CHAPTER 2

Laying Out the Foundations

Now that you've convinced the client that you can create a cool web site to complement his or her activity, it's time to stop celebrating and start thinking about how to put into practice all the promises you've made. As usual, when you lay down on paper the technical requirements you must meet, everything starts to seem a bit more complicated than initially anticipated.

To ensure this project's success, you need to come up with a smart way to implement what you agreed to when you signed the contract. You want to develop the project smoothly and quickly, but the ultimate goal is to make sure the client is satisfied with your work. Consequently, you should aim to provide your site's increasing number of visitors with a positive web experience by creating a pleasant, functional, and responsive web site.

The requirements are high, but this is normal for an e-commerce site today. To maximize the chances of success, we'll analyze and anticipate as many of the technical requirements as possible and implement solutions in a way that supports changes and additions with minimal effort. Your goals for this chapter are to

- Analyze the project from a technical point of view
- Analyze and choose the architecture for your application
- Decide which technologies, programming languages, and tools to use
- Consider naming and coding conventions

Note Be warned that this and the next few chapters are dense, and you may find them pretty challenging if you don't have much experience with PHP or MySQL. Books such as *Beginning PHP and MySQL 5: From Novice to Professional,Second Edition* (W. Jason Gilmore. Apress, 2006.) do a good job of preparing you to build your first e-commerce web site.

Also, we strongly recommend that you consistently follow an efficient project management methodology to maximize the chances of the project's success, on budget and on time. Most project management theories imply that you and your client have signed an initial requirements/specifications document containing the details of the project you're about to create. You can use this document as a guide while creating the solution; it also allows you to charge extra if the client brings new requirements or requests changes after development has started.

Designing for Growth

The word "design" in the context of a web application can mean many things. Its most popular usage probably refers to the visual and user interface design of a web site.

This aspect is crucial because, let's face it, the visitor is often more impressed with how a site looks and how easy it is to use than about which technologies and techniques are used behind the scenes or what operating system the web server is running. If the site is slow, hard to use, or easy to forget, it just doesn't matter what rocket science was used to create it.

Unfortunately, this truth makes many inexperienced programmers underestimate the importance of the way the invisible part of the site is implemented—the code, the database, and so on. The visual part of a site gets visitors interested to begin with, but its functionality makes them come back. A web site can sometimes be implemented very quickly based on certain initial requirements, but if not properly architected, it can become difficult, if not impossible, to change.

For any project of any size, some preparation must be done before starting to code. Still, no matter how much preparation and design work is done, the unexpected does happen, and hidden catches, new requirements, and changing rules always seem to work against deadlines. Even without these unexpected factors, site designers are often asked to change or add functionality many times after the project is finished and deployed. This will also be the case for TShirtShop, which will be implemented in three separate stages, as discussed in Chapter 1.

You will learn how to create the web site so that the site (or you) will not fall apart when functionality is extended or updates are made. Because this is a programming book, instead of focusing on how to design the user interface or on marketing techniques, we'll pay close attention to designing the code that makes them work.

The phrase "designing the code" can have different meanings; for example, we'll need to have a short talk about naming conventions. Yet, the most important aspect that we need to take a look at is the application architecture. The architecture refers to the way you split the code into smaller components (for example, the product search feature) for a simple piece of functionality. Although it might be easier to implement that functionality as quickly and as simply as possible in a single component, you gain great long-term advantages by creating smaller, more simple components that work together to achieve the desired result.

Before talking about the architecture itself, you must determine what you want from this architecture.

Meeting Long-Term Requirements with Minimal Effort

Apart from the fact that you want a fast web site, each of the phases of development we talked about in Chapter 1 brings new requirements that must be met.

Every time you proceed to a new stage, you want to be able to *reuse* most of the already existing solution. It would be very inefficient to redesign the whole site (not just the visual part but the code as well!) just because you need to add a new feature. You can make it easier to reuse a solution by planning ahead, so any new functionality that needs to be added can be plugged in with ease, rather than each change causing a new headache.

When building the web site, implementing a *flexible architecture* composed of pluggable components allows you to add new features—such as the shopping cart, the departments list, or the product search feature—by coding them as separate components and plugging them into the existing application. Achieving a good level of flexibility is one of the main goals regarding the application's architecture, and this chapter shows how you can put this into practice.

You'll see that the flexibility level is proportional to the amount of time required to design and implement it, so we'll try to find a compromise that will provide the best gains without complicating the code too much.

Another major requirement that is common to all online applications is having a *scalable architecture*. Scalability is defined as the capability to increase resources to yield a linear increase in service capacity. In other words, ideally, in a scalable system, the ratio (proportion) between the number of client requests and the hardware resources required to handle those requests is constant, even when the number of clients increases. An unscalable system can't deal with an increasing number of clients, no matter how many hardware resources are provided. Because we're optimistic about the number of customers, we must be sure that the site will be capable of delivering its functionality to a large number of clients without throwing out errors or performing sluggishly.

Reliability is also a critical aspect for an e-commerce application. With the help of a coherent error-handling strategy and a powerful relational database, you can ensure data integrity and ensure that noncritical errors are properly handled without bringing the site to its knees.

The Magic of the Three-Tier Architecture

Generally, the architecture refers to the way we split the code that implements a feature of the application into separate components based on what they do and grouping each kind of component into a single logical tier.

In particular, the three-tier architecture refers to an architecture that is based on these tiers:

- The presentation tier
- The business tier
- The data tier

The *presentation tier* contains the user interface elements of the site and includes all the logic that manages the interaction between the visitor and the client's business. This tier makes the whole site feel alive, and the way you design it has a crucial importance for the site's success. Because your application is a web site, its presentation tier is composed of dynamic web pages.

The *business tier* (also called the *middle tier*) receives requests from the presentation tier and returns a result to the presentation tier depending on the business logic it contains. Almost any event that happens in the presentation tier usually results in the business tier being called (utilized), except events that can be handled locally by the presentation tier, such as simple input data validation, and so on. For example, if the visitor is doing a product search, the presentation tier calls the business tier and says, "Please send me back the products that match this search criterion." Most of the time, the business tier needs to call the data tier for information to be able to respond to the presentation tier's request.

The *data tier* (sometimes referred to as the *database tier*) is responsible for managing the application's data and sending it to the business tier when requested. For the TShirtShop e-commerce site, you'll need to store data about products (including their categories and their departments), users, shopping carts, and so on. Almost every client request finally results in the data tier being interrogated for information (except when previously retrieved data has been cached at the business tier or presentation tier levels), so it's important to have a fast

database system. In Chapters 4 and 5, you'll learn how to design the database for optimum performance.

These tiers are purely logical—there is no constraint on the physical location of each tier. In theory, you are free to place all of the application, and implicitly all of its tiers, on a single server machine, or you can place each tier on a separate machine if the application permits this. Chapter 22 explains how to integrate functionality from other web sites using XML Web Services. XML Web Services permit easy integration of functionality across multiple servers without the hassle of customized code.

An important constraint in the three-tier architecture model is that information must flow in sequential order among tiers. The presentation tier is only allowed to access the business tier, and it can never directly access the data tier. The business tier is the brain in the middle that communicates with the other tiers and processes and coordinates all the information flow. If the presentation tier directly accessed the data tier, the rules of three-tier architecture programming would be broken.

These rules may look like limitations at first, but when utilizing an architecture, you need to be consistent and obey its rules to reap the benefits. Sticking to the three-tier architecture ensures that your site remains easily updated or changed and adds a level of control over who or what can access your data. This may seem to be unnecessary overhead for you right now; however, there is a substantial future benefit of adhering to this system whenever you need to change your site's functioning or logic.

Figure 2-1 is a simple representation of the way data is passed in an application that implements the three-tier architecture.

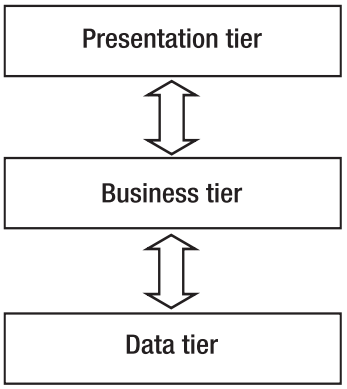

Figure 2-1. *Simple representation of the three-tier architecture*

A Simple Example Using the Three-Tier Architecture

It's easier to understand how data is passed and transformed between tiers if you take a closer look at a simple example. To make the example even more relevant to our project, let's analyze a situation that will actually happen in TShirtShop. This scenario is typical for three-tier applications.

Like most e-commerce sites, TShirtShop will have a shopping cart, which we will discuss later in the book. For now, it's enough to know that the visitor will add products to the shopping cart by clicking an Add to Cart button. Figure 2-2 shows how the information flows through the application when that button is clicked.

At step 1, the user clicks the Add to Cart button for a specific product. At step 2, the presentation tier (which contains the button) forwards the request to the business tier, "Hey, I want this product added to my shopping cart!" At step 3, the business tier receives the request, understands that the user wants a specific product added to the shopping cart, and handles the request by telling the data tier to update the visitor's shopping cart by adding the selected product. The data tier needs to be called, because it stores and manages the entire web site's data, including users' shopping cart information.

At step 4, the data tier updates the database and eventually returns a success code to the business tier. At step 5, the business tier handles the return code and any errors that might have occurred in the data tier while updating the database and then returns the output to the presentation tier.

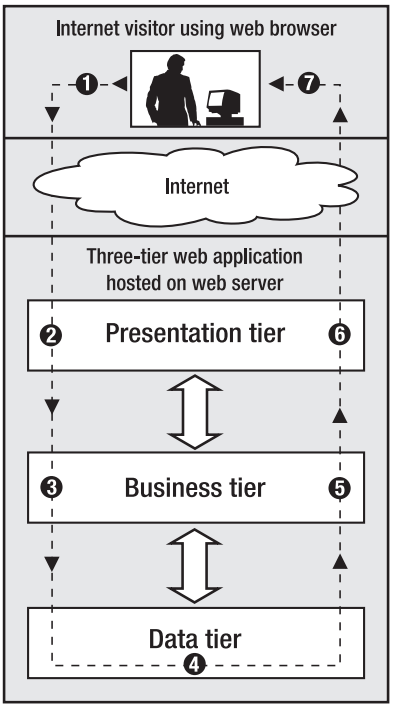

Figure 2-2. *Internet visitor interacting with a three-tier application*

At step 6, the presentation tier generates an updated view of the shopping cart. At step 7, the results of the execution are wrapped up by generating a Hypertext Markup Language (HTML) web page that is returned to the visitor where the updated shopping cart can be seen in the visitor's web browser.

Note that, in this simple example, the business tier doesn't do a lot of processing, and its business logic isn't very complex. However, if new business rules appear for your application, you would change the business tier. If, for example, the business logic specified that a product could be added to the shopping cart only if its quantity in stock was greater than zero, an additional data tier call would have been made to determine the quantity. The data tier would be requested to update the shopping cart only if products are in stock. In any case, the presentation tier is informed about the status and provides human-readable feedback to the visitor.

What's in a Number?

It's interesting to note how each tier interprets the same piece of information differently. For the data tier, the numbers and information it stores have no significance because this tier is an engine that saves, manages, and retrieves numbers, strings, or other data types—to the data tier this data is just arbitrary information, not product quantities or product names. In the context of the previous example, a product quantity of zero represents a simple, plain number without any meaning to the data tier (it is simply zero, a 32-bit integer).

The data only gains significance when the business tier reads it. When the business tier asks the data tier for a product quantity and gets a "0" result, this is interpreted by the business tier as, "Hey, no products in stock!" This data is finally wrapped in a nice, visual form by the presentation tier, such as a label reading, "Sorry, at the moment this product cannot be ordered."

Even if it's unlikely that you want to forbid a customer from adding a product to the shopping cart if the product is not in stock, the example (described in Figure 2-3) is good enough to present in yet another way how each of the three tiers has a different purpose.

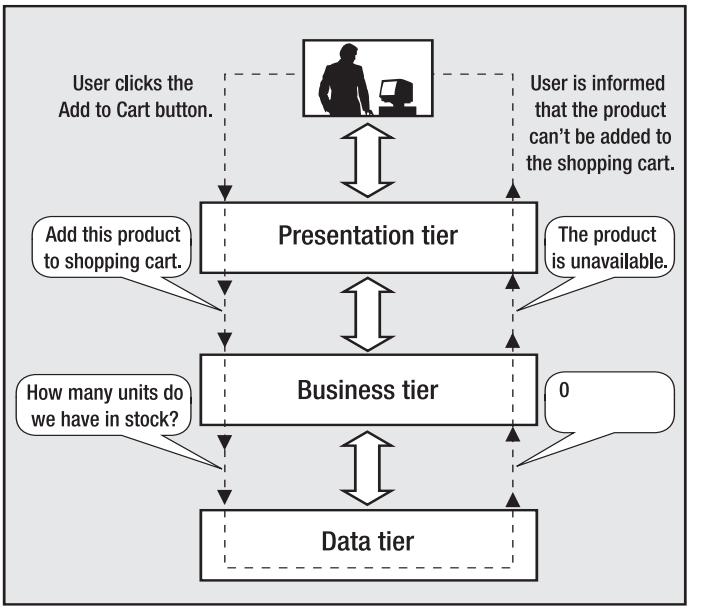

Figure 2-3. *Example of information exchange among application tiers*

The Right Logic for the Right Tier

Because each layer contains its own logic, sometimes it can be tricky to decide where exactly to draw the lines between tiers. In the previous scenario, instead of reading the product's quantity in the business tier and deciding whether the product is available based on that number (resulting ultimately in two database calls), you could have a single stored procedure named `add_product_if_available` that adds the product to the shopping cart only if it's available in stock.

In this scenario, some logic is transferred from the business tier to the data tier. In many other circumstances, you might have the option to place some logic in one tier or another or

maybe in both. In most cases, there is no single best way to implement the three-tier architecture, and you'll need to make a compromise or a choice based on personal preference or external constraints.

Furthermore, there are occasions in which even though you know the *right* way (in respect to the architecture) to implement something, you might choose to break the rules to get a performance gain. As a general rule, if performance can be improved this way, it is OK to break the strict limits between tiers *just a little bit* (for example, add some of the business rules to the data tier or vice versa), *if* these rules are not likely to change in time. Otherwise, keeping all the business rules in the middle tier is preferable, because it generates a cleaner application that is easier to maintain.

Finally, don't be tempted to access the data tier directly from the presentation tier. This is a common mistake that is the shortest path to a complicated, hard-to-maintain, and inflexible system. In many data access tutorials or introductory materials, you'll be shown how to perform basic database operations using a simple user interface application. In these kinds of programs, all the logic is probably written in a short, single file, instead of separate tiers. Although the materials might be very good, keep in mind that most of these texts are meant to teach you how to do different individual tasks (for example, access a database), and not how to correctly create a flexible and scalable application.

A Three-Tier Architecture for TShirtShop

Implementing a three-tier architecture for the TShirtShop web site will help achieve the goals listed at the beginning of the chapter. The coding discipline, imposed by a system that might seem rigid at first sight, allows for excellent levels of flexibility and extensibility in the long run.

Splitting major parts of the application into separate smaller components encourages reusability. More than once when adding new features to the site, you'll see that you can reuse some of the already existing bits. Adding a new feature without needing to change much of what already exists is, in itself, a good example of reusability.

Another advantage of the three-tiered architecture is that, if properly implemented, the overall system is resistant to changes. When bits in one of the tiers change, the other tiers usually remain unaffected, sometimes even in extreme cases. For example, if for some reason the back-end database system is changed (say, the manager decides to use PostgreSQL instead of MySQL), you only need to update the data tier and maybe just a little bit of the business tier.

Why Not Use More Tiers?

The three-tier architecture we've been talking about so far is a particular (and the most popular) version of the *n*-tier architecture. *n-tier architecture* refers to splitting the solution into a number (*n*) of logical tiers. In complex projects, sometimes it makes sense to split the business layer into more than one layer, thus resulting in architecture with more than three layers. However, for our web site, it makes the most sense to stick with the three-layered design, which offers most of the benefits while not requiring too many hours of design or a complex hierarchy of framework code to support the architecture.

Maybe with a more involved and complex architecture, you could achieve even higher levels of flexibility and scalability for the application, but you would need much more time for design before starting to implement anything. As with any programming project, you must find a fair balance between the time required to design the architecture and the time spent to implement it. The three-tier architecture is best suited to projects with average complexity, such as the TShirtShop web site.

You also might be asking the opposite question, "Why not use fewer tiers?" A two-tier architecture, also called *client-server* architecture, can be appropriate for less complex projects. In short, a two-tier architecture requires less time for planning and allows quicker development in the beginning; however, it generates an application that's harder to maintain and extend in the long run. Because we're expecting to have to extend the application in the future, the client-server architecture is not appropriate for our application, so it won't be discussed further in this book.

Now that the general architecture is known, let's see what technologies and tools you will use to implement it. We'll have a brief discussion of the technologies, and in Chapter 3, you'll create the foundation of the presentation and data tiers by creating the first page of the site and the back-end database. You'll start implementing real functionality in each of the three tiers in Chapter 4 when you start creating the web site's product catalog.

Choosing Technologies and Tools

No matter which architecture is chosen, a major question that arises in every development project is which technologies, programming languages, and tools are going to be used, bearing in mind that external requirements can seriously limit your options.

In this book, we're creating a web site using PHP 5, MySQL 5, and related technologies. We really like these technologies, but it doesn't necessarily mean they're the best choice for any kind of project, in any circumstances. Additionally, there are many situations in which you must use specific technologies because of client requirements. The Requirements Analysis stage that is present in most software development process will determine which technologies you must use for creating the application.

Although the book assumes some previous experience with PHP and MySQL, we'll take a quick look at them and see how they fit into our project and into the three-tier architecture.

Using PHP to Generate Dynamic Web Content

PHP is an open source technology for building dynamic, interactive web content. Its short description (on the official PHP web site, `http://www.php.net`) is "PHP is a widely-used general-purpose scripting language that is especially suited for web development and can be embedded into HTML."

PHP stands for PHP: Hypertext Preprocessor (yes, it's a recursive acronym) and is available for free download at its official web site. The story of PHP, having its roots somewhere in 1994, is a successful one. Among the factors that led to its success are the following:

- PHP is free; especially when combined with Linux server software, PHP can prove to be a very cost-efficient technology to build dynamic web content.
- PHP has a shorter learning curve than other scripting languages.
- The PHP community is agile. Many useful helper libraries or new versions of the existing libraries are being developed (such as those you can find in the PEAR repository or at `http://www.phpclasses.org`), and new features are added frequently.
- PHP works very well on a variety of web servers and operating systems (Unix-like platforms, Windows, and Mac OS).

However, PHP is not the only server-side scripting language around for creating dynamic web pages. Among its most popular competitors are JavaServer Pages (JSP), Perl, ColdFusion, and ASP.NET. Among these technologies are many differences but also some fundamental similarities. For example, pages written with any of these technologies are composed of basic HTML, which draws the static part of the page (the template), and code that generates the dynamic part.

■ **Note** You might want to check out *Beginning ASP.NET 2.0 E-Commerce in C# 2005* (Cristian Darie and Karli Watson. Apress, 2005.), which explains how to build e-commerce web sites with ASP.NET 2.0, C#, and SQL Server 2005.

Using Smarty to Separate Layout from Code

Because PHP is simple and easy to start with, it has always been tempting to start coding without properly designing an architecture and framework that would be beneficial in the long run.

What makes things even worse is that the straightforward method of building PHP pages is to mix PHP instructions with HTML because PHP doesn't have, by default, an obvious technique of separating the PHP code from the HTML layout information.

Mixing the PHP logic with HTML has two important disadvantages:

- This technique often leads to long, complicated, and hard-to-manage code. Maybe you have seen those kilometric source files with an unpleasant mixture of PHP and HTML, which are hard to read and impossible to understand after a week.
- These mixed files are the subject of both designers' and programmers' work, which complicates the collaboration more than necessary. This also increases the chances of the designer creating bugs in the code logic while working on cosmetic changes.

These kinds of problems led to the development of template engines, which offer frameworks separating the presentation logic from the static HTML layout. Smarty (`http://smarty.php.net`) is the most popular and powerful template engine for PHP. Its main purpose is to offer you a simple way to separate application logic (PHP code) from its presentation code (HTML).

This separation permits the programmer and the template designer to work independently on the same application. The programmer can change the PHP logic without needing to change the template files, and the designer can change the templates without caring how the code that makes them alive works.

Figure 2-4 shows the relationship between the Smarty design template file and its Smarty plug-in file.

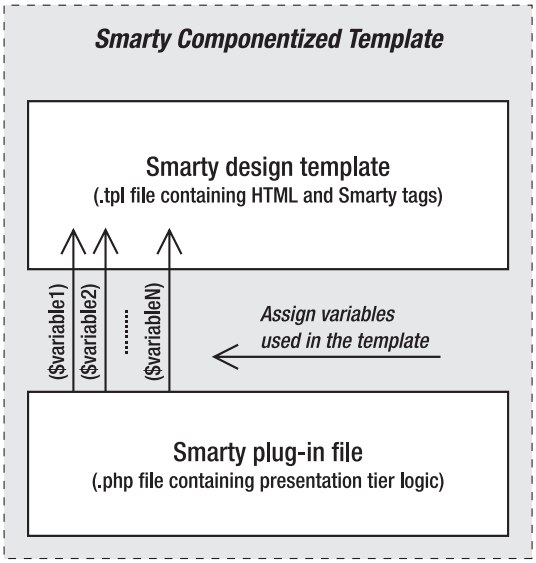

Figure 2-4. *Smarty componentized template*

The Smarty design template (a `.tpl` file containing the HTML layout and Smarty-specific tags and code) and its Smarty plug-in file (a `.php` file containing the associated code for the template) form a *Smarty componentized template*.

In practice, we'll not create a Smarty plug-in file for each template, as shown in Figure 2-4. Instead, we'll create a generic Smarty plug-in that integrates with all your Smarty templates, loading the necessary presentation objects. Presentation objects are classes that provide the template files with the data they need.

You'll learn more about how Smarty works while you're building the e-commerce web site. For a concise introduction to Smarty, read the Smarty Crash Course at `http://smarty.php.net/crashcourse.php`. For a detailed reference, we recommend *Smarty PHP Template Programming and Applications* (Hasin Hayder, J. P. Maia, and Lucian Gheorghe. Packt Publishing, 2006.).

■ **Note** Adding Smarty or another templating engine to a web application's architecture adds some initial coding effort and also implies a learning curve. However, you should try it anyway, because the advantages of using such a modern development technique will prove to be significant later in the process.

What About the Alternatives?

Smarty is not the only template engine available for PHP. You can find many others by Googling for "PHP template engines." You can find the most popular of them nicely listed by Justin Silverton in his article "Top 25 PHP Template Engines" (your favorite search engine will help you, once again, find the article).

Although all template engines follow the same basic principles, we chose to use Smarty in the PHP e-commerce project for this book because of its very good performance results, powerful features (such as template compilation and caching), and wide acceptance in the industry.

Using MySQL to Store Web Site Data

Most of the data your visitors will see while browsing the web site will be retrieved from a relational database. A relational database management system (RDBMS) is a complex software program, the purpose of which is to store, manage, and retrieve data as quickly and reliably as possible. For the TShirtShop web site, it will store all data regarding the products, departments, users, shopping carts, and so on.

Many RDBMSs are available for you to use with PHP, including MySQL, PostgreSQL, Oracle, and so on. However, both formal surveys and real-world practice show MySQL is truly the leading database choice for PHP-driven projects.

MySQL is the world's most popular open source database, and it's a free (for noncommercial use), fast, and reliable database. Another important advantage is that many web hosting providers offer access to a MySQL database, which makes your life easier when going live with your newly created e-commerce web site. We'll use MySQL as the back-end database when developing the TShirtShop e-commerce web site.

The language used to communicate with a relational database is SQL (SQL Query Language, or, according to older specifications, Structured Query Language). However, each database engine recognizes a particular dialect of this language. If you decide to use a different RDBMS than MySQL, you'll probably need to update some of the SQL queries.

Getting in Touch with MySQL

You talk with the database server by formulating an SQL query, sending it to the database engine, and retrieving the results. The SQL query can say anything related to the web site data, or its data structures, such as "give me the list of departments," "remove product number 223," "create a data table," or "search the catalog for yellow t-shirts."

No matter what the SQL query says, we need a way to send it to MySQL. MySQL ships with a simple, text-based interface (named mysql) that permits executing SQL queries and gets back the results. If you find it difficult to use, don't worry; there are alternatives to the command-line interface. Several free, third-party database administration tools allow you to manipulate data structures and execute SQL queries via an easy-to-use graphical interface. Many web-hosting companies offer database access through phpMyAdmin (which is the most widely used MySQL web client interface), which is another good reason for you to get familiar with this tool. However, you can use the visual client of your choice. A popular desktop tool for interacting with MySQL databases is Toad for MySQL (`http://www.quest.com/toad-for-mysql/`).

Apart from needing to interact with MySQL via a direct interface to its engine, you also need to learn how to access MySQL programmatically from PHP code. This requirement is obvious, because the e-commerce web site will need to query the database to retrieve catalog information (departments, categories, products, and so on) when building pages for the visitors.

As for querying MySQL databases through PHP code, the tool you'll rely on here is the PHP Data Objects (PDO) extension.

Implementing Database Integration Using PDO

PDO (*PHP Data Objects*) is a native data-access abstraction library that ships with PHP starting from version 5.1 and is offered as a PECL extension for PHP 5.0 (PECL is a repository of PHP extensions, located at `http://pecl.php.net/`). The official PDO manual, together with installation instructions, is available at `http://php.net/pdo`.

PDO offers a uniform way to access a variety of data sources. Using PDO increases your application's portability and flexibility, because if the back-end database changes, the effects on your data-access code are kept to a minimum (in many cases, all that needs to change is the connection string for the new database).

After you become familiar with the PDO data-access abstraction layer, you can use the same programming techniques on other projects that might require a different database solution.

To demonstrate the difference between accessing the database using the old PHP functions and PDO, let's take a quick look at two short PHP code snippets.

■ **Note** If you aren't familiar with how the code works, don't worry—we'll analyze everything in greater detail in the following chapters.

The following shows database access using PHP native (MySQL-specific) functions:

```
// Connecting to MySQL
$link = mysql_connect('localhost', $username, $password);

if (!$link)
{
   die ('Could not connect: ' . mysql_error());
}

$db_selected = mysql_select_db('tshirtshop', $link);

if (!$db_selected)
{
   die ('Could not select database : ' . mysql_error());
}

// Execute SQL query
$queryString = 'SELECT * FROM product';

$result = mysql_query($queryString);

if (!$result)
{
   die ('Query failed : ' . mysql_error());
}

// Close connection
mysql_close($link);
```

■ **Note** If you are still planning to use PHP MySQL extension instead of PDO in your projects, you should consider using the PHP MySQL improved extension (mysqli). You can find more details about the PHP MySQL improved extension at `http://www.php.net/manual/en/ref.mysqli.php`.

Next, the same action is shown, this time using PDO:

```
try
{
  // Create a new PDO instance
  $database_handler =
    new PDO('mysql:host=localhost;dbname=tshirtshop',
            $username, $password);

  // Build the SQL query
  $sqlQuery = 'SELECT * FROM product';

  // Execute SQL query
  $statement_handler = $database_handler->query($sqlQuery);

  // Fetch data
  $result = $statement_handler->fetchAll(PDO::FETCH_ASSOC);

  // Clear the PDO object instance
  $database_handler = null;
}
catch (PDOException $e)
{
  /* If something goes wrong we catch the exception thrown by
     the object, print the message, and stop the execution of
     script */
  print 'Error! <br />' . $e->getMessage() . '<br />';

  exit;
}
```

The version of the code that uses PDO is longer, but it includes a powerful error-handling mechanism—a very helpful tool when debugging your application If these concepts sound foreign, once again, wait until the later chapters where we'll put PDO to work, and you'll learn more about it there.

Also, when using PDO, you won't need to change the data access code if, for example, you decide to use PostgreSQL instead of MySQL. On the other hand, the first code snippet, which uses MySQL-specific functions, would need to change completely (use `pg_connect` and `pg_query` instead of `mysql_connect` and `mysql_query`, and so on). In addition, some PostgreSQL-specific functions have different parameters than the similar MySQL functions.

When using a database abstraction layer (such as PDO), you'll probably only need to change the connection string when changing the database back end. Note that here we're only talking

about the PHP code that interacts with the database. In practice, you might also need to update some SQL queries if the database engines support different dialects of SQL.

■ **Note** To keep your SQL queries as portable as possible, keep their syntax as close as possible to the SQL-92 standard. You'll learn more about SQL details in Chapter 4.

MySQL and the Three-Tier Architecture

It is clear by now that MySQL is somehow related to the data tier. However, if you haven't worked with databases until now, it might be less than obvious that MySQL is more than a simple store of data. Apart from the actual data stored inside, MySQL is also capable of storing logic in the form of stored procedures, to maintain table relationships, to ensure various data integrity rules are obeyed, and so on.

You can communicate with MySQL through SQL, which is a language used to interact with the database. SQL is used to transmit to the database instructions such as "send me the last 10 orders" or "delete product number 123."

Although it's possible to compose SQL statements in your PHP code and then submit them for execution, this is generally a *bad practice*, because it incurs security, consistency, and performance penalties. In our solution, we'll store all data tier logic using *database functions*.

The code presented in this book was tested with MySQL 5.0 and MySQL 5.1. The role of the MySQL server in the three-tier architecture is described in Figure 2-5.

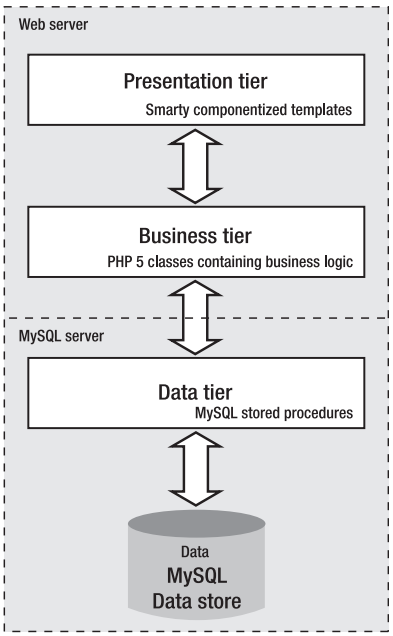

Figure 2-5. *The technologies you'll use to develop TShirtShop*

Choosing Naming and Coding Standards

Although coding and naming standards might not seem that important at first, they definitely shouldn't be overlooked. Not following a set of rules for your code will almost always result in code that's hard to read, understand, and maintain. On the other hand, when you follow a consistent way of coding, you can almost say your code is already half documented, which is an important contribution toward the project's maintainability, especially when multiple people are working on the same project at the same time.

■ **Tip** Some companies have their own policies regarding coding and naming standards, whereas in other cases, you'll have the flexibility to use your own preferences. In either case, the golden rule to follow is *be consistent in the way you code*. Commenting your code is another good practice that improves the long-term maintainability of your code.

Naming conventions refer to many elements within a project, simply because almost all of a project's elements have names: the project itself, files, classes, variables, methods, method parameters, database tables, database columns, and so on. Without some discipline when naming all those elements, after a week of coding, you won't understand a single line of what you've written.

When developing TShirtShop, we followed a set of naming conventions that are popular among PHP developers. Some of the most important rules are summarized here and in the piece of code that follows:

- Class names and method names should be written using Pascal casing (uppercase letters for the first letter in every word), such as `WarZone`.
- Public class attribute names follow the same rules as class names but should be prepended with the character "m". So, valid public attribute names look like this: `$mSomeSoldier`.
- Private class attribute names follow the same rules as public class attribute names, except they're also prepended with an underscore, such as in `$_mSomeOtherSoldier`.
- Method argument names should use camel casing (uppercase letters for the first letter in every word except the first one), such as `$someEnemy`, `$someOtherEnemy`.
- Variable names should be written in lowercase, with an underscore as the word separator, such as `$master_of_war`.
- Database objects use the same conventions as variable names (the `department_id` column).
- Try to indent your code using a fixed number of spaces (say, four) for each level. (The code in this book uses two spaces because of physical space limitations.)

Here's a sample code snippet:

```
class WarZone
{
  public  $mSomeSoldier;
  private $_mSomeOtherSoldier;

  function SearchAndDestroy($someEnemy, $someOtherEnemy)
  {
    $master_of_war = 'Soldier';

    $this->mSomeSoldier = $someEnemy;
    $this->_mSomeOtherSoldier = $someOtherEnemy;
  }
}
```

Among the decisions that need to be made is whether to use quotes for strings. JavaScript, HTML, and PHP allow using both single quotes and double quotes. For the code in this book, we'll use double quotes in HTML and JavaScript code, and we'll use single quotes in PHP. Although for JavaScript it's a matter of taste (you can use single quotes, as long as you use them consistently), in PHP, the single quotes are processed faster, are more secure, and are less likely to cause programming errors. Learn more about PHP strings at `http://php.net/types.string`. You can find two useful articles on PHP strings at `http://www.sitepoint.com/print/quick-php-tips` and `http://www.jeroenmulder.com/weblog/2005/04/php_single_and_double_quotes.php`.

Summary

Hey, we covered a lot of ground in this chapter, didn't we? We talked about the three-tier architecture and how it helps you create great flexible and scalable applications. We also saw how each of the technologies used in this book fits into the three-tier architecture.

If you feel overwhelmed, please don't worry. In the next chapter, we will begin to create the first part of our site. We will explain each step as we go, so you will have a clear understanding of each element of the application.