# 1

# AJAX and the Future of Web Applications

"Computer, draw a robot!" said my young cousin to the first computer he had ever seen. (Since I had instructed it not to listen to strangers, the computer wasn't receptive to this command.) If you're like me, your first thought would be "how silly" or "how funny"—but this is a mistake. Our educated and modeled brains have learned how to work with computers to a certain degree. People are being educated to accommodate computers, to compensate for the lack of ability of computers to understand humans. (On the other hand, humans can't accommodate very well themselves, but that's another story.)

This little story is relevant to the way people instinctively work with computers. In an ideal world, that spoken command should have been enough to have the computer please my cousin. The ability of technology to be user-friendly has evolved very much in the past years, but there's still a long way till we have real intelligent computers. Until then, people need to learn how to work with computers—some to the extent that they end up loving a black screen with a tiny command prompt on it.

Not incidentally, the computer-working habits of many are driven by software with user interfaces that allow for intuitive (and enjoyable) human interaction. This probably explains the popularity of the right mouse button, the wonder of fancy features such as drag and drop, or that simple text box that searches content all over the Internet for you in just 0.1 seconds (or so it says). The software industry (or the profitable part of it, anyway) has seen, analyzed, and learned. Now the market is full of programs with shiny buttons, icons, windows, and wizards, and people are *paying a lot of money for them*.

What the software industry has learned is that the equivalent of a powerful engine in a red sports car is *usability and accessibility* for software. And it's wonderful when what is good from the business point of view is also good from a human point of view, because the business profits are more or less proportional to customers' satisfaction.

We plan to be very practical and concise in this book, but before getting back to your favorite mission (writing code) it's worth taking a little step back, just to remember what we are doing and why we are doing it. We love technology to the sound made by each key stroke, so it's very easy to forget that the very reason technology exists is to serve people and make their lives at home more entertaining, and at work more efficient.

Understanding the way people's brains work would be the key to building the ultimate software applications. While we're far from that point, what we do understand is that end users need intuitive user interfaces; they don't really care what operating system they're running as long as the functionality they get is what they *expect*. This is a very important detail to keep in mind, as many programmers tend to think and speak in technical terms even when working with end users (although in a typical development team the programmer doesn't interact directly with the end user). If you disagree, try to remember how many times you've said the word *database* when talking to a non-technical person.

By observing people's needs and habits while working with computer systems, the term **software usability** was born—referring to the *art* of meeting users' interface expectations, understanding the nature of their work, and building software applications accordingly.

Historically, usability techniques were applied mainly to **desktop applications**, simply because the required tools weren't available for **web applications**. However, as the Internet gets more mature, the technologies it enables are increasingly potent.

Modern Internet technologies not only enable you to build a better online presence, but also allow building better intranet/dedicated applications. Having friendly websites is crucial for online business, because *the Internet never sleeps*, and customers frequently migrate to the next "big thing" that looks better or *feels* to move faster. At the same time, being able to build friendly web interfaces gives alternative options for intranet software solutions, which were previously built mainly as desktop applications.

Building user-friendly software has always been easier with desktop applications than with web applications, simply because the Web was designed as a means for delivering text and images, and not complex functionality. This problem has gotten significantly more painful in the last few years, when more and more software services and functionality are delivered via the Web.

Consequently, many technologies have been developed (and are still being developed) to add flashy lights, accessibility, and power to web applications. Notable examples include **Java applets** and **Macromedia Flash**, which require the users to install separate libraries into their web browsers.

# Delivering Functionality via the Web

Web applications are applications whose functionality is processed on a web server, and is delivered to the end users over a network such as the Internet or an intranet. The end users use a **thin client** (web browser) to run web applications, which knows how to display and execute the data received from the server. In contrast, desktop applications are based on a **thick client** (also called a rich client or a fat client), which does most of the processing.

Web applications evolve dreaming that one day they'll look and behave like their mature (and powerful) relatives, the desktop applications. The behavior of any computer software that interacts with humans is now even more important than it used to be, because nowadays the computer user base varies much more than in the past, when the users were technically sound as well. Now you need to display good looking reports to Cindy, the sales department manager, and you need to provide easy-to-use data entry forms to Dave, the sales person.

Because end-user satisfaction is all that matters, the software application you build must be satisfactory to all the users that interact with it. As far as web applications are concerned, their evolution-to-maturity process will be complete when the application's interface and behavior will not reveal whether the functionality is delivered by the local desktop or comes through fiber or air. Delivering usable interfaces via the Web used to be problematic simply because features that people use with their desktop application, such as drag and drop, and performing multiple tasks on the same window at the same time, were not possible.

Another problem with building web applications is **standardization**. Today, everything web-accessible must be verified with at least two or three browsers to ensure that all your visitors will get the full benefit of your site.

## Advantages of Web Applications

Yes, there are lots of headaches when trying to deliver functionality via the Web. But why bother trying to do that in the first place, instead of building plain desktop applications? Well, even with the current problems that web applications have with being user-friendly, they have acquired extraordinary popularity because they offer a number of major technological advantages over desktop applications.

- **Web applications are easy and inexpensive to deliver**. With web applications, a company can reduce the costs of the IT department that is in charge of installing the software on the users' machines. With web applications, all that users need is a computer with a working web browser and an Internet or intranet connection.
- **Web applications are easy and inexpensive to upgrade**. Maintenance costs for software have always been significant. Because upgrading an existing piece of software is similar to installing a new one, the web applications' advantages mentioned above apply here as well. As soon as the application on the server machine is upgraded, everyone gets the new version.
- **Web applications have flexible requirements for the end users**. Just have your web application installed on a server—any modern operating system will do—and you'll be able to use it over the Internet/Intranet on any Mac, Windows, or Linux machine and so on. If the application is properly built, it will run equally well on any modern web browser, such as Internet Explorer, Mozilla Firefox, Opera, or Safari.
- **Web applications make it easier to have a central data store**. When you have several locations that need access to the same data, having all that data stored in one place is much easier than having separate databases in each location. This way you avoid potential data synchronization operations and lower security risks.

In this book we'll further investigate how to use modern web technologies to build better web applications, to make the most out of the possibilities offered by the Web. But before getting into the details, let's take a *short* history lesson.

# Building Websites Since 1990

Although the history of the Internet is a bit longer, 1991 is the year when **HyperText Transfer Protocol (HTTP**), which is still used to transfer data over the Internet, was invented. In its first few initial versions, it didn't do much more than opening and closing connections. The later versions of HTTP (version 1.0 appeared in 1996 and version 1.1 in 1999) became the protocol that now we all know and use.

## HTTP and HTML

HTTP is supported by all web browsers, and it does very well the job it was conceived for—retrieving simple web content. Whenever you request a web page using your favorite web browser, the HTTP protocol is assumed. So, for example, when you type `www.mozilla.org` in the location bar of Firefox, it will assume by default that you meant `http://www.mozilla.org`.

The standard document type of the Internet is **HyperText Markup Language (HTML**), and it is built of markup that web browsers *understand*, *parse*, and *display*. HTML is a language that describes documents' formatting and content, which is basically composed of static text and images. HTML wasn't designed for building complex web applications with interactive content or user-friendly interfaces. When you need to get to another HTML page via HTTP, you need to initiate a full page reload, and the HTML page you requested must exist at the mentioned location, as a static document, prior to the request. It's obvious that these restrictions don't really encourage building anything interesting.

Nevertheless, HTTP and HTML are still a very successful pair that both web servers and web clients (browsers) understand. They are the foundation of the Internet as we know it today. Figure 1.1 shows a simple transaction when a user requests a web page from the Internet using the HTTP protocol:

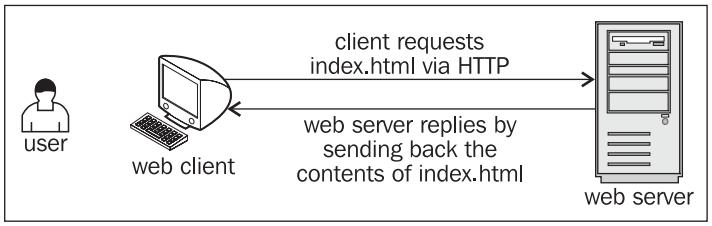


Figure 1.1: A Simple HTTP Request

Three points for you to keep in mind:

1. HTTP transactions always happen between a *web client* (the software making the request, such as a web browser) and a *web server* (the software responding to the request, such as **Apache** or **IIS**). *From now on in this book, when saying 'client' we refer to the web client, and when saying 'server' we refer to the web server.*
2. The user is the person using the client.
3. Even if HTTP (and its secure version, **HTTPS**) is arguably the most important protocol used on the Internet, it is not the only one. Various kinds of web servers use different protocols to accomplish various tasks, usually unrelated to simple web browsing. The protocol we'll use most frequently in this book is HTTP, and when we say 'web request' we'll assume a request using HTTP protocol, unless other protocol will be mentioned explicitly.

Sure thing, the HTTP-HTML combination is very limited in what it can do—it only enables users to retrieve static content (HTML pages) from the Internet. To complement the lack of features, several technologies have been developed.

While all web requests we'll talk about from now on still use the HTTP protocol for transferring the data, the data itself can be built dynamically on the web server (say, using information from a database), and this data can contain more than plain HTML allowing the client to perform some functionality rather than simply display static pages.

The technologies that enable the Web to act smarter are grouped in the following two main categories:

- **Client-side technologies** enable the web client to do more interesting things than displaying static documents. Usually these technologies are extensions of HTML, and don't replace it entirely.
- **Server-side technologies** are those that enable the server to store logic to build web pages on the fly.

## PHP and Other Server-Side Technologies

Server-side web technologies enable the web server to do much more than simply returning the requested HTML files, such as performing complex calculations, doing object-oriented programming, working with databases, and much more.

Just imagine how much data processing Amazon must do to calculate personalized product recommendations for each visitor, or Google when it searches its enormous database to serve your request. Yes, server-side processing is the engine that caused the web revolution, and the reason for which Internet is so useful nowadays.

The important thing to remember is that no matter what happens on the server side, the response received by the client must be a language that the client understands (obviously)—such as HTML, which has many limits, as mentioned earlier.

**PHP** is one of the technologies used to implement server-side logic. Chapter 3 will serve an introduction to PHP, and we'll use PHP in this book when building the **AJAX** case studies. It's good to know, though, that PHP has many competitors, such as **ASP.NET** (Active Server Pages, the web development technology from Microsoft), **Java Server Pages** (**JSP**), **Perl**, **ColdFusion**, **Ruby on Rails**, and others. Each of these has its own way of allowing programmers to build server-side functionality.

PHP is not only a server-side technology but a scripting language as well, which programmers can use to create PHP scripts. Figure 1.2 shows a request for a PHP page called `index.php`.This time, instead of sending back the contents of `index.php`, the server executes `index.php` and sends back the results. These results must be in HTML, or in other language that the client understands.

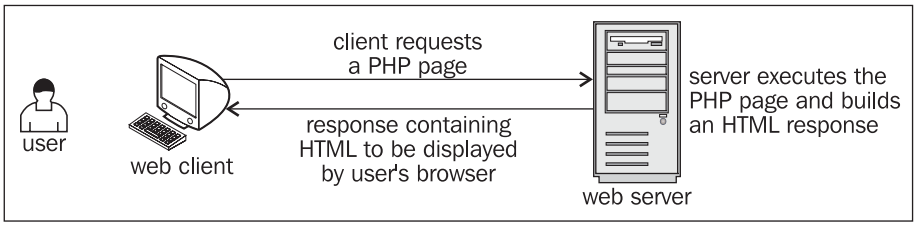


Figure 1.2: Client Requests a PHP Page

On the server side you'll usually need a **database server** as well to manage your data. In the case studies of this book we'll work with **MySQL**, but the concepts are the same as any other server. You'll learn the basics of working with databases and PHP in Chapter 3.

However, even with PHP that can build custom-made database-driven responses, the browser still displays a static, boring, and not very smart web document.

The need for smarter and more powerful functionality on the web client generated a separated set of technologies, called client-side technologies. Today's browsers know how to parse more than simple HTML. Let's see how.

## JavaScript and Other Client-Side Technologies

The various client-side technologies differ in many ways, starting with the way they get loaded and executed by the web client. **JavaScript** is a scripting language, whose code is written in plain text and can be embedded into HTML pages to empower them. When a client requests an HTML page, that HTML page can contain JavaScript. JavaScript is supported by all modern web browsers without requiring users to install new components on the system.

JavaScript is a language in its own right (theoretically it isn't tied to web development), it's supported by most web clients under any platform, and it has some object-oriented capabilities. JavaScript is not a compiled language so it's not suited for intensive calculations or writing device drivers and it must arrive in one piece at the client browser to be interpreted so it is not secure either, but it does a good job when used in web pages.

With JavaScript, developers could finally build web pages with snow falling over them, with client-side form validation so that the user won't cause a whole page reload (incidentally losing all typed data) if he or she forgot to supply all the details (such as password, or credit card number), or if the email address had an incorrect format. However, despite its potential, JavaScript was never used consistently to make the web experience truly user friendly, similar to that of users of desktop applications.

Other popular technologies to perform functionality at the client side are Java applets and Macromedia Flash. Java applets are written in the popular and powerful Java language, and are executed through a **Java Virtual Machine** that needs to be installed separately on the system. Java applets are certainly the way to go for more complex projects, but they have lost the popularity they once had over web applications because they consume many system resources. Sometimes they even need long startup times, and are generally too heavy and powerful for the small requirements of simple web applications.

Macromedia Flash has very powerful tools for creating animations and graphical effects, and it's the de-facto standard for delivering such kind of programs via the Web. Flash also requires the client to install a browser *plug-in*. Flash-based technologies become increasingly powerful, and new ones keep appearing.

Combining HTML with a server-side technology and a client-side technology, one can end up building very powerful web solutions.

## What's Been Missing?

So there are options, why would anyone want anything new? What's missing?

As pointed out in the beginning of the chapter, technology exists to serve existing market needs. And part of the market wants to deliver more powerful functionality to web clients without using Flash, Java applets, or other technologies that are considered either too flashy or heavy-weight for certain purposes. For these scenarios, developers have usually created websites and web applications using HTML, JavaScript, and PHP (or another server-side technology). The typical request with this scenario is shown in Figure 1.3, which shows an HTTP request, the response made up of HTML and JavaScript built programmatically with PHP.

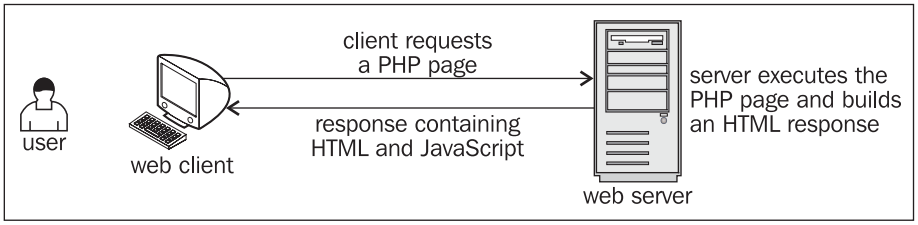


Figure 1.3: HTTP, HTML, PHP, and JavaScript in Action

The hidden problem with this scenario is that each time the client needs new data from the server, a new HTTP request must be made to reload the page, freezing the user's activity. The **page reload** is the new evil in the present day scenario, and AJAX comes in to our rescue.

# Understanding AJAX

AJAX is an acronym for **Asynchronous JavaScript and XML**. If you think it doesn't say much, we agree. Simply put, AJAX can be read "empowered JavaScript", because it essentially offers a technique for client-side JavaScript to make background server calls and retrieve additional data as needed, updating certain portions of the page without causing full page reloads. Figure 1.4 offers a visual representation of what happens when a typical AJAX-enabled web page is requested by a visitor:

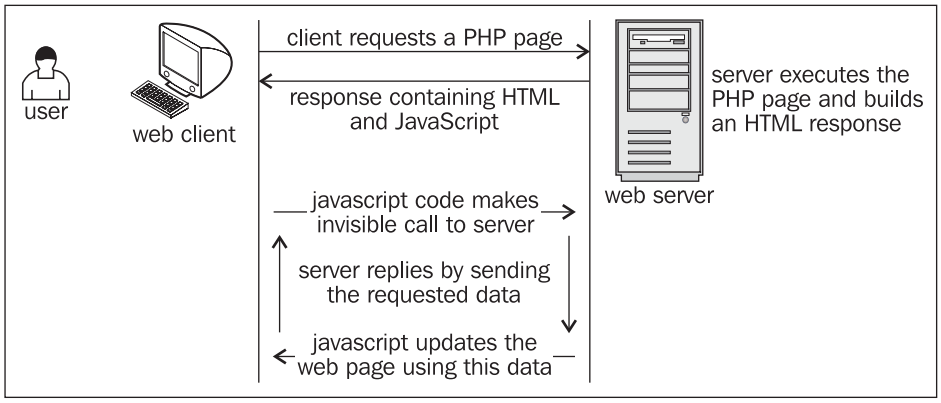


Figure 1.4: A Typical AJAX Call

When put in perspective, AJAX is about reaching a better balance between client functionality and server functionality when executing the action requested by the user. Up until now, client-side functionality and server-side functionality were regarded as separate bits of functionality that work one at a time to respond to user's actions. AJAX comes with the solution to balance the load between the client and the server by allowing them to communicate in the background *while the user is working* on the page.

To explain with a simple example, consider web forms where the user is asked to write some data (such as name, email address, password, credit card, etc) that has to be validated before reaching the business tier of your application. Without AJAX, there were two form validation techniques. The first was to let the user type all the required data, let him or her *submit* the page, and perform the validation on the server. In this scenario the user experiences a *dead time* while waiting for the new page to load. The alternative was to do this verification at the client, but this wasn't always possible (or feasible) because it implied loading too much data on the client (just think if you needed to validate that the entered city and the entered country match).

In the AJAX-enabled scenario, the web application can validate the entered data by making server calls in the background, while the user keeps typing. For example, after the user selects a country, the web browser calls the server to load on the fly the list of cities for that country, without

interrupting the user from his or her current activity. You'll find an example of AJAX form validation in Chapter 4.

The examples where AJAX can make a difference are endless. To get a better feeling and understanding of what AJAX can do for you, have a look at these live and popular examples:

- **Google Suggest** helps you with your **Google** searches. The functionality is pretty spectacular; check it out at `http://www.google.com/webhp?complete=1`. Similar functionality is offered by **Yahoo! Instant Search**, accessible at `http://instant.search.yahoo.com/`. (You'll learn how to build similar functionality in Chapter 6.)
- **GMail** (`http://www.gmail.com`). GMail is very popular by now and doesn't need any introduction. Other web-based email services such as **Yahoo! Mail** and **Hotmail** have followed the trend and offer AJAX-based functionality.
- **Google Maps** (`http://maps.google.com`), **Yahoo Maps** (`http://maps.yahoo.com`), and **Windows Live Local** (`http://local.live.com`).
- Other services, such as `http://www.writely.com` and `http://www.basecamphq.com`.

You'll see even more examples over the course of this book.

> Just as with any other technology, AJAX can be *overused*, or used the wrong way. Just having AJAX on your website doesn't guarantee your website will be better. It depends on you to make good use of the technology.

So AJAX is about creating more versatile and interactive web applications by enabling web pages to make asynchronous calls to the server transparently while the user is working. AJAX is a tool that web developers can use to create smarter web applications that behave better than traditional web applications when interacting with humans.

The technologies AJAX is made of are already implemented in all modern web browsers, such as Mozilla Firefox, Internet Explorer, or Opera, so the client doesn't need to install any extra modules to run an AJAX website. AJAX is made of the following:

- JavaScript is the essential ingredient of AJAX, allowing you to build the client-side functionality. In your JavaScript functions you'll make heavy use of the **Document Object Model** (**DOM**) to manipulate parts of the HTML page.
- The **XMLHttpRequest** object enables JavaScript to access the server asynchronously, so that the user can continue working, while functionality is performed in the background. Accessing the server simply means making a simple HTTP request for a file or script located on the server. HTTP requests are easy to make and don't cause any firewall-related problems.
- A server-side technology is required to handle the requests that come from the JavaScript client. In this book we'll use PHP to perform the server-side part of the job.

For the client-server communication the parts need a way to *pass data* and *understand* that data. Passing the data is the simple part. The client script accessing the server (using the `XMLHttpRequest` object) can send name-value pairs using **GET** or **POST**. It's very simple to read these values with any server script.

The server script simply sends back the response via HTTP, but unlike a usual website, the response will be in a format that can be simply parsed by the JavaScript code on the client. The suggested format is XML**,** which has the advantage of being widely supported, and there are many libraries that make it easy to manipulate XML documents. But you can choose another format if you want (you can even send plain text), a popular alternative to XML being **JavaScript Object Notation** (**JSON**).

This book assumes you already know the taste of the AJAX ingredients, except maybe the `XMLHttpRequest` object, which is less popular. However, to make sure we're all on the same page, we'll have a look together at how these pieces work, and how they work together, in Chapter 2 and Chapter 3. Until then, for the remainder of this chapter we'll focus on the big picture, and we will also write an AJAX program for the joy of the most impatient readers.

> None of the AJAX components is new, or revolutionary (or at least evolutionary) as the current buzz around AJAX might suggest: all the components of AJAX have existed since sometime in 1998. The name AJAX was born in 2005, in Jesse James Garret's article at `http://www.adaptivepath.com/publications/essays/archives/000385.php`, and gained much popularity when used by Google in many of its applications.
>
> What's new with AJAX is that for the first time there is enough energy in the market to encourage standardization and focus these energies on a clear direction of evolution. As a consequence, many AJAX libraries are being developed, and many AJAX-enabled websites have appeared. Microsoft through its Atlas project is pushing AJAX development as well.

AJAX brings you the following potential benefits when building a new web application:

- It makes it possible to create better and more responsive websites and web applications.
- Because of its popularity, it encourages the development of patterns that help developers avoid reinventing the wheel when performing common tasks.
- It makes use of existing technologies.
- It makes use of existing developer skills.
- Features of AJAX integrate perfectly with existing functionality provided by web browsers (say, re-dimensioning the page, page navigation, etc).

Common scenarios where AJAX can be successfully used are:

- Enabling immediate server-side form validation, very useful in circumstances when it's unfeasible to transfer to the client all the data required to do the validation when the page initially loads. Chapter 4 contains a form validation case study.

- Creating simple online chat solutions that don't require external libraries such as the Java Runtime Machine or Flash. You'll build such a program in Chapter 5.
- Building Google Suggest-like functionality, like an example you'll build in Chapter 6.
- More effectively using the power of other existing technologies. In Chapter 7, you'll implement a real-time charting solution using **Scalable Vector Graphics** (**SVG**), and in Chapter 10, you'll use an external AJAX library to create a simple drag-and-drop list.
- Coding responsive **data grids** that update the server-side database on the fly. You'll create such an application in Chapter 8.
- Building applications that need real-time updates from various external sources. In Chapter 9, you'll create a simple **RSS** aggregator.

Potential problems with AJAX are:

- Because the page address doesn't change while working, you can't easily bookmark AJAX-enabled pages. In the case of AJAX applications, bookmarking has different meanings depending on your specific application, usually meaning that you need to save state somehow (think about how this happens with desktop applications—there's no bookmarking there).
- Search engines may not be able to index all portions of your AJAX application site.
- The Back button in browsers, doesn't produce the same result as with classic web applications, because all actions happen inside the same page.
- JavaScript can be disabled at the client side, which makes the AJAX application non-functional, so it's good to have another plan in your site, whenever possible, to avoid losing visitors.

Finally, before moving on to write your first AJAX program, here are a number of links that may help you in your journey into the exciting world of AJAX:

- `http://ajaxblog.com` is an AJAX dedicated blog.
- `http://www.fiftyfoureleven.com/resources/programming/xmlhttprequest` is a comprehensive article collection about AJAX.
- `http://www.ajaxian.com` is the AJAX website of Ben Galbraith and Dion Almaer, the authors of Pragmatic AJAX.
- `http://www.ajaxmatters.com` is an informational site about AJAX, containing loads of very useful links.
- `http://ajaxpatterns.org` is about reusable AJAX design patterns.
- `http://www.ajaxinfo.com` is a resource of AJAX articles and links.
- `http://dev.fiaminga.com` contains many links to various AJAX resources and tutorials.

- `http://ajaxguru.blogspot.com` is a popular AJAX-related web blog.
- `http://www.sitepoint.com/article/remote-scripting-ajax` is Cameron Adams' excellent article *AJAX: Usable Interactivity with Remote Scripting*.
- `http://developer.mozilla.org/en/docs/AJAX` is Mozilla's page on AJAX.
- `http://en.wikipedia.org/wiki/AJAX` is the Wikipedia page on AJAX.

The list is by no means complete. If you need more online resources, Google will surely be available to help. In the following chapters, you'll be presented with even more links, but more specific to the particular technologies you'll be learning about.

# Building a Simple Application with AJAX and PHP

Let's write some code then! In the following pages you'll build a simple AJAX application.

> This exercise is for the most impatient readers willing to start coding ASAP, but it assumes you're already familiar with JavaScript, PHP, and XML. If this is not the case, or if at any time you feel this exercise is too challenging, feel free to skip to Chapter 2. In Chapter 2 and Chapter 3 we'll have a much closer look at the AJAX technologies and techniques and everything will become clear.

You'll create here a simple AJAX web application called **quickstart** where the user is requested to write his or her name, and the server keeps sending back responses while the user is writing. Figure 1.5 shows the initial page, `index.html`, loaded by the user. (Note that `index.html` gets loaded by default when requesting the `quickstart` web folder, even if the file name is not explicitly mentioned.)

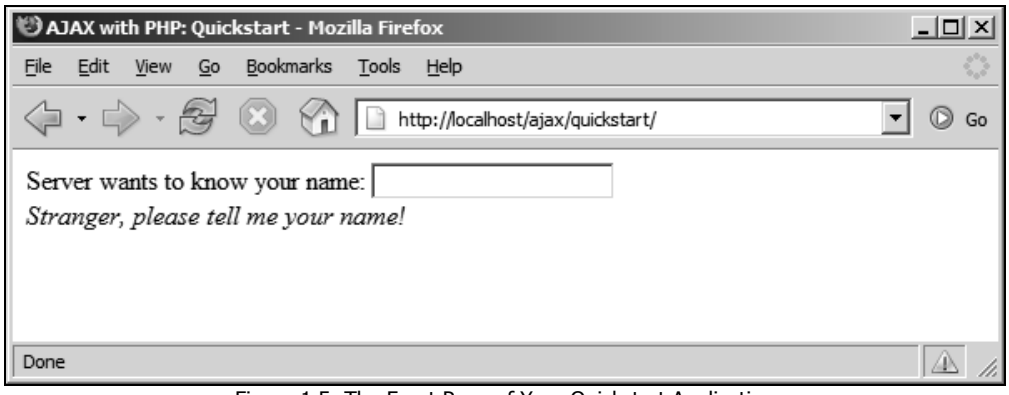


Figure 1.5: The Front Page of Your Quickstart Application

While the user is typing, the server is being called asynchronously, at regular intervals, to see if it recognizes the current name. The server is called automatically, approximately one time per second, which explains why we don't need a button (such as a 'Send' button) to notify when we're

done typing. (This method may not be appropriate for real log-in mechanisms but it's very good to demonstrate some AJAX functionality.)

Depending on the entered name, the message from the server may differ; see an example in Figure 1.6.

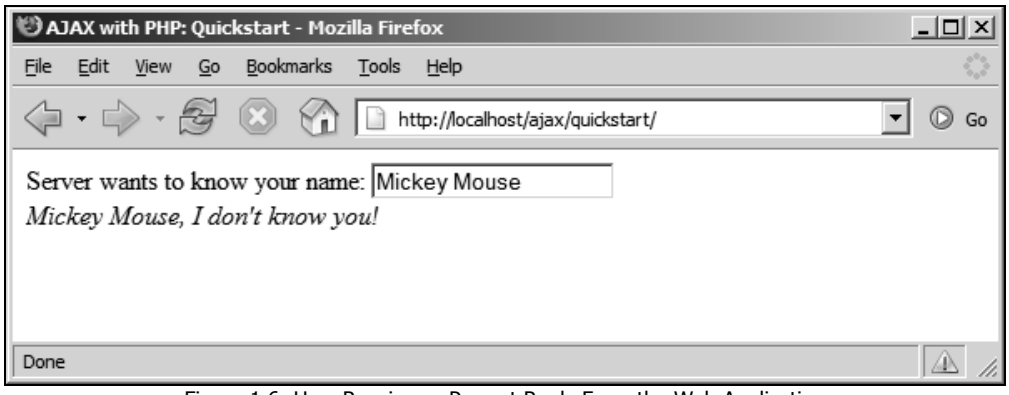


Figure 1.6: User Receives a Prompt Reply From the Web Application

Check out this example online at `http://ajaxphp.packtpub.com/ajax/quickstart`

Maybe at first sight there's nothing extraordinary going on there. We've kept this first example simple on purpose, to make things easier to understand. What's special about this application is that the displayed message comes automatically from the server, without interrupting the user's actions. (The messages are displayed as the user types a name). **The page doesn't get reloaded to display the new data, even though a server call needs to be made to get that data.** This wasn't a simple task to accomplish using non-AJAX web development techniques.

The application consists of the following three files:

1. `index.html` is the initial HTML file the user requests.
2. `quickstart.js` is a file containing JavaScript code that is loaded on the client along with `index.html`. This file will handle making the asynchronous requests to the server, when server-side functionality is needed.
3. `quickstart.php` is a PHP script residing on the server that gets called by the JavaScript code in `quickstart.js` file from the client.

Figure 1.7 shows the actions that happen when running this application:

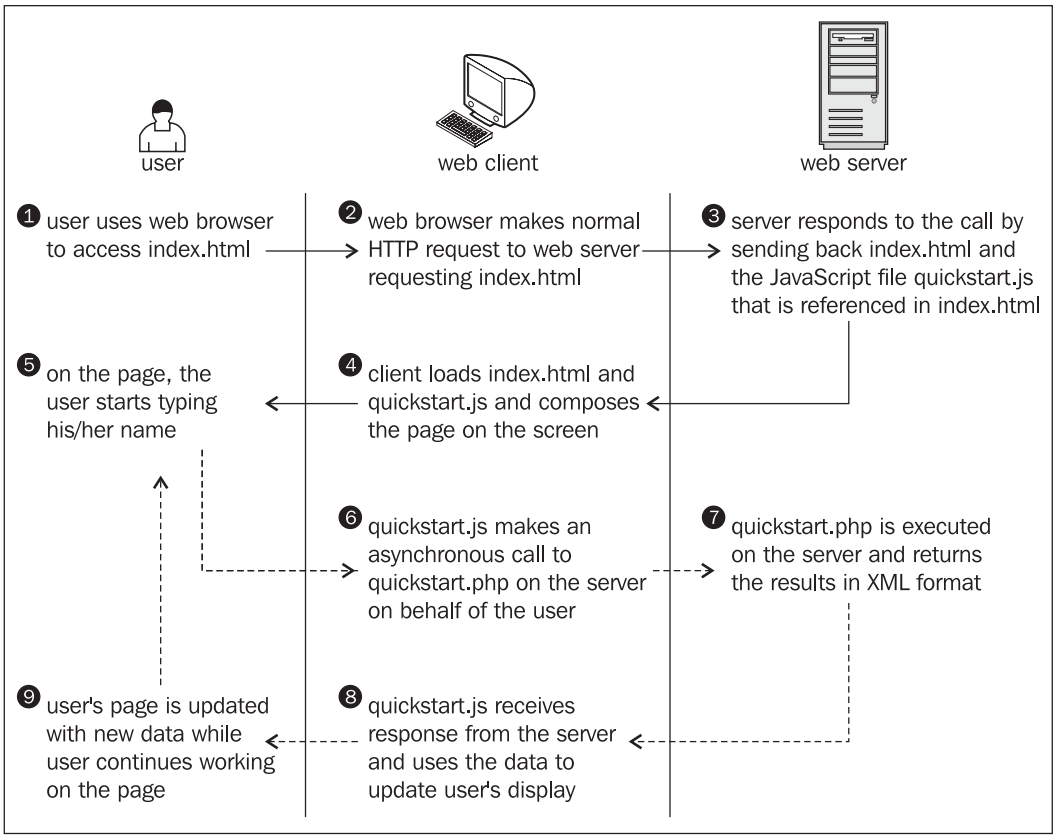


Figure 1.7: The Diagram Explaining the Inner Works of Your Quickstart Application

Steps 1 through 5 are a typical HTTP request. After making the request, the user needs to wait until the page gets loaded. With typical (non-AJAX) web applications, such a page reload happens every time the client needs to get new data from the server.

Steps 5 through 9 demonstrate an AJAX-type call—more specifically, a sequence of asynchronous HTTP requests. The server is accessed in the background using the `XMLHttpRequest` object. During this period the user can continue to use the page normally, as if it was a normal desktop application. No page refresh or reload is experienced in order to retrieve data from the server and update the web page with that data.

Now it's about time to implement this code on your machine. Before moving on, ensure you've prepared your working environment as shown in Appendix A, where you're guided through how to install and set up PHP and Apache, and set up the database used for the examples in this book. (You won't need a database for this quickstart example.)

All exercises from this book assume that you've installed your machine as shown in Appendix A. If you set up your environment differently you may need to implement various changes, such as using different folder names, and so on.

## Time for Action—Quickstart AJAX

1. In Appendix A, you're instructed to set up a web server, and create a web-accessible folder called `ajax` to host all your code for this book. Under the `ajax` folder, create a new folder called `quickstart`.
2. In the `quickstart` folder, create a file called `index.html`, and add the following code to it:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>AJAX with PHP: Quickstart</title>
    <script type="text/javascript" src="quickstart.js"></script>
  </head>
  <body onload='process()'>
    Server wants to know your name:
    <input type="text" id="myName" />
    <div id="divMessage" />
  </body>
</html>
```

3. Create a new file called `quickstart.js`, and add the following code:

```
// stores the reference to the XMLHttpRequest object
var xmlHttp = createXmlHttpRequestObject();

// retrieves the XMLHttpRequest object
function createXmlHttpRequestObject()
{
  // will store the reference to the XMLHttpRequest object
  var xmlHttp;
  // if running Internet Explorer
  if(window.ActiveXObject)
  {
    try
    {
      xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    catch (e)
    {
      xmlHttp = false;
    }
  }
  // if running Mozilla or other browsers
  else
  {
    try
    {
      xmlHttp = new XMLHttpRequest();
    }
    catch (e)
    {
      xmlHttp = false;
    }
  }
  // return the created object or display an error message
  if (!xmlHttp)
```

```
    alert("Error creating the XMLHttpRequest object.");
  else
    return xmlHttp;
}

// make asynchronous HTTP request using the XMLHttpRequest object
function process()
{
  // proceed only if the xmlHttp object isn't busy
  if (xmlHttp.readyState == 4 || xmlHttp.readyState == 0)
  {
    // retrieve the name typed by the user on the form
    name = encodeURIComponent(document.getElementById("myName").value);
    // execute the quickstart.php page from the server
    xmlHttp.open("GET", "quickstart.php?name=" + name, true);
    // define the method to handle server responses
    xmlHttp.onreadystatechange = handleServerResponse;
    // make the server request
    xmlHttp.send(null);
  }
  else
    // if the connection is busy, try again after one second
    setTimeout('process()', 1000);
}

// executed automatically when a message is received from the server
function handleServerResponse()
{
  // move forward only if the transaction has completed
  if (xmlHttp.readyState == 4)
  {
    // status of 200 indicates the transaction completed successfully
    if (xmlHttp.status == 200)
    {
      // extract the XML retrieved from the server
      xmlResponse = xmlHttp.responseXML;
      // obtain the document element (the root element) of the XML structure
      xmlDocumentElement = xmlResponse.documentElement;
      // get the text message, which is in the first child of
      // the the document element
      helloMessage = xmlDocumentElement.firstChild.data;
      // update the client display using the data received from the server
      document.getElementById("divMessage").innerHTML =
                                            '<i>' + helloMessage + '</i>';
      // restart sequence
      setTimeout('process()', 1000);
    }
    // a HTTP status different than 200 signals an error
    else
    {
      alert("There was a problem accessing the server: " +
xmlHttp.statusText);
    }
  }
}
```

4. Create a file called `quickstart.php` and add the following code to it:

```
<?php
// we'll generate XML output
header('Content-Type: text/xml');
// generate XML header
echo '<?xml version="1.0" encoding="UTF-8" standalone="yes"?>';
// create the <response> element
echo '<response>';
```

```
// retrieve the user name
$name = $_GET['name'];
// generate output depending on the user name received from client
$userNames = array('CRISTIAN', 'BOGDAN', 'FILIP', 'MIHAI', 'YODA');
if (in_array(strtoupper($name), $userNames))
  echo 'Hello, master ' . htmlentities($name) . '!';
else if (trim($name) == '')
  echo 'Stranger, please tell me your name!';
else
  echo htmlentities($name) . ', I don\'t know you!';
// close the <response> element
echo '</response>';
?>
```

5. Now you should be able to access your new program by loading `http://localhost/ajax/quickstart` using your favorite web browser. Load the page, and you should get a page like those shown in Figures 1.5 and 1.6.

> Should you encounter any problems running the application, check that you correctly followed the installation and configuration procedures as described in Appendix A. Most errors happen because of small problems such as typos. In Chapter 2 and Chapter3 you'll learn how to implement error handling in your JavaScript and PHP code.

### What Just Happened?

Here comes the fun part—understanding what happens in that code. (Remember that we'll discuss much more technical details over the following two chapters.)

Let's start with the file the user first interacts with, `index.html`. This file references the mysterious JavaScript file called `quickstart.js`, and builds a very simple web interface for the client. In the following code snippet from `index.html`, notice the elements highlighted in bold:

```
<body onload='process()'>
  Server wants to know your name:
  <input type="text" id="myName" />
  <div id="divMessage" />
</body>
```

When the page loads, a function from `quickstart.js` called `process()` gets executed. This somehow causes the `<div>` element to be populated with a message from the server.

Before seeing what happens inside the `process()` function, let's see what happens at the server side. On the web server you have a script called `quickstart.php` that builds the XML message to be sent to the client. This XML message consists of a `<response>` element that packages the message the server needs to send back to the client:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<response>
  ... message the server wants to transmit to the client ...
</response>
```

If the user name received from the client is empty, the message will be, "Stranger, please tell me your name!". If the name is Cristian, Bogdan, Filip, Mihai, or Yoda, the server responds with "Hello, master <user name>!". If the name is anything else, the message will be "<user name>, I don't know you!". So if Mickey Mouse types his name, the server will send back the following XML structure:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<response>
  Mickey Mouse, I don't know you!
</response>
```

The `quickstart.php` script starts by generating the XML document header and the opening `<response>` element:

```
<?php
// we'll generate XML output
header('Content-Type: text/xml');
// generate XML header
echo '<?xml version="1.0" encoding="UTF-8" standalone="yes"?>';
// create the <response> element
echo '<response>';
```

The highlighted header line marks the output as an XML document, and this is important because the client expects to receive XML (the **API** used to parse the XML on the client will throw an error if the header doesn't set `Content-Type` to `text/xml`). After setting the header, the code builds the XML response by joining strings. The actual text to be returned to the client is encapsulated in the `<response>` element, which is the root element, and is generated based on the name received from the client via a `GET` parameter:

```
// retrieve the user name
$name = $_GET['name'];
// generate output depending on the user name received from client
$userNames = array('CRISTIAN', 'BOGDAN', 'FILIP', 'MIHAI', 'YODA');
if (in_array(strtoupper($name), $userNames))
  echo 'Hello, master ' . htmlentities($name) . '!';
else if (trim($name) == '')
  echo 'Stranger, please tell me your name!';
else
  echo htmlentities($name) . ', I don\'t know you!';
// close the <response> element
echo '</response>';
?>
```

The text entered by the user (which is supposed to be the user's name) is sent by the client to the server using a `GET` parameter. When sending this text back to the client, we use the `htmlentities` PHP function to replace special characters with their HTML codes (such as &, or >), making sure the message will be safely displayed in the web browser eliminating potential problems and security risks.

> Formatting the text on the server for the client (instead of doing this directly at the client) is actually a bad practice when writing production code. Ideally, the server's responsibility is to send data in a generic format, and it is the recipient's responsibility to deal with security and formatting issues. This makes even more sense if you think that one day you may need to insert exactly the same text into a database, but the database will need different formatting sequences (in that case as well, a database handling script would do the formatting job, and not the server). For the quickstart scenario, formatting the HTML in PHP allowed us to keep the code shorter and simpler to understand and explain.

If you're curious to test `quickstart.php` and see what it generates, load `http://localhost/ajax/quickstart/quickstart.php?name=Yoda` in your web browser. The advantage of sending parameters from the client via `GET` is that it's very simple to emulate such a request using your web browser, since `GET` simply means that you append the parameters as name/value pairs in the URL query string. You should get something like this:

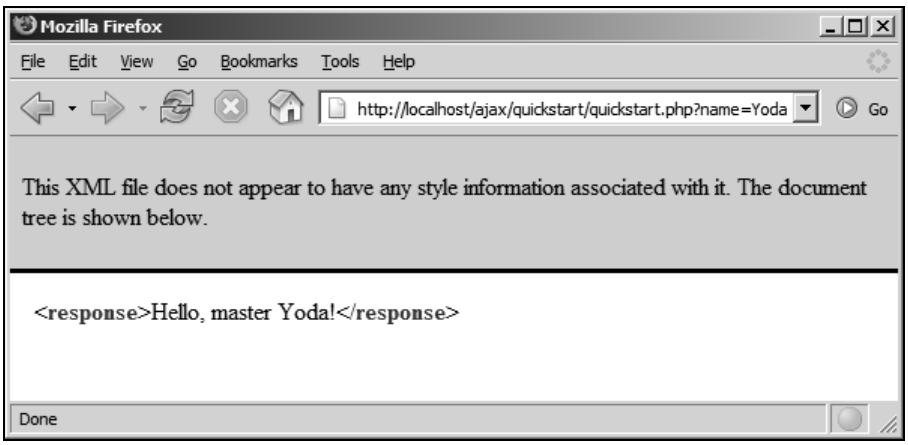


Figure 1.8: The XML Data Generated by quickstart.php

This XML message is read on the client by the `handleServerResponse()` function in `quickstart.js`. More specifically, the following lines of code extract the "Hello, master Yoda!" message:

```
// extract the XML retrieved from the server
xmlResponse = xmlHttp.responseXML;
// obtain the document element (the root element) of the XML structure
xmlDocumentElement = xmlResponse.documentElement;
// get the text message, which is in the first child of
// the document element
helloMessage = xmlDocumentElement.firstChild.data;
```

Here, `xmlHttp` is the `XMLHttpRequest` object used to call the server script `quickstart.php` from the client. Its `responseXML` property extracts the retrieved XML document. XML structures are hierarchical by nature, and the root element of an XML document is called the *document element*. In our case, the document element is the `<response>` element, which contains a single child, which is the text message we're interested in. Once the text message is retrieved, it's displayed on the client's page by using the DOM to access the `divMessage` element in `index.html`:

```
// update the client display using the data received from the server
document.getElementById('divMessage').innerHTML = helloMessage;
```

`document` is a default object in JavaScript that allows you to manipulate the elements in the HTML code of your page.

The rest of the code in `quickstart.js` deals with making the request to the server to obtain the XML message. The `createXmlHttpRequestObject()` function creates and returns an instance of the `XMLHttpRequest` object. This function is longer than it could be because we need to make it

cross-browser compatible—we'll discuss the details in Chapter 2, for now it's important to know what it does. The `XMLHttpRequest` instance, called `xmlHttp`, is used in `process()` to make the asynchronous server request:

```
// make asynchronous HTTP request using the XMLHttpRequest object
function process()
{
  // proceed only if the xmlHttp object isn't busy
  if (xmlHttp.readyState == 4 || xmlHttp.readyState == 0)
  {
    // retrieve the name typed by the user on the form
    name = encodeURIComponent(document.getElementById("myName").value);
    // execute the quickstart.php page from the server
    xmlHttp.open("GET", "quickstart.php?name=" + name, true);
    // define the method to handle server responses
    xmlHttp.onreadystatechange = handleServerResponse;
    // make the server request
    xmlHttp.send(null);
  }
  else
    // if the connection is busy, try again after one second
    setTimeout('process()', 1000);
}
```

What you see here is, actually, the heart of AJAX—the code that makes the asynchronous call to the server.

Why is it so important to call the server asynchronously? Asynchronous requests, by their nature, don't freeze processing (and user experience) while the call is made, until the response is received. Asynchronous processing is implemented by *event-driven* architectures, a good example being the way graphical user interface code is built: without events, you'd probably need to check continuously if the user has clicked a button or resized a window. Using events, the button notifies the application automatically when it has been clicked, and you can take the necessary actions in the event handler function. With AJAX, this theory applies when making a server request—you are automatically notified when the response comes back.

If you're curious to see how the application would work using a synchronous request, you need to change the third parameter of `xmlHttp.open` to `false`, and then call `handleServerResponse` manually, as shown below. If you try this, the input box where you're supposed to write your name will freeze when the server is contacted (in this case the freeze length depends largely on the connection speed, so it may not be very noticeable if you're running the server on the local machine).

```
// function calls the server using the XMLHttpRequest object
function process()
{
  // retrieve the name typed by the user on the form
  name = encodeURIComponent(document.getElementById("myName").value);
  // execute the quickstart.php page from the server
  xmlHttp.open("GET", "quickstart.php?name=" + name, false);
  // make synchronous server request (freezes processing until completed)
  xmlHttp.send(null);
  // read the response
  handleServerResponse();
}
```

The `process()` function is supposed to initiate a new server request using the `XMLHttpRequest` object. However, this is only possible if the `XMLHttpRequest` object isn't busy making another

request. In our case, this can happen if it takes more than one second for the server to reply, which could happen if the Internet connection is very slow. So, `process()` starts by verifying that it is clear to initiate a new request:

```
// make asynchronous HTTP request using the XMLHttpRequest object
function process()
{
  // proceed only if the xmlHttp object isn't busy
  if (xmlHttp.readyState == 4 || xmlHttp.readyState == 0)
  {
```

So, if the connection is busy, we use `setTimeout` to retry after one second (the function's second argument specifies the number of milliseconds to wait before executing the piece of code specified by the first argument:

```
    // if the connection is busy, try again after one second
    setTimeout('process()', 1000);
```

If the line is clear, you can safely make a new request. The line of code that prepares the server request but doesn't commit it is:

```
    // execute the quickstart.php page from the server
    xmlHttp.open("GET", 'quickstart.php?name=' + name, true);
```

The first parameter specifies the method used to send the user name to the server, and you can choose between `GET` and `POST` (learn more about them in Chapter 3). The second parameter is the server page you want to access; when the first parameter is `GET`, you send the parameters as `name/value` pairs in the query string. The third parameter is `true` if you want the call to be made asynchronously. When making asynchronous calls, you don't wait for a response. Instead, you define another function to be called *automatically* when the state of the request changes:

```
    // define the method to handle server responses
    xmlHttp.onreadystatechange = handleServerResponse;
```

Once you've set this option, you can rest calm—the `handleServerResponse` function will be executed by the system when anything happens to your request. After everything is set up, you initiate the request by calling `XMLHttpRequest`'s `send` method:

```
    // make the server request
    xmlHttp.send(null);
  }
```

Let's now look at the `handleServerResponse` function:

```
// executed automatically when a message is received from the server
function handleServerResponse()
{
  // move forward only if the transaction has completed
  if (xmlHttp.readyState == 4)
  {
    // status of 200 indicates the transaction completed successfully
    if (xmlHttp.status == 200)
    {
```

The `handleServerResponse` function is called multiple times, whenever the status of the request changes. Only when `xmlHttp.readyState` is `4` will the server request be completed so you can move forward to read the results. You can also check that the HTTP transaction reported a status of 200, signaling that no problems happened during the HTTP request. When these conditions are met, you're free to read the server response and display the message to the user.

After the response is received and used, the process is restarted using the `setTimeout` function, which will cause the `process()` function to be executed after one second (note though that it's not necessary, or even AJAX specific, to have repetitive tasks in your client-side code):

```
// restart sequence
setTimeout('process()', 1000);
```

Finally, let's reiterate what happens after the user loads the page (you can refer to Figure 1.7 for a visual representation):

1. The user loads `index.html` (this corresponds to steps 1-4 in Figure 1.7).
2. User starts (or continues) typing his or her name (this corresponds to step 5 in Figure 1.7).
3. When the `process()` method in `quickstart.js` is executed, it calls a server script named `quickstart.php` asynchronously. The text entered by the user is passed on the call as a query string parameter (it is passed via `GET`). The `handeServerResponse` function is designed to handle request state changes.
4. `quickstart.php` executes on the server. It composes an XML document that encapsulates the message the server wants to transmit to the client.
5. The `handleServerResponse` method on the client is executed multiple times as the state of the request changes. The last time it's called is when the response has been successfully received. The XML is read; the message is extracted and displayed on the page.
6. The user display is updated with the new message from the server, but the user can continue typing without any interruptions. After a delay of one second, the process is restarted from step 2.

# Summary

This chapter was all about a quick introduction to the world of AJAX. In order to proceed with learning how to build AJAX applications, it's important to understand why and where they are useful. As with any other technology, AJAX isn't the answer to all problems, but it offers means to solve some of them.

AJAX combines client-side and server-side functionality to enhance the user experience of your site. The `XMLHttpRequest` object is the key element that enables the client-side JavaScript code to call a page on the server asynchronously. This chapter was intentionally short and probably has left you with many questions—that's good! Be prepared for a whole book dedicated to answering questions and demonstrating lots of interesting functionality!