IN THIS CHAPTER

- Exploring the hype about MVC programming
- Comparing MVC to other programming methods
- Implementing MVC
- Using MVC in a client/server environment

Chapter 1
The MVC Method

In Book 6, I walk you through how to create a complete dynamic web application using object-oriented PHP programming techniques. But there's more than one way to design and program an object-oriented PHP application. There are, in fact, many different theories on just how best to design and code your dynamic web applications with PHP, all with their own pros and cons. This chapter walks you through one of the more popular methods for designing object-oriented web applications and compares it to some other methods available.

Getting Acquainted with MVC

If you've been spending any time reading articles, books, or even discussion group posts about PHP programming, you've come across the term *MVC.* The acronym stands for *model–view–controller,* which is a method of splitting your object-oriented program code into multiple parts to make it easier to code and implement in an object-oriented environment.

This section first describes how MVC works and then digs deeper into each of the separate components that make up an MVC application.

Exploring the MVC method

The MVC method of programming actually predates the web programming world by quite a bit. It was designed in the early days of graphical desktop programming as a way to help organize applications that required lots of coding to support user interaction. Instead of intertwining the display code inside the application code, developers decided early on it was best to try to separate those features into separate components.

The MVC method divides a graphical application into three basic components:

- **The model:** One or more classes that interact with the application data and implement the coding logic required to store and manipulate application data
- **The view:** A class that displays the application data in the graphical environment
- **The controller:** A class that listens for user input and passes the input to the appropriate model class methods for processing

Web application programming has many similarities to desktop graphical application programming, so many web developers have adopted the MVC method for creating object-oriented web applications.

Most MVC experts agree on the three basic components of the application, but there are several different theories on how they should interact with one another. Before your eyes start glazing over hearing the word *theory*, let me just show you the most popular interpretation of MVC theory, as shown in Figure 1-1.

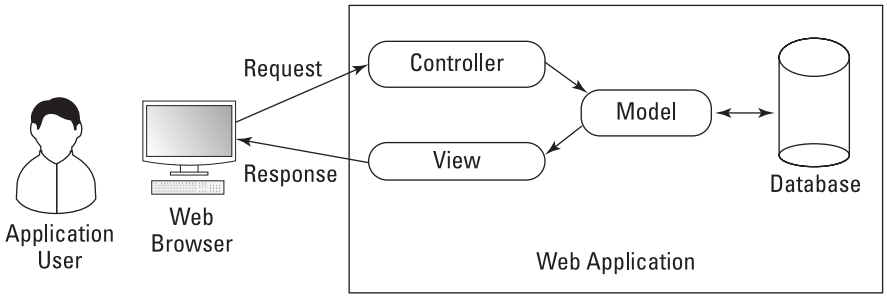

FIGURE 1-1: A basic MVC theory diagram.

Most MVC experts agree that the controller component is the "front door" to your application. The controller's job is to be the traffic cop of the application, shuttling user requests to the correct model class method to interact with the

requested data — whether that's inserting new data, updating existing data, or just listing the existing data (or some subset of the data). The model class methods do the actual work to produce the data required to support the user request and then pass the result to the view.

The view's job is to organize the result of the request into a visual output. That visual output could be as simple as a notice that the operation was successful, or as complex as displaying a full report of requested data elements. Whatever the result of the request, it's the view's job to communicate it to the application user.

You can probably start to see how the MVC method can easily fit into your dynamic web applications. Web applications receive requests from client browsers requesting some type of data operation, and then need to display the requested data back to the client browser in the form of a web page. Being able to split these functions into separate components can be handy.

This feature is attractive in large development environments where it's easier to separate the different coding requirements between development groups. With the MVC programming method, you can allow your best HTML and CSS programmers to focus on the view code, while your best PHP and SQL coders can focus on the model and controller code.

This also means that you can have multiple development groups working on the application code at the same time, without causing problems for each other. With different groups simultaneously writing code, the application may get done quicker than with other object-oriented approaches.

Of course, with all coding methods, there are downsides, and that's certainly the case with the MVC method of application development. One of the biggest complaints you'll see regarding applications developed using the MVC method is that they can be somewhat difficult to understand and troubleshoot.

Breaking code up into separate components can make trying to follow the application code more complicated. As a client makes a request for data, multiple class files get involved with the process. Instead of being able to trace the execution of a single application file, you'll find yourself having to dig through several different smaller application files, looking for the one bug that's causing the issue.

Yet another complaint about the MVC method is that it can be somewhat hard to implement in a programming environment. It can take time and practice incorporating the MVC programming method into an application, time that most web application developers don't have. Fortunately, there are tools available to help out with that, which I cover in Chapter 2 of this minibook.

Digging into the MVC components

Now that you've seen the high-level overview of just what MVC is, let's take a look at the internals required for the individual components. This section goes a little more in depth into what each of the MVC components does.

The model

The model component of the MVC method is where the majority of the PHP application coding takes place. Its job is to provide a common interface between the application and any data that the application requires.

Of course these days, most web applications use some type of database system to store the application data (such as the MySQL server in the application example). The model code sits between the application and the database tables. Any access to the data must go through the model code.

Most MVC model implementations use a technique called *object-relational mapping* (ORM) to provide this interface. The ORM class is responsible for handling the methods for all interaction with the underlying table:

- Creating new data records
- Reading existing data records
- Updating existing data records
- Deleting existing data records

The combination of the create, read, update, and delete methods are commonly referred to as CRUD. Besides the four CRUD methods, the model class often contains additional methods for any type of data manipulation that are required to support the application.

There are two different approaches to how the ORM interacts with the data used in an application:

- **Relational data method:** In the relational data method, you create a model class for each table contained in the application database. For example, in the AuctionHelper application discussed in Book 6, which contains a Bidders table and an Items table, you'd need to create a model class for the Bidders table and a second model class for the Items table. The model classes use standard SQL to interact with the data contained in the tables. Figure 1-2 demonstrates this method.

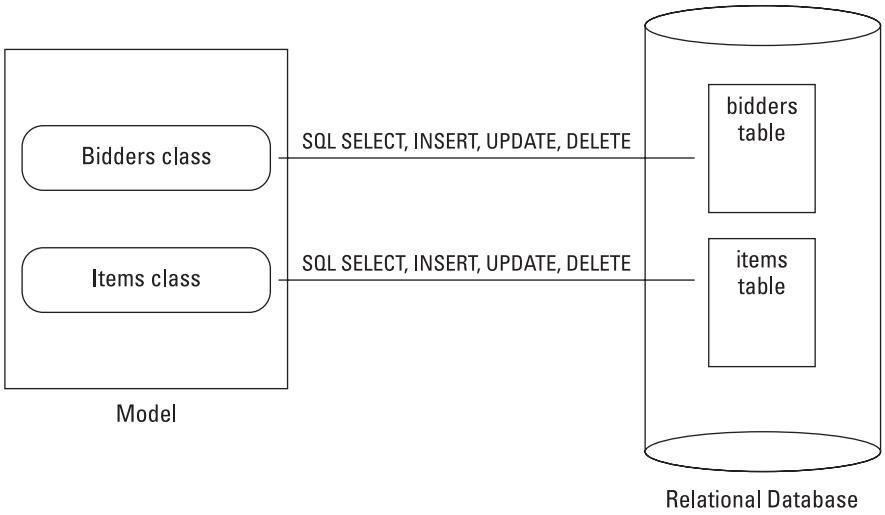

FIGURE 1-2: The relational data method model.

» **Object-oriented data method:** The object-oriented data method takes a slightly different approach to interfacing the application to the underlying database tables. Instead of using a relational database method, this uses an *object-oriented database management system* (OODBMS), which stored data as objects instead of tables. Because the data is stored as objects, the model class objects can more easily map directly to the database objects. Figure 1-3 demonstrates the object-oriented data method.

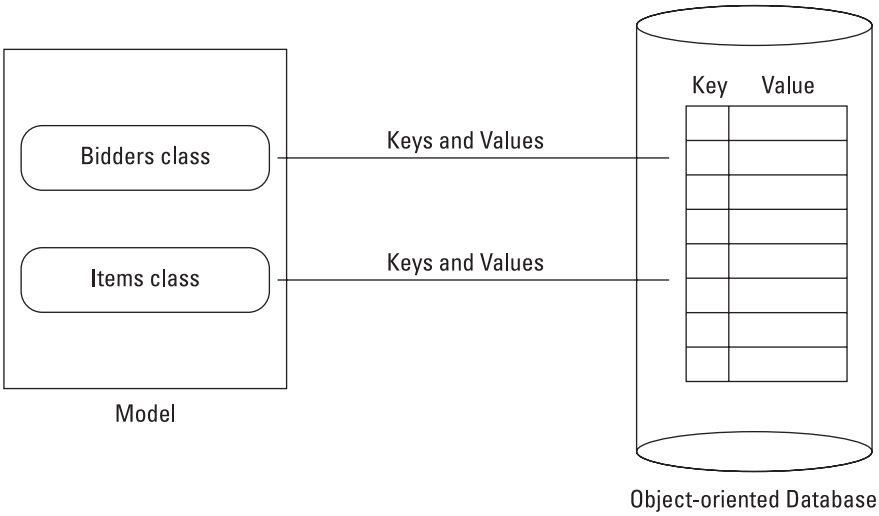

FIGURE 1-3: The object-oriented data method model.

With an OODBMS, related data is automatically grouped together in the database as keys and their associated values, such as the bidders as keys and the items they've won as their values. There is no formal separation of the data into tables like in the relational database method. By grouping these objects together, you create a virtual object that is quicker to query and retrieve the information by just submitting keys to the database and retrieving the associated values.


TIP

With the growing popularity of OODBMS theory, a few different OODBMS database servers have been developed. Currently, the popular OODBMS server is the NoSQL server project. It stores data as XML files that can be easily added and appended as the data grows in an application.

The view

The view component is responsible for all the output from the application. It takes the raw data provided by the model component and formats it in a way that's visually pleasing to the application user. For our web applications, the view component is where all the HTML and CSS code resides.

The view component code is often placed into a folder area in the application, with different files responsible for creating different features of the application. This completely separates the view code from the model and component code in the application.

In the AuctionHelper application, I chose not to implement a separate view component, but instead placed the code to display the application data directly in the files that controlled the data. This requires mixing the PHP, HTML, CSS, and even JavaScript code into the same files. That's fine if you're writing your own application, but it can get confusing if you're working in a programming environment that divides the modules up between separate programming groups.

One area where having a separate code component to handle the view comes in handy is when working with mobile devices. These days, it may not be sufficient to write your application solely to display web pages for a normal desktop browser environment. With the popularity of mobile devices, your application may need to be usable (and readable) in both the desktop and mobile environments.

That may require that you create different sets of CSS styles (and sometimes even different sets of HTML code) for different display environments. Mobile devices have a much smaller display area and need some extra consideration to ensure your website visitors can interact properly with the application. Trying to maintain two separate code bases can get confusing, especially if you're embedding the view code within your application.

Isolating the code required to generate the display output into a separate component makes it easier to incorporate multiple code for multiple devices — one set of view code for desktop browsers and another set for mobile devices. Figure 1-4 demonstrates how this works.

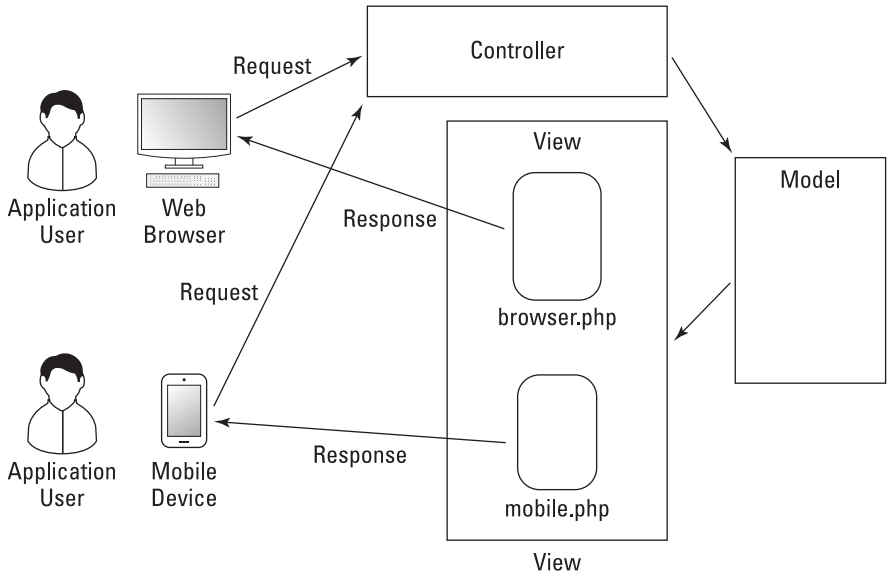

FIGURE 1-4: Using multiple view modules for different display environments.

Two devices submit the same HTTP request to the controller, which forwards both requests to the model. The model sends the same responses to the view, but the view processes the responses differently.

TIP

Both CSS3 and JavaScript provide ways for you to detect the device monitor size and determine whether the website visitor is viewing the application on a mobile device or desktop browser. In CSS3, the `max-width` property in the `@media` rule allows the browser to select which CSS rules to apply based on the size of the browser window. For JavaScript, you can use the `screen.width` global property to determine the viewing area size.

The controller

The controller accepts requests from the application user and sends them to the components required to satisfy the request. That means being able to communicate between all the model classes, as well as the view files required to display the data.

The controller uses *routing* to determine which model class method to run based on the client browser's request. Routing maps the specific HTTP `GET` or `POST` request received from a client browser to a specific model class method.

In the AuctionHelper application, I did that by setting the HTML `content` variable/value pair. The `index.php` code acted as the application controller, directing which include file to display as the main web page content. So, to display the details of the bidder with a `bidderid` value of 100, the client browser sent the following request:

```
index.php?content=showbidder&id=100
```

The `index.php` code in the AuctionHelper application retrieved the `content` HTML variable/value pair and then used that value to include the `showbidder.inc.php` include file, which then used the `id` value of 100 to display the appropriate bidder information.

MVC controllers use a similar method but utilize the *rewrite rules* feature of web servers to help clean up the format of the request URL. Rewrite rules allow you to customize the format of the URL to pass information in a cleaner-looking format than what the standard `GET` method uses. For example, the URL request to show information for bidder 100 might look like this:

```
index.php/bidders/show/100
```

The web server parses the URL to set the HTML variable/value pairs. Then the controller routing rules direct the application to call the `show` method of the `bidders` model class and pass it the `bidderid` value of 100.

TIP

Search engine optimization (SEO) is the process of designing your application to make it easier for Internet search engines (such as Google) to find and catalog your website pages. Web server rewrite rules can play a crucial role in helping add to your search engine visibility, as many search engines won't scan websites with URLs that contain long lists of variable/value pairs. By parsing out the URL data automatically, you can trim off quite a bit of length in your URLs, making them more SEO-friendly!

Communicating in MVC

In the MVC method, because you must divide all the functions of your web application into separate components, communication between the components becomes crucial. Each component must know when and how to communicate information to the other components for the application to function correctly.

Earlier, Figure 1-1 showed the classical MVC communication method. There were five separate steps for communicating with a website visitor's request:

1. The controller receives the request from the website visitor's browser.
2. The controller passes the request to the appropriate class method in the model component.
3. The model class method performs the appropriate action with the data based on the request.
4. The model class method passes any resulting data or status to the view.
5. The view sends a response back to the website visitor with the data, formatted appropriately for the visitor's display device.

That all seems organized and proper, but there are some holes in this process that can cause issues in the application:

- **The controller is responsible for handling the client request but is not responsible for returning the response.** If any special communication is required for the session (such as an encryption key or session ID), the view must get that information from the controller.
- **The model is responsible for retrieving the data required to satisfy the request, but the view is responsible for the format in which the data appears in the display.** This can make common web page features such as paging through long result sets of data on multiple web pages more complex. Paging through data is usually more easily handled with SQL directives in the model code rather than PHP in the view code. This means the view and the model must communicate information between each other as well.
- **The view must have knowledge of the client browser's environment to format the data to display properly on the client device.** This may require communication between the view and the controller that initiated the session.

Because of little issues like these, many MVC implementations violate the strict MVC method rules and implement communication between the different component classes. This helps eliminate issues within the application and provide a smoother interface for the website visitor.

Comparing MVC to Other Web Models

As you might guess, the MVC method is not the only theory available for creating object-oriented web applications. This section explores a couple of other popular methods that you may encounter as you explore the world of object-oriented web applications: the MVP method and the MVVM method.

The MVP method

The *model–view–presenter* (MVP) method is another popular method of creating object-oriented web applications. At first, its name may sound a bit redundant — the presenter sounds as if it's doing the same job as the view.

The MVP method takes a more linear approach to the process of handling client requests, as shown in Figure 1-5.

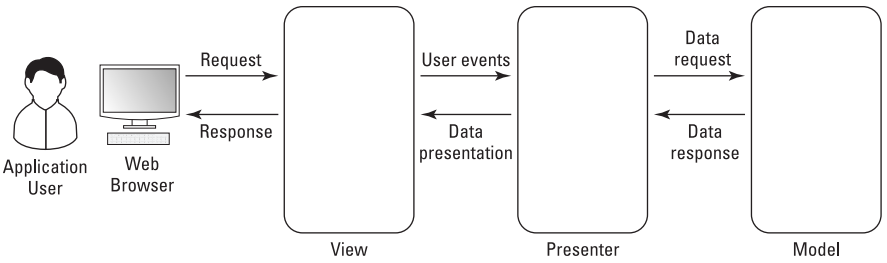

FIGURE 1-5: The model–view–presenter method.

In the MVP method, the view handles both the request and response parts of the process, taking on the MVC controller's function of communicating with the client.

The presenter acts as the middleman between the model and the view. It interprets the client requests and calls the appropriate model class methods. After the model processes the request and generates the appropriate response, it sends the response to the presenter, which passes it to the view to format for display.

As you can see, the MVP method basically splits the controller jobs from the MVC method between the view and presenter modules, making things a little more streamlined. This helps eliminate some of the communication issues presented in the MVC method.

The MVVM method

The *model–view–viewmodel* (MVVM) method is similar to the MVP method, but with a slight twist, as shown in Figure 1-6.

The viewmodel acts as a middleman between the view and the model, similar to the presenter module in MVP. But unlike the MVP presenter module, the viewmodel doesn't manipulate the data — it just provides an interface between the view and the model.

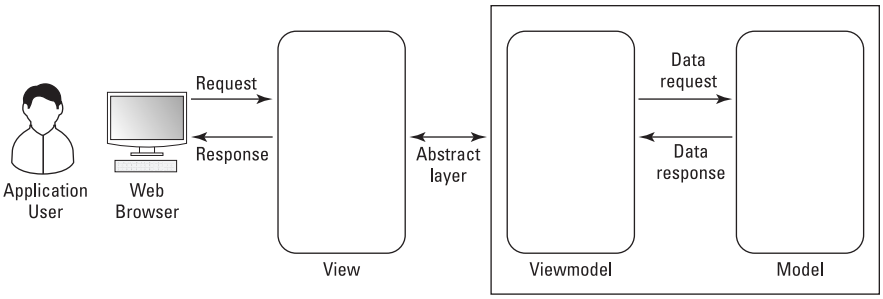

FIGURE 1-6: The model–view–viewmodel method.

The viewmodel creates an abstract layer between the graphical environment of the view and the data-centric environment of the model. This abstract layer allows the programmers working on the view code to provide an interface to the data in the model without having to know the details of the underlying data or how the model handles it.

This data abstraction helps the development process, because the user interface often changes more frequently than the underlying data. The graphical designers can make changes without worrying about messing up the database designers.

Seeing How MVC Fits into N-Tier Theory

If you've been doing any type of web application development in a large-scale environment, you've probably heard of or even used the *multitier architecture* (often called *n-tier*) approach to web applications. With *n*-tier architecture, developers divide a web application into separate physical servers, based on functionality, as shown in Figure 1-7.

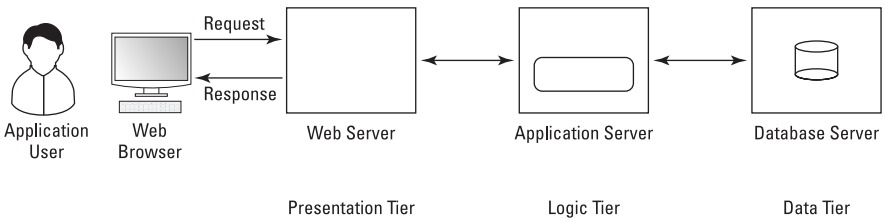

FIGURE 1-7: The *n*-tier theory architecture.

The *n*-tier architecture layout often consists of three physical servers:

- A web server that interacts with client browsers (called the *presentation tier*)
- An application server that runs the server-side application code (called the *logic tier*)
- A database server that stores the application data (called the *data tier*)

The main goal of the *n*-tier architecture is to divide the separate functions of a web application into separate physical servers. This action helps prevent server bottlenecks and makes it easier to both expand individual servers as needed to support application load or share server resources between web applications.

The web–application–database layout of the *n*-tier architecture sounds a lot like the model–view–controller model of MVC, but there's a difference. It's important to remember that the *n*-tier architecture refers to the physical server environment of the application and not the application software. The MVC method represents a method of dividing the software requirements of a web application, regardless of the underlying hardware.

The *n*-tier architecture can support any type of software application method, and your MVC application can run in any type of server environment. It's certainly possible to implement your MVC application in an *n*-tier server environment, but you can just as easily run it within a standard one-server web environment.

Implementing MVC

In the AuctionHelper application, you use PHP code on the server and HTML, CSS, and JavaScript code on the client to create the application. The first question that often comes to mind when considering the MVC programming method is "Where is that implemented?"

The MVC method isn't necessarily a client-side or server-side programming paradigm. There are parts of the MVC method that work in either side of the web application environment. Trying to figure out which parts fit where can be somewhat of a challenge.

Most MVC implementations focus on the server-side programming environment and leave the client-side code to presenting the data from the view component. However, with Ajax technology (see Book 6, Chapter 3), you can implement parts of the model in the client as well, retrieving data as needed from the database tables on the server. Because this book is primarily about writing PHP applications, I focus on using MVC in the PHP server-side programming code.

Fortunately, there are many tools available to help you utilize MVC concepts within your PHP applications. The next chapter discusses the use of these programming tools.