','::1'),
    ),
),
```

To use a different module, just add its name to the configuration file:

```
# protected/config/main.php
'modules'=>array(
    'modulename',
    'gii'=>array(
        'class'=>'system.gii.GiiModule',
        'password'=>'password',
        'ipFilters'=>array('127.0.0.1','::1'),
    ),
),
```

The value used needs to match the ID value of the module itself (which also corresponds to the name of the module's directory).

With the module enabled via the configuration file, it's now usable by your site. You can test this by heading to **http://www.example.com/index.php/modulename** in your browser.

As with the primary application, if no controller or action IDs are provided, the module will run the default action of the default controller. With a module, that's the "index" action of `DefaultController`. As generated, this action merely renders the **modulename/views/default/index.php** view file (**Figure 19.9**).

Figure 1.9: *A default module layout.*

```
# protected/modules/modulename/controllers/DefaultController.php
<?php
class DefaultController extends Controller {
    public function actionIndex() {
        $this->render('index');
    }
}
```

As you can see, this controller extends the `Controller` class (which, in turn, extends `CController`), so it can be used and behaves like any other controller in your application. You can now also create other controllers, other actions, and other view files. If your module needs models, you can create and use those, too. You'll see more of this in a forthcoming example.

> *{TIP}* You can nest one modules within another by adding the submodule to the configuration of the parent module, and placing the submodule in the **modules** directory of the parent module.

Routing Modules When using modules, an additional dimension is added to your URLs. The full syntax of a URL in Yii is /index.php////.

If no action ID is provided, the default action of the requested controller is used. If no controller ID is provided, the default controller of the application or module is used. If no module ID is provided, the default application is used.

When it comes to creating URLs with modules, you'll still use the `createUrl()` or `createAbsoluteUrl()` methods of the `CController` class. As is the case when not using modules, if you specify the action and nothing else, the route will assume the same controller, and, in the case of a module, module:

```
# protected/modulename/controllers/DefaultController.php
public function actionDummy() {
    // modulename/default/index:
    $url = $this->createUrl('index');
```

If you specify the controller, the URL will be to that controller and action, but still within the same module:

```
# protected/modulename/controllers/DefaultController.php
public function actionDummy() {
    // modulename/other/index:
    $url = $this->createUrl('other/index');
```

If you specify the module name, you can create a URL to a different module within the application:

```
# protected/modulename/controllers/DefaultController.php
public function actionDummy() {
    // testmodule/other/index:
    $url = $this->createUrl('/test/other/index');
```

Note that in this case, you need to start with a slash, to indicate the base of the URL. The same goes if you want to create a URL to the primary application from within a module:

```
# protected/modulename/controllers/DefaultController.php
public function actionDummy() {
    // /other/index:
    $url = $this->createUrl('/other/index');
```

Configuring the Module Modules, like any application component or the application itself, can be pre-configured in the primary configuration file. To do that, assign a value to any public property of the main module class:

```
<?php
class TestModule extends CWebModule {
    public $var;
```

```
# protected/config/main.php
'modules'=>array(
    'test' = array(
        'var' => 'value',
    ),
),
```

{TIP} If you'd rather, you can create your own configuration file for the module, and then have the main configuration file include it.

Because the base class extends `CWebModule`, you can configure any public property of that class, too:

```
# protected/config/main.php
'modules'=>array(
    'test' = array(
        'defaultController' => 'something',
        'layout' => 'home'
    ),
),
```

Within the module or the application, you can also access any public property through the module itself. First, you'd get access to the module, which returns a class instance. Then you could access its properties:

```
$m = Yii::app()->getModule('moduleID');
echo $m->var;
```

Within a module itself, you can always refer to the current module using `Yii::app()->controller->module`:

```
$m = Yii::app()->controller->module;
echo $m->var;
```

Further, Yii will create a root alias for each module. Having created the "test" module, in path names in your application, "test" will equate to **protected/modules/test**.

An Example Module

To wrap up this discussion of modules, I'm going to write a more complete version of the Stripe extension begun earlier. This was originally an incomplete widget example, that would be better developed as a module.

Stripe doesn't offer shopping carts or inventory management, it simply provides an easy and secure way to process a credit card payment. As explained earlier, it's a two-step process:

- In the client, take the customer's payment information, pass it to Stripe, and receive a token in return
- On the server, use the token to process the charge

Stripe's Checkout functionality handles all of step one, as already demonstrated. Now, let's extend (ha!) that functionality so that the charge is actually processed. Because the processing of the payment is separate from the other aspects of the e-commerce site, the only dynamic piece of information this module would need is the amount to be charged.

To make this module more complex, configurable, and interesting, I'm tossing in a few more features:

- A custom payment form, in lieu of Stripe Checkout
- Recording of the charges in a database

With all this in mind, let's create and use this module.

> *{TIP}* You'll probably find this easier to follow if you also download the code from [http://yii.larryullman.com].

Generating the Shell The first thing you'll need to do is enable Gii, and follow the steps outlined earlier to create the shell of the module. This will require that you have a writable **protected/modules** directory, if you don't already.

For this module, I'm going to call it "pay"(**Figure 19.10**).

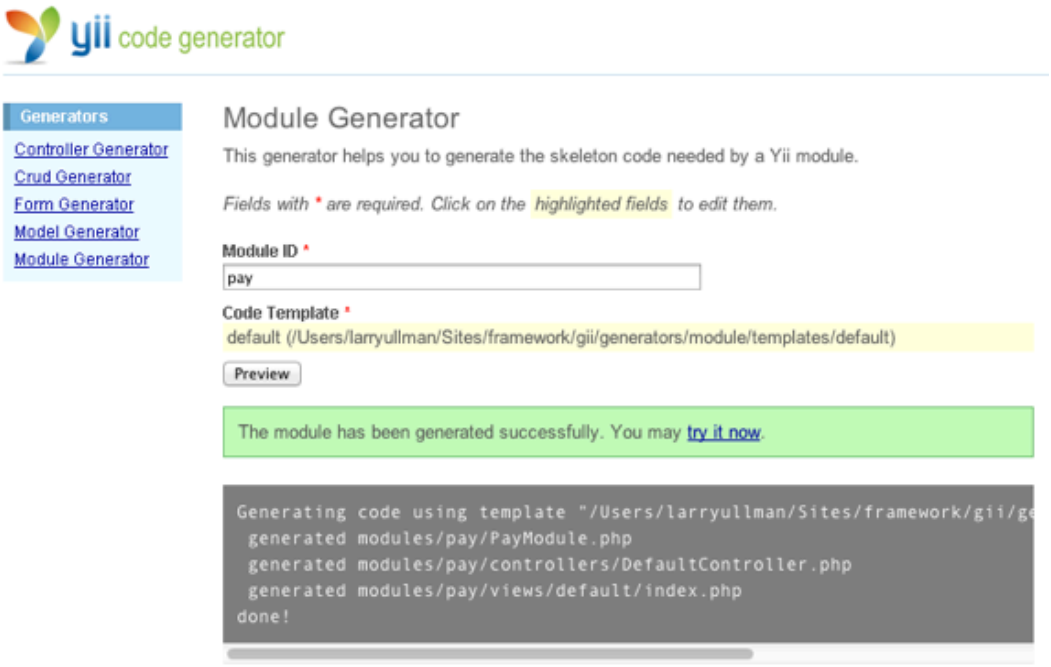

Figure 1.10: *Creating a new module.*

The result is the **protected/modules/pay** directory. Within it, you'll find:

- **controllers**
- **controllers/DefaultController.php**
- **views**
- **controllers/default**
- **controllers/default/index.php**
- **PayModule.php**

Now it's time to edit, and add to, those files and directories.

Adding the Stripe Library Performing a charge requires the Stripe PHP library, so the extension needs to have that.

Commonly, third-party libraries go within a **vendors** directory, so start by creating **protected/modules/pay/vendors**. Within **vendors**, create a **stripe** folder.

Next, head to https://stripe.com/docs/libraries and download the PHP library.

Expand the downloaded file. The result will be a folder named something like *stripe-php-1.10.1*. Within that, you'll find:

- **CHANGELOG**
- **composer.json**
- **lib**
- **LICENSE**
- **README.rdoc**
- **test**
- **VERSION**

Copy the **lib** folder to **protected/modules/pay/vendors/stripe**.

There! Now the extension has its required external libraries.

Updating the Module Class Next, I'm going to think about how I might want the module to be configured when added to a site. With this particular module, there are two pieces of information that must be provided: the user's public and private Stripe keys. These can be provided during the configuration so long as they're public attributes of the main module class:

```
class PayModule extends CWebModule {
    public $public_key;
    public $private_key;
```

Because these values are required (no default value would work), the class's `init()` method should throw exceptions if they're not provided (**Figure 19.11**):

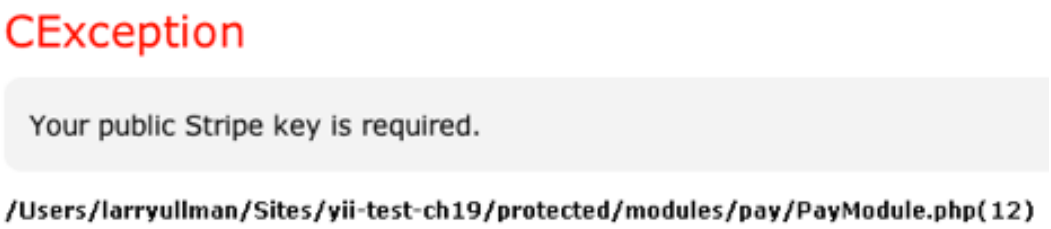

Figure 1.11: *Stripe keys are required.*

```
public function init() {
    if ($this->public_key == null) {
        throw new CException('Your public Stripe key is
required.');
    }
    if ($this->private_key == null) {
        throw new CException('Your private Stripe key is
required.');
    }
}
```

Creating the Database Table The Stripe extension module will need one model class defined. The class will be used to create the form, add validation, and store the data in the database. For that reason, a model based upon Active Record makes sense. Gii can create the model, but the underlying database table needs to exist first. I imagine the database table would be defined like so:

```
CREATE TABLE payment (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    charge_id VARCHAR(60) NOT NULL,
    email VARCHAR(80) NOT NULL,
    amount INT UNSIGNED NOT NULL,
    date_added TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (charge_id),
    INDEX (email)
) ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
COLLATE = utf8_general_ci
```

> *{NOTE}* You may notice that the table is not storing any credit card details. When using Stripe, absolutely no credit card information will touch your server.

You *could* define that SQL command in a text file that you distribute with the extension. But since reusability is the goal, and to do things the Yii way, let's

create a *migration* for it.

The first thing you'll want to do is setup your application to support migrations, if you have not already. See Chapter 18, "Advanced Database Issues", for those explicit instructions.

Next, from the command-line, within the **protected** directory, execute this command (**Figure 19.12**):

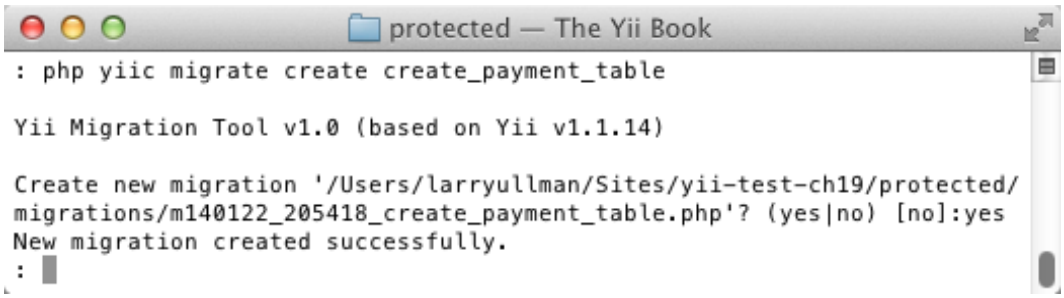

Figure 1.12: *Creating a migration.*

```
yiic migrate create create_payment_table
```

> *{NOTE}* You may need to change the command used to execute `yiic` to suit your environment.

This will create a file named something like *m140122_205418_create_payment_table*, and stored in the **protected/migrations** directory.

Since this migration isn't particular to the application–it's part of the module, create a **migrations** directory within your module folder, and then move this new file there. Then open the file for editing.

The migration class has two methods: `up()`, for implementing the migration, and `down()`, for undoing its impact. In this case, the `up()` method should create the `payment` table and `down` should delete it:

```
public function up() {
    $this->createTable('payment', array(
        'id' => 'pk',
        'charge_id' => 'string NOT NULL',
        'email' => 'string NOT NULL',
        'amount' => 'integer UNSIGNED NOT NULL',
        'date_added' => 'timestamp NOT NULL DEFAULT
            CURRENT_TIMESTAMP'
    ));
    $this->createIndex('charge_id', 'payment', 'charge_id', true);
```

```
    $this->createIndex('email', 'payment', 'email');
    echo "The 'payment' table has been created.\n";
}
public function down() {
    $this->dropTable('payment');
    echo "The 'payment' table has been dropped.\n";
    return false;
}
```

With that file defined, you now just need to execute the migration. Normal application migrations are executed from the command-line (from within the **protected** directory) using:

```
yiic migrate
```

By default, this command looks for migrations to be executed found within the **protected/migrations** folder. To change that, provide the path to the module's **migrations** directory (**Figure 19.13**):

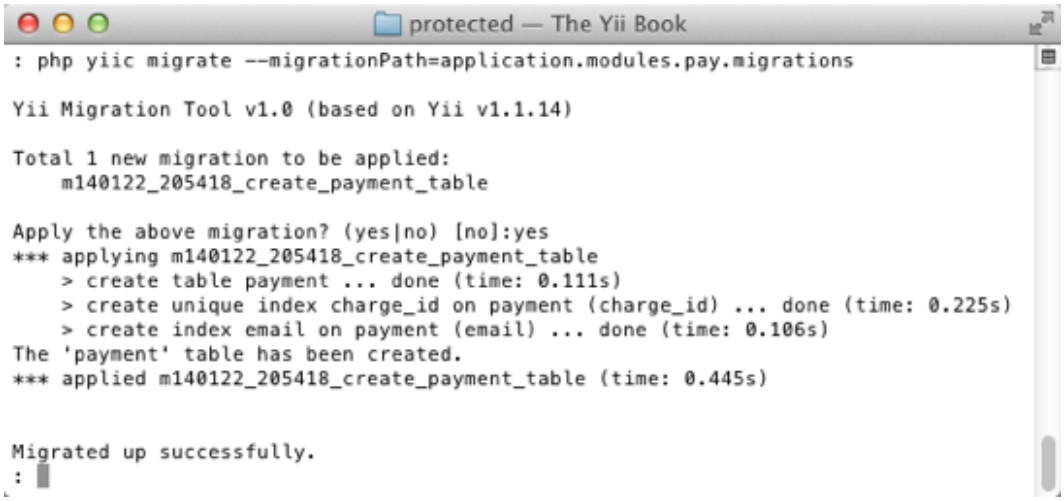

Figure 1.13: *Implementing the migration.*

```
yiic migrate --migrationPath=application.modules.pay.migrations
```

And now the database table should be created! Next, you can generate and edit the corresponding model. More importantly, you've just made your extension that much easier for others to use!

Creating the Model For the model, start by having Gii generate it based upon the database table definition:

1. Log into Gii.
2. For the table name, use "payment".
3. For the model class, use "Payment" (the default).
4. Click Preview.
5. Assuming everything is okay, click Generate (**Figure 19.14**).

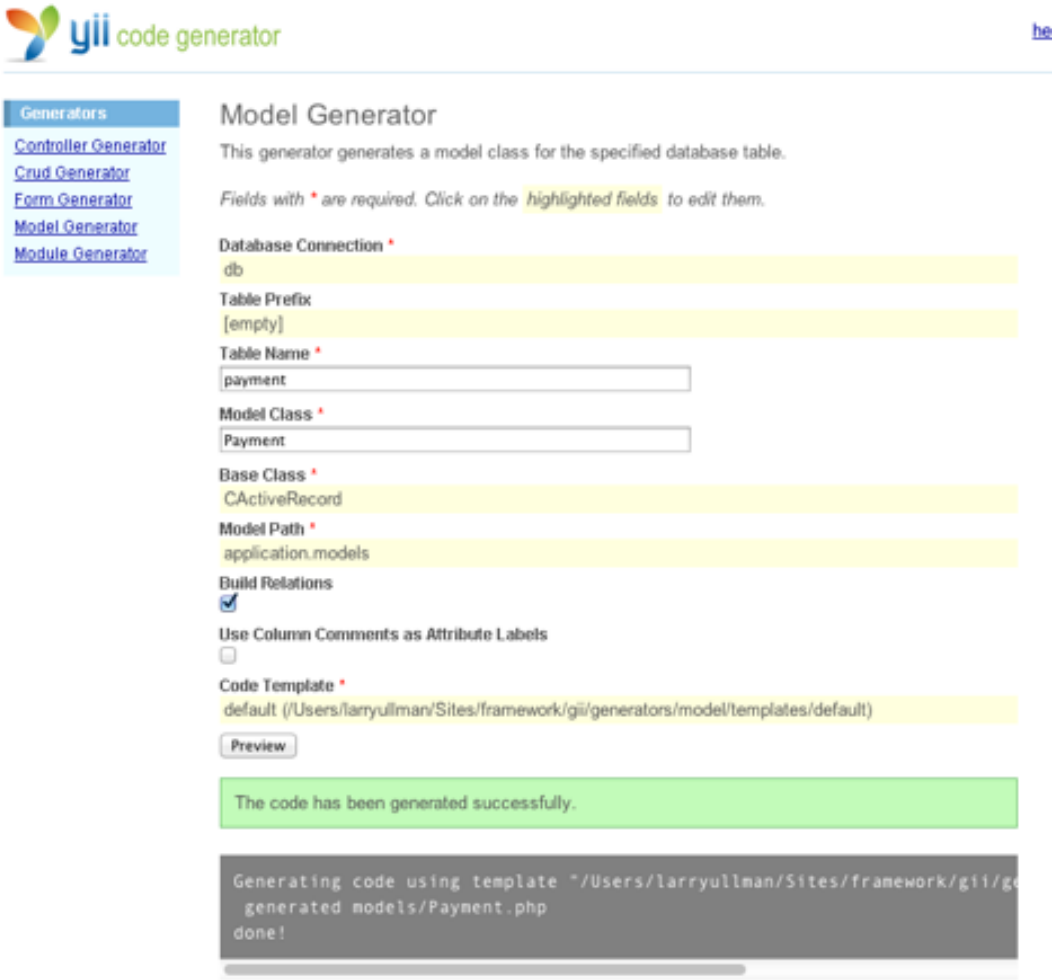

Figure 1.14: *Creating a new model.*

The resulting code will end up in **protected/models/Payment.php**. Create a **models** directory for your extension (in the extension's folder), and move this file there. Open the file to edit it.

First, you'll want to add one public property to the model:

```
class Payment extends CActiveRecord {
    public $token;
```

The token is the only additional piece of information that is required through the HTML form, but won't be stored in the database. For this reason, it becomes a public property.

None of the credit card information–number, CVC, or expiration date–will touch the server, so there's little point in representing those in the model.

Now you'll want to tweak the validation rules slightly:

```
public function rules() {
    return array(
        array('charge_id, email, amount, token',
        'required'),
        array('charge_id, email', 'length', 'max'=>255),
        array('email', 'email'),
        array('charge_id', 'unique'),
        array('amount', 'numerical', 'integerOnly'=>true,
        'min'=>50),
        array('charge_id, email, amount, date_added',
        'safe', 'on'=>'search'),
    );
}
```

Hopefully there's nothing too surprising here, but see Chapter 5 for more on validation rules, if need be. The token is marked as required here. The amount must be an integer at least 50 or larger, which corresponds to the minimum amount that can be processed through Stripe.

You might want to update the labels, too:

```
public function attributeLabels()
{
    return array(
        'id' => 'ID',
        'charge_id' => 'Charge ID',
        'email' => 'Email',
        'amount' => 'Amount',
        'date_added' => 'Date Charged',
    );
}
```

{TIP} You may also want to set custom error messages.

And, it's not a big deal, but I would remove `id` from being a search criteria. I only put that property in the model because I prefer my database tables to be numerically indexed, rather than using the `charge_id`.

Editing the Controller With the model created and edited, you can turn to the controller (and then the view files). By default, modules use the "index" action of the `DefaultController`. Although I might be inclined to use more meaningful terminology, changing these defaults just for that purpose is moot.

The workflow for the extension is pretty simple:

1. The payment form should be shown.
2. Upon successful completion of the payment form, the charge attempt needs to be made of Stripe.
3. If the charge attempt succeeds, the user should be shown a "thanks" page.

All of this can be done in two actions and two view files, although one of those view files will require some heavy JavaScript. First, though, the display of the form should only be loaded over HTTPS, so a filter can be added for that:

```
# pay/controllers/DefaultController.php
public function filters() {
    return array(
        'httpsOnly + index',
    );
}
public function filterHttpsOnly($fc) {
    if (Yii::app()->getRequest()->getIsSecureConnection()) {
        $fc->run();
    } else {
        throw new CHttpException(400, 'This page needs to be
        accessed securely.');
    }
}
```

The first chunk of code applies the "httpsOnly" filter to the "index" action. Next, the "httpsOnly" filter is defined as part of the controller. For an even better integration, you could create "httpsOnly" as a filter extension, include it with the project, and then use it here.

> *{TIP}* If you're testing this and don't have SSL setup, remove the use of the filter.

The rest of the controller has two actions: "index" and "thanks". The "index" action does the bulk of the work, and I'll build it up in two sections of this chapter. All of the following code (until I say otherwise), goes, in order, within the "index" action, starting by creating a model instance:

```
$model=new Payment;
```

Most of this action's code is similar to that you'd find in any controller's "create" action. Note that you can create an instance of a `Payment` model here because the primary module class imports classes from the module's **models** directory:

```
# protected/modules/pay/PayModule.php
class PayModule extends CWebModule {
    public function init() {
        $this->setImport(array(
            'pay.models.*',
            'pay.components.*',
        ));
    }
```

Returning to the controller, next, you'll need to validate that an amount was provided:

```
$model->amount = 12575;
```

After creating the model instance, it should access the amount to be charged. On a real, live site, this amount would most likely be determined dynamically, and would be provided to the extension in some manner:

- Saved in the session
- Calculated by another controller
- Retrieved from the database

For this demonstration, I'm just hardcoding a value there. Remember that it has to be an integer, representing the amount in cents!

Next, the action checks if the form has been submitted. If so, the safe attributes are assigned. Then the `validate()` method is called to confirm that all the data matches the validation rules. Note that `save()` isn't called here yet, as the model is only saved after successfully making the charge.

```
if(isset($_POST['Payment'])) {
    $model->attributes=$_POST['Payment'];
    // Temporary:
    $model->charge_id = 'temp';
    if($model->validate()) {
```

In the model, `charge_id` is required, but this value will only be provided once the charge is processed with Stripe. To circumvent this issue, but still pass validation, the `charge_id` is assigned a temporary value first.

Next, it's time to process the charge:

```
// Charge via Stripe!
if (/* charged */) {
    $model->save();
    $this->redirect(array(
        'thanks',
        'amount' => $model->amount)
    );
}
```

If the payment went through, then the model is saved, and the user is redirected to the "thanks" page, passing along the amount in the URL. (This code goes within the `if ($model->validate())` conditional.)

Finally, the action renders the "form" page, passing along the model instance and the public Stripe key:

```
    } // Not validated.
} // Not posted.
// Show the form:
$this->render('form',array(
    'model'=>$model,
    'key' => Yii::app()->controller->module->public_key
));
```

As with any create-like action, the code will only get to this point if the user was not redirected to the "thanks" page. That would happen in three scenarios:

- The form is loaded for the first time
- The form is not completely filled out and errors need to be shown
- The payment could not be made with Stripe

The code above handles all of this except for the payment attempt, to be added later.

Finally, you'll need the "thanks" action. It receives the amount charged in the URL and passes it to the view file:

```
public function actionThanks($amount) {
    $this->render('thanks',array(
        'amount'=>$amount,
    ));
}
```

That's the bulk of the controller (lacking only the most important piece!). Fleshing out this example completely, I would be inclined to add an admin feature to this extension, allowing the administrator to view the list of payments. Neither "update" nor "delete" actions would ever be warranted.

> *{TIP}* Stripe supports the creation of customers and recurring billing via subscriptions. Ability to do both would be a great addition to this extension.

Creating the View Files With the bulk of the controller in place, let's create the two view files. The "thanks" page is easy, something like:

```
# protected/modules/pay/views/default/thanks.php
<?php
/* @var $this DefaultController */

$this->breadcrumbs=array(
    $this->module->id,
);
?>
<h1>Thank you for your order!</h1>

<p>You have been charged $<?php echo number_format($amount/100, 2); ?>.</p>
```

The "form" page is much more complicated. It must display an HTML form and apply some JavaScript. And because the functionality of Stripe depends upon the JavaScript, it's a bit more complicated than most uses.

To create the form, I started with the standard syntax for forms in Yii, like those created by Gii:

```
<h1>Pay</h1>
<div class="form">
<?php $form=$this->beginWidget('CActiveForm', array(
    'id'=>'pay-form',
    'enableAjaxValidation'=>false,
)); ?>
<p class="note">Fields with <span class="required">*</span> are required.</p>
<?php echo $form->errorSummary($model); ?>
<div class="row">
    <?php echo $form->labelEx($model,'email'); ?>
    <?php echo $form->textField($model,'email',
        array('size'=>60,'maxlength'=>60)); ?>
    <?php echo $form->error($model,'email'); ?>
</div>
```

Ajax validation should definitely be disabled, as the form's data should not be sent to the server. And, most of the form's data–specifically, the credit card information–isn't represented by the model anyway. Speaking of which…

```
<div class="row">
    <label for="cc-num" class="required">Credit Card Number
    <span class="required">*</span></label>
    <input size="20" maxlength="20" id="cc-num" type="text"
    autocomplete="off">
</div>
<div class="row">
    <label class="required">Credit Card Expiration (MM/YYYY)
    <span class="required">*</span></label>
    <input size="2" maxlength="2" id="cc-exp-month" type="text"
    autocomplete="off"> / <input size="4" maxlength="4" id="cc-exp-year"
    type="text" autocomplete="off">
</div>
<div class="row">
    <label class="required">Credit Card CVC
    <span class="required">*</span></label>
    <input size="4" maxlength="4" id="cc-cvc" type="text"
    autocomplete="off">
</div>
```

For the credit card elements–the number, the expiration month and year, and the CVC–these are not tied to the model, and so they're created using straight HTML, not even `CHtml`. I made this choice for a few reasons:

1. Because they aren't tied to models, no server-side validation will be used, and no element-specific error messages will automatically apply.
2. It's absolutely imperative that the values entered here aren't submitted to the server (i.e., they cannot have `name` attributes).
3. The JavaScript will be easier if the ID values of the elements are known and constant.

Next, the form is completed:

```
    <div class="row buttons">
        <?php echo CHtml::submitButton('Pay $' .
        number_format($model->amount/100, 2),
        array('id'=>'submit-btn')); ?>
    </div>
<?php $this->endWidget(); ?>
</div><!-- form -->
```

For extra flair, the submit button shows the amount to be charged, which can be found in the model. Because it's an integer, it needs to be divided by 100 (**Figure 19.15**).

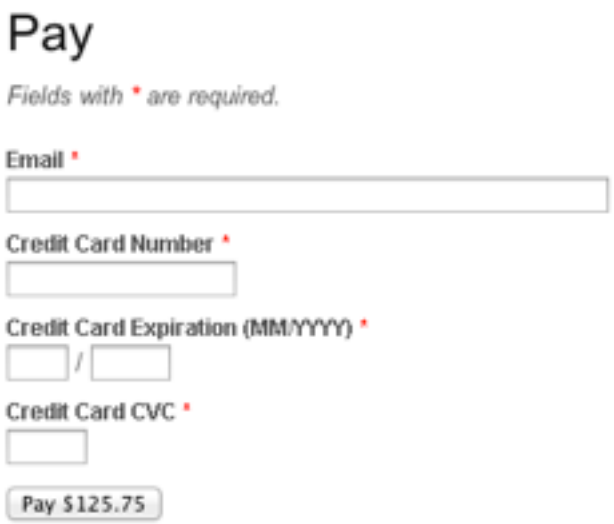

Figure 1.15: *The payment form.*

Now it's a matter of adding the JavaScript. First, the page needs to include the **Stripe.js** file:

```
<?php echo CHtml::scriptFile('https://js.stripe.com/v2/'); ?>
```

> *{TIP}* You can find more detailed explanations of much of this code in my Stripe blog series.

Next, the page needs to do a bunch of stuff:

- Set the Stripe public key
- Validate the payment information
- Send the payment information to Stripe
- Handle the Stripe response
- Report any errors

Arguably, most of this could go in an external JavaScript library that is then published to the **assets** folder. Instead, I'm just adding it to the page. All of the JavaScript goes within this:

```
<?php
Yii::app()->clientScript->registerScript('stripe', "
// All JavaScript here.
");
?>
```

Because all of the JavaScript is placed between double quotes, the JavaScript code has to use only single quotes, or escaped double quotes. I'll explain that code in pieces (again, the following all goes within the second `registerScript()` argument).

First, the Stripe public key is set:

```
Stripe.setPublishableKey('$key');
```

This is provided to the view file from the controller, and the main module class throws an exception if it doesn't exist.

The next bit of code handles the form submission. Within that code, the JavaScript must:

- Disable the submit button
- Get the four form values (for the credit card details)
- Validate the form values
- Pass the details to Stripe

Here's how the submit event handler is created, using jQuery:

```
$('#pay-form').submit(function(){
    // Prevent the form from submitting:
    return false;
}); // Submit function.
```

Now the other steps are performed within that bit (just before `return false`:

```
// Flag variable:
var error = false;

// disable the submit button to prevent repeated clicks:
$('#submit-btn').attr('disabled', 'disabled');

// Get the values:
var ccNum = $('#cc-num').val(), cvcNum = $('#cc-cvc').val(),
expMonth = $('#cc-exp-month').val(),
expYear = $('#cc-exp-year').val();
```

```
// Validate the number:
if (!Stripe.card.validateCardNumber(ccNum)) {
    error = true;
    reportError('The credit card number appears to be invalid.');
}

// Validate the CVC:
if (!Stripe.card.validateCVC(cvcNum)) {
    error = true;
    reportError('The CVC number appears to be invalid.');
}

// Validate the expiration:
if (!Stripe.card.validateExpiry(expMonth, expYear)) {
    error = true;
    reportError('The expiration date appears to be invalid.');
}

// Validate other form elements, if needed!

// Check for errors:
if (!error) {

    // Get the Stripe token:
    Stripe.card.createToken({
        number: ccNum,
        cvc: cvcNum,
        exp_month: expMonth,
        exp_year: expYear
    }, stripeResponseHandler);

}
```

There are comments inline that explain each step, and most of the information can also be found in Stripe's documentation. The validation steps make repeated reference to a `reportError()` function. For now, that's defined as so:

```
function reportError(msg) {

    // Show the error in the form:
    alert(msg);

    // re-enable the submit button:
    $('#submit-btn').prop('disabled', false);
```

```
    return false;
}
```

Again, this all goes within the `registerScript()` line. Obviously there are many ways you could improve the `reportError()` function; I'm mostly focusing on easy-to-understand and functional-enough here.

Finally, when the Stripe request returns a response, it's handled by the `stripeResponseHandler()` function:

```
function stripeResponseHandler(status, response) {

    // Check for an error:
    if (response.error) {

        reportError(response.error.message);

    } else { // No errors, submit the form:

      var f = $('#pay-form');

      // Token contains id, last4, and card type:
      var token = response['id'];

      // Insert the token into the form so it gets submitted to the server
      f.append('<input type=\"hidden\" name=\"Payment[token]\"
      value=\"' + token + '\">');

      // Submit the form:
      f.get(0).submit();

    }

} // End of stripeResponseHandler() function.
```

This code checks the response from Stripe for an error. If one exists, it's sent to the `reportError()` function. Otherwise, the token is grabbed from the response and then stored in a hidden input. Finally, the form is submitted.

Whew! Complicated JavaScript, but it works. Now on to the PHP code that actually processes the payment with Stripe.

Making the Payment Just to reiterate what the process is, Stripe requires both client-side code (the form and the JavaScript) and server-side code. The client-side code securely sends the customer's payment information to Stripe so that it doesn't

touch your server. But it's the server-side code that performs the actual charge. That will be done in the controller action that handles the form's submission. All of the following will go in `actionIndex()`, after `if($model->validate()) {`.

All of the new code should go within a `try...catch` block in order to handle the errors that could occur. Within the `try`, the code must first import the Stripe library and set the secret key:

```
try {
    Yii::import('pay.vendors.stripe.lib.Stripe');
    Stripe::setApiKey(Yii::app()->controller
        ->module->private_key);
```

Next, you can attempt the charge at Stripe, passing along:

- The amount
- The currency
- The token
- A description

```
$charge = Stripe_Charge::create(array(
    "amount" => $model->amount,
    "currency" => "usd",
    "card" => $model->token,
    "description" => $model->email
    )
);
```

Next, the code should check the response, confirming that it was paid. If so, then the `charge_id` (set in Stripe) can be assigned, the model can be saved, and the user redirected:

```
// Check that it was paid:
if ($charge->paid == true) {
    $model->charge_id = $charge->id;
    $model->save();
    $this->redirect(array('thanks', 'amount' => $model->amount));
}
```

If an exception occurred, it must be caught. Most exceptions will be of type `Stripe_CardError`:

```
} catch (Stripe_CardError $e) {
    $e_json = $e->getJsonBody();
    $err = $e_json['error'];
    $model->addError('Credit Card', $err['message']);
```

That's how you parse the error message out of the Stripe response. The message will be sufficiently useful. To have it display on the page, I'm assigning it to a non-existent model property. The view file will still display this (**Figure 19.16**).

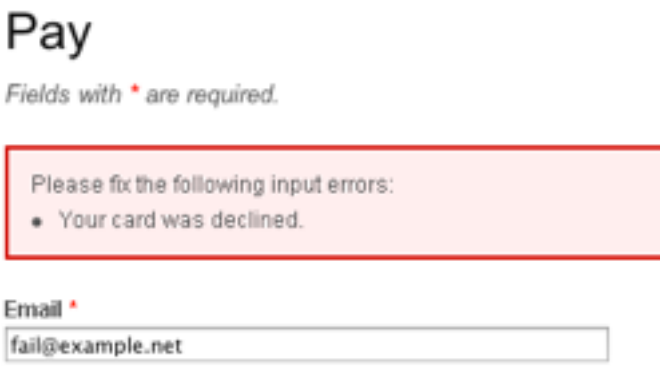

Figure 1.16: *How the Stripe error message appears on the page.*

Catch other Stripe exception types:

```
} catch (Stripe_ApiConnectionError $e) {
    // Network problem, perhaps try again.
} catch (Stripe_InvalidRequestError $e) {
    // You screwed up in your programming. Shouldn't happen!
} catch (Stripe_ApiError $e) {
    // Stripe's servers are down!
} catch (Stripe_CardError $e) {
    // Something else that's not the customer's fault.
}
```

You would want to handle these in other ways, but you can use the same construct already laid out to have the error be shown.

And that's it! Should the charge succeed, the model will be saved in the database and the user redirected (**Figure 19.17**).

Should it fail, the error message will be shown on the page, giving the user the chance to try again.

And that completes the module. It is ready to be used. Once you polish it up, provide adequate documentation, and so forth…

To test the Stripe payment integration (after configuring the module), see Stripe's documentation.

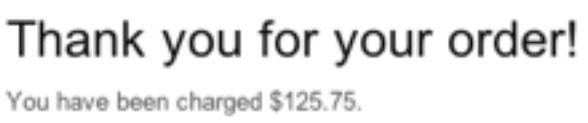

Figure 1.17: *The result of a successful payment.*

Configuring the Module To use this module, it just needs to be provided with the right Stripe keys:

```
'modules'=>array(
    'stripetest' => array(
        'public_key' => 'pk_test_APr32Tly9WH6K9XfZpJeEKCH',
        'private_key' => 'sk_test_Ca0U4KkAPr32TZpJeEKCH'
    ),
```

Everything else is handled by the module, save for the amount issue, to be discussed in more detail shortly. Other configuration ideas include allowing for the extension user to

- Change the currency
- Customize the payment description
- Opt-in to creating customer objects in Stripe as well as payments

Improving the Module There are many ways this module could be written differently or changed. I've already mentioned a couple, but the most important step in making this a distributable extension would be to provide good documentation as to how you'd use it. This is particularly important as the extension's user might want to customize the layout of the form, and you'd want to set the right rules for doing so.

Towards that end, one improvement would be to change the form creation from a static view file to a use of form builder. This would further separate the requirements from the display, making it much easier for the extension user to change the look of the form.

The only requirement that's not well handled by the extension is how the amount gets to it. Truly, an exception ought to be thrown when no amount value exists, as it is required. But the amount could come from many places, so a sophisticated solution would be to have this as a configurable option. The extension user could indicate the source and attribute or index where the amount can be found: the X attribute of the Y model; the X index of the session; etc. Then the module could retrieve it and throw an exception on error.

I welcome anyone to improve upon this extension in any way, and even to distribute it!

Deploying Extensions

If you've created an extension that you think is worth sharing (yay!), you can offer it up on your own Web site, but it'd be better placed among the other Yii extensions.

In order to share an extension, you'll need to be a registered Yii forum member. You'll also want to create a new forum thread within the extensions forum where you can answer questions and get feedback. User feedback is a great way to improve the quality of your work (believe me, I know!), so pay attention to your forum thread, duly consider the feedback you get, and continue to maintain your extension.

And thanks for sharing! Not to be overly sentimental about it, but the strength of the community is one of the things makes Yii so great.