In this chapter, we discuss REST services and how to keep them secure.

REST (REpresentational State Transfer) is an architecture that allows clients to request information from a server and then receive appropriate responses in different data formats. The responses themselves are representations of resources, such as a list of status updates from a Twitter user’s timeline, that can be delivered as XML or JSON strings.

It’s important to note that when we talk about REST in this chapter, we are specifically referring to server-to-server communication or application-to-server communication, not merely a human being accessing an arbitrary URL and getting back a response (i.e., the way that the Web operates currently for the vast majority of casual consumers).

What Is REST?

In a typical REST architecture, a client sends a request to the server, which responds with a representation of the requested resource. A resource can be almost any informational object, like a database or a document, and its representation is usually a formatted document (often XML or JSON) that acts as a snapshot of its current or requested state.

REST resources are typically identified using meaningful URLs that accept different request “verbs”—GET, POST, PUT, and DELETE. These verbs are somewhat analogous to the create-retrieve-update-delete (CRUD) model that many developers are familiar with.

For example, if you want to retrieve data safely (in other words, with idempotence, or not changing anything while doing so), use a GET request; to create data, use a POST request; to update data, use a PUT request; and finally, to delete data, use a DELETE request.

Another important factor to consider is the response. A RESTful service typically provides two meaningful components in its responses: the response body itself and a status code. Many REST servers actually allow users to specify a response format (such as JSON, XML, CSV, serialized PHP, or plain text) either by sending in an ACCEPT parameter or by specifying a file extension (for example, /api/users.xml or /api/users.json). Other REST servers, like the one you’re going to implement here, have hard-coded response formats. These are equally acceptable as long as they are documented.

Response codes tend to be HTTP status codes. The beauty of this schema is that you can use existing, well-known status codes to identify errors or successes. A 201 status code (CREATED) is a perfect response to a successful POST request. A 500 error code indicates a failure on your end, but a 400 error code indicates a failure on the client’s end (BAD REQUEST). Send a 503 (SERVICE UNAVAILABLE) if something is wrong with the server. A long list of HTTP 1.1 status codes is available at http://en.wikipedia.org/wiki/List_of_HTTP_status_codes.
What Is JSON?

JSON, which stands for Javascript Object Notation, is a bit beyond the scope of this book, but it’s important to take a quick detour to discuss it, as it seems to be emerging as the most relevant format for REST services. So what is JSON? JSON is a lightweight text-based data interchange format that is easy for both humans and computers to digest and consume. At its inception, JSON was designed to represent simple data structures. Although it was originally conceived as a way to transmit JavaScript-friendly data in particular, parsers are now available for it in virtually every computer language. In PHP, a pair of native JSON functions help you do a lot of heavy lifting (json_encode and json_decode). Simply send in an array of data (or even a simple string) to json_encode, and a JSON object will emerge.

```php
$data = array(
    'firstname' => 'Tom',
    'lastname' => 'Smith',
    'age' => 40
);
print_r($data); /* prints Array([firstname] => Tom [lastname] => Smith [age] => 40) */
echo json_encode($data); /* prints { "firstname": "Tom", "lastname": "Smith", "age":40 */
```

Notice that the typical PHP array that results from an SQL query (where the keys match database field names and the values match the data) is easily transported as a JSON object. Upon arrival, the data can simply be evaluated with eval() (for example, from within an Ajax context), or it can be decoded with json_decode() in PHP to be remade into a data array.

JSON data supports various data types besides objects: strings, null values, numbers (integer or real), Boolean values, and arrays (comma-separated sequences of values enclosed in square brackets). As a result, JSON users experience much flexibility when working with data.

REST Security

The best thing about REST is its lightweight approach to exposing APIs and resources. Typically, if you want to allow an application to list out the contents of a database table (for example), you can provide a RESTful address that accepts a GET request and then reply with a JSON or XML string.