Beginning Google Web Toolkit: From Novice to Professional

Copyright © 2008 by Bram Smeets, Uri Boness, and Roald Bankras

ISBN-13 (pbk): 978-1-4302-1031-3

ISBN-13 (electronic): 978-1-4302-1032-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Massimo Nardone
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Sofia Marchant
Copy Editor: Benjamin Berg
Associate Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Octal Publishing, Inc.
Proofreader: Dan Shaw
Indexer: Julie Grady
Artist: Octal Publishing, Inc.
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 1

Introducing Rich Internet Applications (RIAs)

The Google Web Toolkit (GWT) is an open source framework that makes building RIAs easy for Java developers. You can use your Java expertise to build “fat clients” that can be deployed as web applications. These desktop-like applications are typically written in JavaScript in order to leverage the huge installed base of web browsers. But JavaScript is quite a different language from Java (its name was chosen for marketing reasons) and therefore requires different development practices.

However, GWT allows you to develop JavaScript applications in Java! This is accomplished with the most important part of GWT, the Java-to-JavaScript compiler. This compiler translates your Java code into JavaScript that runs in the user’s browser. As an added bonus, GWT copes with most browser quirks, allowing you to focus on writing code that actually does something.

This book aims to show you how to use GWT to build RIAs. But before introducing GWT in any detail, we first need to give you some historical perspective. If you’re already familiar with the inner workings of the Web, and if you know about the advantages of Ajax compared with other approaches to building RIAs (such as Flex), feel free to skip this chapter and start with Chapter 2, which introduces GWT.

This chapter provides a short history of software systems, how we typically interact with them, and how we make them available to the user. We introduce different types of applications, including RIAs, and describe Ajax as an approach to building RIAs. Finally, we compare some different approaches to developing RIAs, before zooming in on one in particular in the next chapter: GWT.

A Short History

Software systems have been around for several decades, but it’s only fairly recently that they started to be used by countless millions of people around the world. Only 20 years ago, the majority of software applications were used by trained professionals, whereas today most inhabitants of the world interact directly with one or more software applications every day. This enormously rapid growth in the number of people regularly using computers couldn’t have taken place without the great advances in usability and user interface techniques that accompanied it.

Looking back, it's hard to believe that interacting with early computers was fun for some people (although I might get into a fight with those of you still using Vim). Most people who remember "green screen" terminals will agree that working with a command prompt is generally not the most pleasant user experience. Issuing a command using the keyboard, pressing Enter, and waiting for the output to show up on the screen hardly constitutes a rich client (although for some tasks and some users, it's still appropriate and sometimes even more productive).

In order to avoid confusion, let's explain what characterizes a rich client. The "richness" of the client is determined by the interaction model that the client offers to the user. A rich user interaction model is one that can support a variety of input methods and responds intuitively and in a timely fashion. As a rule of thumb, in order to be considered rich, the user interaction should be as good as the current generation of desktop applications, such as word processors and spreadsheets. This includes features such as providing different means of interaction (for example, using keyboard and mouse to navigate, inline editing, and drag and drop) and direct visual feedback (for example, changing the cursor shape, indications using colors, and highlighting buttons and windows).

The change from those old kinds of application to modern rich web applications that this book deals with has been a long, gradual one. Figure 1-1 provides an overview of the main stages of this change, and will serve as the basis for our short history. We can roughly distinguish four stages in the evolution of software applications. The arrow depicts the path through those stages over time.

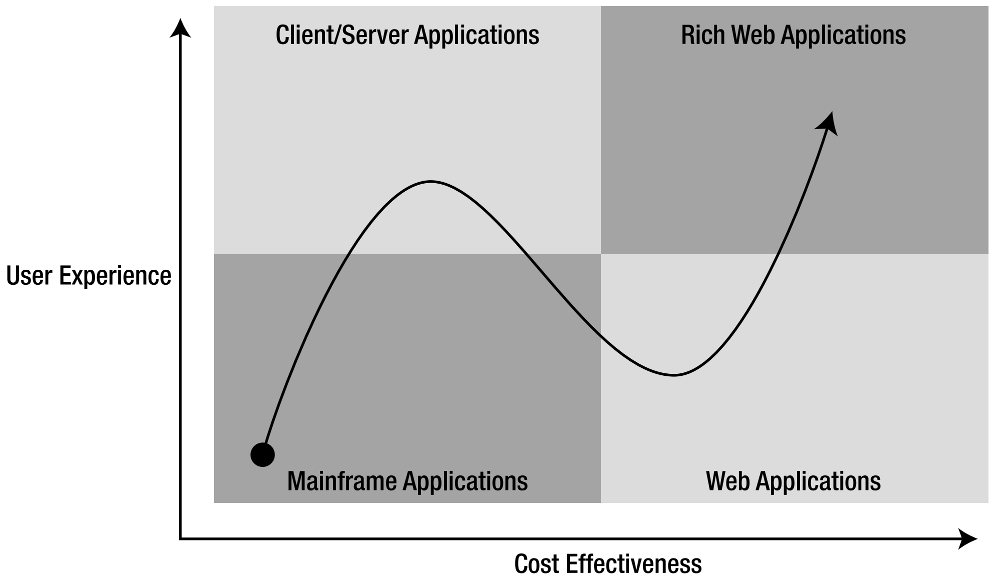

Figure 1-1. *An overview of the history of software applications*

Mainframe Applications

Starting around the 1960s, mainframe applications, whose users gained access through punch cards and lsater terminals (or terminal emulations), formed the first stage of software applications. The "green screen" terminals (the monochrome monitors of most terminals and early

PCs) provided a text-based user interface for interacting with the server-side application. Obviously a text-based user interface doesn't allow for rich interactivity such as drag and drop or instant feedback while the user types. Furthermore, terminal applications can't provide the best responsiveness, as the user input always has to be processed on the server before the user can get feedback.

From a cost effectiveness perspective, mainframe applications are often criticized for spiraling maintenance costs that are the result of a vicious circle of lack of documentation and the inability to upgrade. Applications had to be developed for the specific operating system that the application needed to be deployed on, possibly resulting in an even larger code base depending on the number of supported operating systems.

Client/Server Applications

As the personal computer became more popular, people began to have computing power on or under their desks. With this revolution came the change from a command-line UI to a more desktop-like UI, resulting from the WIMP (windows, icons, menus, and pointer device) model invented at Xerox PARC in the 1970s. Although the early adoptions by Apple and later Microsoft were poor, they allowed developers to create applications featuring richer interaction with the user. The graphical power of desktop machines helped applications become more user friendly by providing better visual feedback. However, these applications still required centralized data storage and processing, therefore needing to interact with a central server, hence the term *client/server*.

In terms of cost effectiveness, things didn't improve much, for in addition to the server side of the software, a client side also had to be developed. And as the requirements for the client side usually dictated support for more than just one OS, development and maintenance costs grew even worse. Another cost-increasing factor was that because the software didn't run in one central place but on numerous individual machines, a new version of the software required an update to all the machines that it was installed on.

Web Applications

At the same time that software applications were developed on or around a central server, the Internet and the Web were becoming more popular and more widespread. The Web was originally meant as a platform to allow people to share information by publishing documents and cross-referencing them using hyperlinks. As usage of the Web became more widespread, more and more people started having software on their computers that could interact with this document structure. The web browser provided common functionality, such as going back and forward through the navigation history and bookmarking certain pages for later retrieval. But the main advantage was that instead of software vendors needing to create specific versions of their applications for different operating systems, they had this huge installed base at their disposal. If only they could tap into that!

In order to understand the next steps in the evolution of the software application, we have to look in little more detail at the structure and inner workings of the Web. Web browsers render documents formatted using the Hypertext Markup Language (HTML) (`http://www.w3.org/HTML`). They find the document that the user wants using a uniform resource locator (URL) (`http://www.w3.org/Adressing`) and use a Hypertext Transfer Protocol (HTTP) (`http://www.w3.org/Protocols`) request to retrieve the document from a remote web server.

This is an important concept of the Web: all documents actually reside on one or more web servers, and the web browser (the client) retrieves the document from the server. It then reads the HTML document, applies the formatting rules defined in the file (as discussed later), and renders it on screen for the user to read.

The enormous popularity of the Web is largely due to the fact that it's unique; there's only one Web, wherever you go and whatever platform you use. Moreover, the Web is defined by a handful of vendor-neutral industry standards that are laid down and managed by bodies such as the World Wide Web Consortium (W3C) (`http://www.w3.org`) and the Internet Engineering Task Force (IETF) (`http://www.ietf.org`). That has prevented individual vendors, such as Microsoft, from taking over the Web by adding proprietary extensions. The Web only works because people agree on the standards that browsers can implement and that content providers and application developers can adhere to. Not only HTML, but other standards such as Cascading Style Sheets (CSS) (`http://www.w3.org/Style/CSS`), Document Object Model (DOM) (`http://www.w3.org/DOM`), Scalable Vector Graphics (SVG) (`http://www.w3.org/Graphics/SVG`), and Portable Network Graphics (PNG) (`http://www.w3.org/Graphics/PNG`) have contributed to the Web's success. Most of these standards will be discussed in the course of this book.

Instead of just using the Web to serve static HTML documents to users, someone came up with the idea of letting users request a dynamically generated document. That way, for instance, the user could take advantage of the (already installed) web browser to review real-time statistics or personalized content. This is where the web application was born. A *web application* is an application that resides on a central server and can be accessed by users through web browsers that they already have.

From a cost effectiveness perspective, this is a great way to develop software applications. You develop and deploy the applications once, and instead of having to install client applications on every user's machine, the clients already have all the necessary software preinstalled. Also, when you develop a new version of your application, you just replace the central version, and because clients interact with that central version, they will automatically receive the new version.

However, from a richness point of view, things went back to the mainframe days. Instead of having the rich user experience of the desktop application, allowing multiple means of interaction, responsiveness, and visual feedback, things went back to the issue-a-command-and-wait interaction model. Every action in a web application results in a call to the server to generate the next page or document. Therefore, as a client, you issue a command by clicking on a link or submitting a form, and then you might have to wait hundreds of milliseconds or even seconds for the entire page/document to return from the server. In the meantime, you have no way to interact with the application.

Even though there's an obvious setback in terms of usability, the greatly improved cost effectiveness has made web applications today's most popular type of software application.

Rich Web Applications

Although web applications have become the de facto standard for developing software applications, they're mostly used to develop general-purpose, publicly available applications. Still, numerous applications that depend heavily on rich user interaction to accomplish certain day-to-day tasks are developed as client/server applications. The reason for this is obvious, as the Web and its document-centric approach don't allow for rich user interaction. Imagine a spreadsheet-like web application, where every time you enter or modify data in a cell, you have to wait for the entire page to reload with recalculated values. This is even worse when the communication between client and server goes over a public network, typically leading to higher latency between requests and responses. Having too much latency for direct feedback might lead to an unresponsive application, and is therefore not suited to performing a person's everyday job. Therefore these kinds of applications, up until recently, remained in the client/server domain, leveraging the capacities of a truly rich application, but still carrying the burden of being targeted at different environments.

This is where Ajax comes in to solve this issue by allowing developers to create user-friendly applications and still reap the benefits of being able to deploy applications on the Web.

Introducing Ajax

As sketched in the previous sections, developers needed a way to develop interactive applications while still being able to deploy those applications on the web. Ajax meets exactly that need. For instance, developers can use Ajax to provide autocomplete functionality that retrieves and displays appropriate suggestions as users type into an input field. They can also write powerful, Web-based chat applications that don't need to refresh the entire page while the user is typing a new message.

Ajax does all this by using JavaScript in the browser to modify the UI directly, using the `XMLHttpRequest` object, which is discussed later, to communicate with the server without having to refresh the entire page. Then it uses the information returned from the server, usually in XML or another text format, to update the UI.

Even before the term *Ajax* was coined, developers were already developing applications like those described, using browser-specific features (such as Netscape's LiveConnect: `http://developer.mozilla.org/en/docs/LiveConnect`). But it wasn't until Jesse James Garrett coined the term in his article "Ajax: A New Approach to Web Applications" (`http://www.adaptivepath.com/publications/essays/archives/000385.php`) that this way of developing applications really took off. Although Ajax is now just a word that stands for nothing, Garrett suggested it as an acronym for *Asynchronous JavaScript and XML*. Let's first look at each of the components in this acronym in order to understand what Ajax is all about.

Asynchronous

In order to understand the main difference between Ajax's and the typical applications that were developed before Ajax, we have to look closer at how users interact with web applications. Figure 1-2 illustrates this for a typical web application. The user starts off by requesting a web page. The server processes the request and sends the result back to the browser, where it's subsequently rendered for the user. A typical web application then allows the user to do only a limited number of things. The user can provide input data by using *form widgets* (clicking on a link or button to submit the information), or by requesting a new page. The result of either action is that the user has to wait for the server to return a response. Meanwhile, the user can no longer use the application. This is called a *synchronous* interaction model. All user interaction halts until the server returns with a response, and only then can the user continue to use the application.

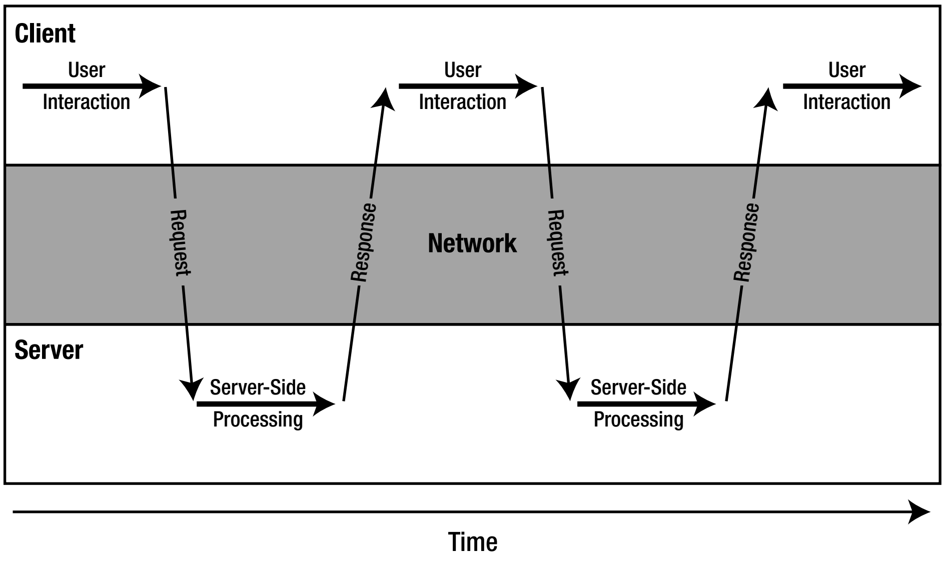

Figure 1-2. *The interaction model of a typical web application*

Although this model is acceptable for browsing through web pages (say, at a news site), it's unacceptable for more day-to-day applications, like our earlier spreadsheet example. In the case of a spreadsheet application, modifying a value and then having to wait for the server to return the recalculated outcome of a formula is unacceptable. First of all, you want to continue interacting with the spreadsheet while the result of the action is recalculated. But even more importantly, you also want to avoid having to receive and (re)render the entire page. This provides extra overhead because the server has to regenerate the entire page, the entire page is sent over the network, and the browser has to render the entire page again.

It would be so much better if we could update only the relevant cells in the spreadsheet. This is where the *asynchronous* model comes into play, taking away the gaps in the interaction and allowing the user to continue interacting with the application while previous actions are handled by the server. This model is shown in Figure 1-3.

The problem with the interaction shown in Figure 1-3 is that it breaks the classic Web model of HTTP requests for HTML pages, whose simplicity is one of its greatest strengths. The way to do this without losing more than we gain is by introducing an *Ajax engine*, a layer between the user interaction and the server communication. This engine runs inside the browser and delegates actions to the server while also handling the results. Because the engine dispatches calls to the server and pushes results to the user, the user can continue to interact with the application in the meantime. Because the engine adheres to the same standards, the Web model remains intact.

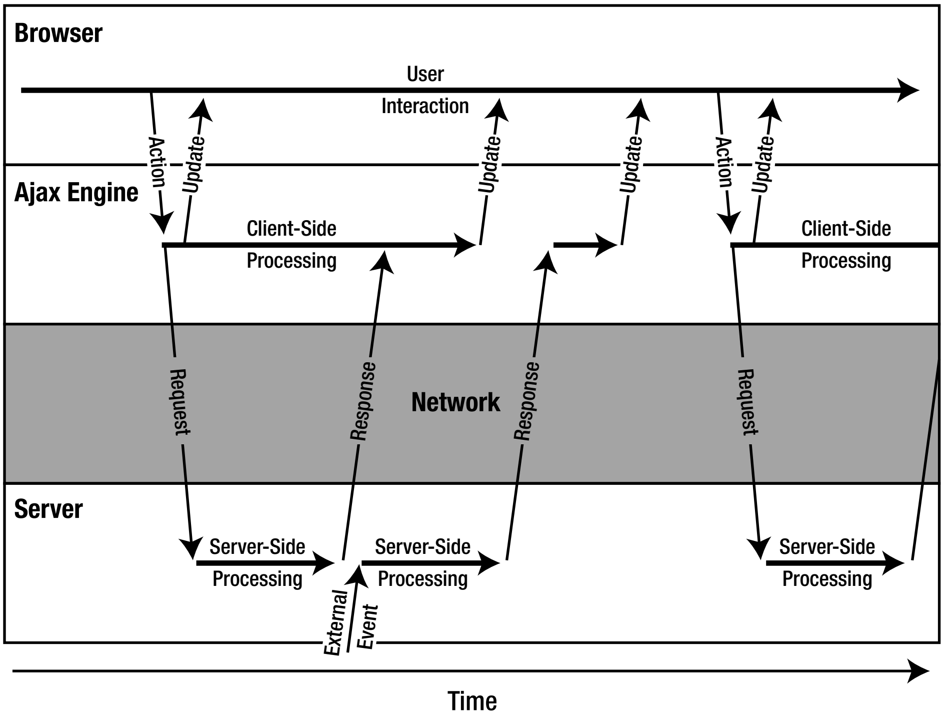

Figure 1-3. *The interaction model of an Ajax application*

To implement this asynchronous model, we need a way to send requests to the server asynchronously, without having to do a page refresh. As already mentioned, the Web was originally meant for linking documents together and navigating between them. Therefore, the Web and its standards don't directly support Ajax-style operations. However, developers are creative people, and if there's a way, it will be used to get the job done. It turned out that there was a way to do these asynchronous calls from the browser, only not in a browser-independent manner. Microsoft Internet Explorer ships with a built-in ActiveX control (`XmlHttpRequest`) that could be used to make an asynchronous call. Mozilla/Firefox contains its own similar mechanism, conveniently also called `XmlHttpRequest`, only implemented as a native object. Both mechanisms are similar in the way you use them. This allowed developers to apply this asynchronous model in applications that work in most browsers. This quickly made Ajax quite popular.

Note Although the use of `XmlHttpRequest`, either as an ActiveX component or as a native object, is by far the most popular way to dispatch asynchronous calls to the server, there are other ways. Two examples of other remoting mechanisms are using a hidden `iframe` and dynamically adding a `script` tag in the head of the document.

JavaScript

JavaScript is a scripting language created by Brendan Eich when he was working at Netscape. Originally it was named *Mocha* and later *LiveScript*, but renamed JavaScript around 1995. Although the name suggests otherwise, the language is quite different from the Java programming language, although they share a common ancestor in the C syntax. JavaScript was probably named after the Java language as a marketing move that was part of a strategic alliance between Sun Microsystems and Netscape. Some say they used Java in the name to give JavaScript the allure of being the new "hot" web programming language. It's mostly because of the support in Netscape that JavaScript became the most widely supported scripting language in browsers. Microsoft started off with its own dialect called *JScript*, but later switched to supporting JavaScript. In 1996, JavaScript was submitted for standardization, resulting in the ECMAScript specification with JavaScript as one of its implementations, others being *ActionScript* (heavily used by Adobe) and *JScript*, which is still used for Microsoft's Active Scripting.

JavaScript is a dynamic, weakly typed, prototype-based language with first-class functions. It supports the structured programming syntax of C, including loops, switch statements, and so on. One exception is that it doesn't support block-level scoping.

In order to get a better idea of JavaScript, let's look at Listing 1-1, which shows a simple HTML page that uses JavaScript to display an alert to the user with a value that the user has supplied in an input field.

Listing 1-1. *Sample JavaScript Embedded in an HTML Page*

```
<html>
    <head>
        <title>JavaScript sample</title>

        <script type="text/javascript">
            function handleButtonClick() {
                var input = document.getElementById('query');
                alert('Query: ' + input.value);
            }
        </script>
    </head>
    <body>
        Query:
        <input id="query" type="text" value=""/>
        <button onclick="handleButtonClick()">Show</button>
    </body>
</html>
```

The JavaScript fragment in Listing 1-1 looks much like Java, and it's easy to understand what the code is trying to achieve. However, you should also note the subtle differences, such as the variable and function declarations. Luckily, as you'll see in the remainder of this chapter, you don't need to know much more about JavaScript if you want to develop RIAs using GWT.

However, if you want more information about JavaScript, try the *JavaScript 1.5 User's Guide* (`http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide`), *Beginning JavaScript with DOM Scripting and Ajax: From Novice to Professional* by Christian Heilmann (Apress, 2006), and *Pro JavaScript Techniques* by John Resig (Apress, 2006).

XML

Extensible Markup Language (XML) has become popular in the developer world as a means of transferring data in a language-neutral manner. Basically, a markup language is a language that employs tags to describe how its content is to be structured, laid out, or formatted. For instance, the sample XML fragment in Listing 1-2 describes the structure of a menu.

Listing 1-2. *Sample XML Describing a Menu Structure*

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
    <item id="1">
        <description>
           A menu item
        </description>
    </item>
    <item id="2">
        <description>
            Another menu item
        </description>
    </item>
</menu>
```

As you can see, the information in Listing 1-2 is structured in a hierarchical way. XML is a simplified form of the Standard Generalized Markup Language (SGML). Both SGML and XML are meta-markup languages, allowing you to define your own markup languages. One application of SGML is HTML, as depicted in Figure 1-4. Listing 1-3 shows some textual content marked up using HTML.

Listing 1-3. *Sample Content Containing HTML Markup*

```
<div>
    <h1>New version of GWT released</h1>
    <p>
        Last night, word reached us that a new version of the hugely popular
        application development framework by Google<sup>TM</sup> is released.
        <br>
        More information:
        <a href="http://code.google.com/webtoolkit">
            http://code.google.com/webtoolkit
        </a>
    </p>
    <p>Sept. 21, 2008</p>
</div>
```

Basically, HTML provides a set of predefined tags that developers can use to add markup information to content. Note that in HTML, you can't use your own tags. So for instance, changing the `h1` tag to `name` would result in the interpreter either failing or just ignoring it.

This is where XML comes in: it allows you to define your own tags. XML allows different ways to describe your document structure by means of schemas. Different schema languages exist for XML, including the Document Type Definition (DTD), which provides a limited set of functionality, and the more powerful XML Schema. But as these document descriptions are optional, you can easily do without them. So the content from Listing 1-3 could be expressed using XML as shown in Listing 1-4.

Listing 1-4. *Content Containing XML Markup*

```
<newsitem>
    <name>New version of GWT released</name>
    <description>
        Last night, word reached us that a new version of the
        hugely popular application development framework by
        Google<sup>TM</sup> is released.
    </description>
    <link>http://code.google.com/webtoolkit</link>
    <date>Sept. 21, 2008</date>
</newsitem>
```

If you compare Listings 1-4 and 1-3, you'll note that XML adds much more semantic value to the text. As its name spells out, XML is a markup language. But unlike HTML, which uses markup for a single well-defined purpose, XML is also a language for defining markup languages. It adds semantic value by describing the content within the tags. Still, one could write an interpreter that uses the semantic information described by the tags and use that to format the information to the user.

Although it may seem so, HTML is not an extension of XML. This is because in contrast to XML, HTML allows for single tags: opening tags with no closing tags, such as the `<br>` in Listing 1-3. This is allowed in SGML but not in XML; in XML, each opening tag should have a corresponding closing tag or it should be an empty tag: it should be either `<br></br>` or `<br/>`. In order for HTML to be more easily validated and accessible for more devices and platforms, Extensible Hypertext Markup Language (XHTML) recently has become popular. This combines the set of tags defined in HTML with the stricter syntax of XML, allowing for easier validation and interpretation. Figure 1-4 gives an overview of all the markup languages we've discussed so far.

Before we look at XML and more specifically how it's used in Ajax, let's digress for a moment. As mentioned earlier, the Web works because of standards. One of these standards is the Document Object Model (DOM): see `http://www.w3.org/DOM/`. This platform- and language-neutral interface allows programs and scripts to dynamically access and update the content, structure, and style of documents (for example, XML and HTML documents). The DOM represents a document as a hierarchical tree of nodes that can be accessed and manipulated. All browsers that support JavaScript use the DOM to represent and render HTML documents. This allows developers to use JavaScript to inspect and modify the document and therefore the UI. If you look back at Listing 1-1, you can see how we use the DOM to access the input field and retrieve its value. We used the JavaScript method `getElementById(String)` on the DOM representation of the HTML page.

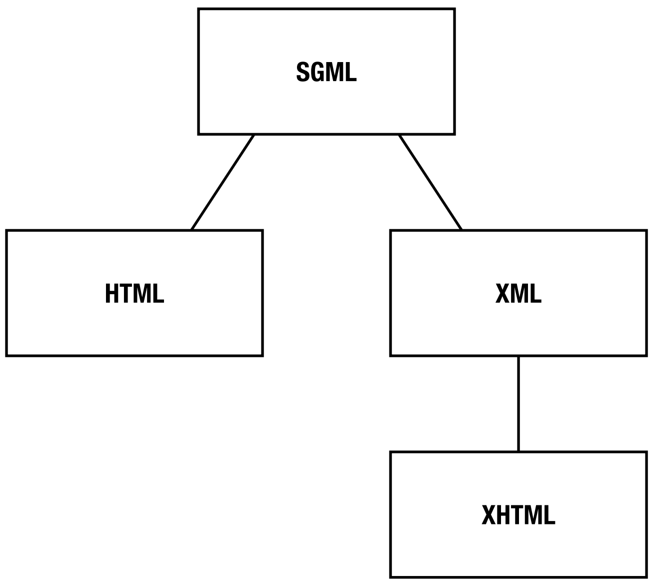

Figure 1-4. *SGML and a selection of its child languages*

As mentioned earlier, XML allows data to be transferred in an interchangeable way. This is exactly what it's used for in Ajax applications. It allows you to transfer data from the client to the server and vice versa. This makes it easy for the client-side platform to be quite different from the server-side platform. So for instance, while the client side is written in JavaScript, the server side can be written in any possible programming language. Therefore, XML seemed like an ideal data transfer format when Ajax was created. However, as we'll see in the next section, that turned out to be incorrect.

From AJAX to Ajax

So far we've discussed the major components that, in the past, made up the acronym AJAX. However, in 2006 Garrett redefined the AJAX as *Ajax* (a simple name, standing for nothing in particular).

Why did he make this change, and why is it important to discuss here? Let's first look at why he made the change. After several years of developing applications based on this new paradigm, two of the three components expanded beyond their original context. The first, *asynchronous*, remained in place, although it was more and more implemented without using the `XmlHttpRequest` object that had originally been one of the biggest catalysts behind the whole movement. But the other two, JavaScript and XML, were slowly being expanded. As we'll see in the next section, other approaches such as Flex use a language other than JavaScript to do their scripting in the browser environment. Flex for instance uses ActionScript for scripting (`http://www.adobe.com/devnet/actionscript`), but also has support for other scripting languages.

Therefore, the *JavaScript* part of the AJAX acronym was no longer valid. But even more obvious was the need for a change to the *XML* part of the acronym. XML turned out not to be the best choice for exchanging data between the client and the server. Obviously in some cases you could use XHTML to communicate the content and place it directly inside a page, but this is inconvenient when sending plain data. This is mostly because browsers tend to work differently with XML, both because APIs tend to differ, but even more importantly because the performance can be very different.

Therefore, after initially using XML, developers started to look for alternatives. Basically, they found three alternatives to XML:

- **A proprietary protocol**—either a textual or binary protocol that allows for data communication between the client and server. The advantage of a proprietary protocol is that it can easily be optimized for both size and speed.
- **JavaScript Object Notation (JSON)**—JSON is a lightweight data interchange format based on name/value pairs. Originally based on JavaScript, and implemented as a subset of it, JSON provides a good bridge between a JavaScript client and a server implementation using Java. For more information, see `http://www.json.org`.
- **Plain JavaScript**—instead of using a specific interchange protocol, you could send plain JavaScript over the wire. Although this is the most powerful and flexible option (because you can use all features and syntax provided by the language), it requires the server to generate JavaScript. As the server-side code is generally written in a language other than JavaScript, this isn't recommended. It's usually much better to use some interchange format such as JSON.

Each solution has its advantages and disadvantages, but if we look at the majority of Ajax applications today, we must conclude that XML is no longer the most widely used data transport format.

So Garrett redefined Ajax as a name rather than an acronym. He defined Ajax as an application developed for the Web, using DHTML (see sidebar) and doing some kind of remote communication with the server in an asynchronous manner. That's the kind of application that you'll learn to develop in this book.

DYNAMIC HTML (DHTML)

The term *DHTML* describes the usage of several of the previously described technologies to create interactive and animated web sites. DHTML combines a static markup language (such as HTML), a client-side scripting language (such as JavaScript), a presentation definition language (such as CSS), and the DOM. Combining these technologies allows you to dynamically change the appearance and behavior of a web page. More recently, DHTML is also referred to as *DOM Scripting*, although this somewhat narrows the meaning of the term.

Advantages and Disadvantages of RIAs

Before exploring different approaches to building rich Internet applications, let's first look at their advantages and disadvantages compared to classical and web applications. In the previous sections, we have already touched on some of these.

RIA Benefits

Next to the obvious, intrinsic benefits of building a rich Internet application, it also comes with the following additional benefits:

- **No installation required**—the application is downloaded automatically and runs inside the browser. The software that actually runs the application is already installed on the client machine.
- **Updates are automatic**—new versions of the application are also downloaded automatically by simply revisiting the application's web page.
- **Platform independent**—a rich Internet application can potentially run on every platform and operating system, as long as it has an Internet browser and connection to the Internet.
- **More secure**—applications run in the restricted environment of a web browser and are therefore much less likely to be harmful than applications that need to be installed.
- **More responsive**—because not all user actions require server communication, rich Internet applications tend to be more responsive than classic web applications.
- **More scalable**—a great part of the computational work as well as state keeping can be offloaded to the client, so the server can handle many more users. It no longer needs to maintain state, or at least not as much.
- **More network efficient**—in classical web applications, every user action requires the server to regenerate the entire page and send that over the network. In the case of a rich Internet application, the entire application UI only needs to be communicated once. All other requests to the server require only the actual data to be sent to the client.

RIA Shortcomings

With every good thing, there are some drawbacks. The same goes for Ajax and building rich Internet applications. The following are some of the limitations:

- **Requires JavaScript or specific plug-in**—because the entire application runs through the JavaScript interpreter on the client, what happens when the user has turned off JavaScript completely? Usually, the application does little or nothing. Obviously it's possible to have a backup plan for those users, but then you have to maintain two separate applications, which is far from ideal.
- **No access to system resources**—as Ajax applications run inside a browser, they're limited in the resources they can access. For instance, an Ajax application can't access the client file system.
- **Hard for search engines to index fully**—because most search engines don't (yet) support applications that do partial page updates or use specific plug-ins such as Flash, most rich Internet applications are badly indexed by search engines. The larger search engines plan to improve their support for these kinds of applications as their popularity grows, but it will always be difficult to sufficiently index them.

- **Accessibility issues**—doing partial page updates using JavaScript or a specific plug-in can break accessibility. The biggest and most notorious issue is that existing screen readers don't handle this correctly. Although screen readers will try to provide better support, application developers should always keep accessibility in mind when developing software applications, and even more when developing rich Internet applications, because accessibility is easier to break.
- **Depends on an Internet connection**—because these applications are served from the Web and run in the web browser, they require at least an initial Internet connection. But even during usage, an Internet connection is needed to communicate with the server. When the connection is (temporarily) not available, most rich Internet applications fail to function. However, there are some attempts to use local services provided by the browser for temporary storage while a connection is unavailable.

When Should You Use Ajax?

Based on the benefits and shortcomings outlined in the previous section, we strongly believe that rich Internet applications aren't suitable for every application. But there are definite guidelines as to when to build a rich Internet application and when to stick to developing classical web applications as you've probably been doing for some time. Consider the following list:

- You want to develop an application that users need on a daily basis, and they spend a lot of time using the application.
- Business tasks rely on direct feedback from the application to enhance productivity.
- Most, if not all, of the application requires user authentication and authorization, so indexability by search engines is not the top priority.
- It's safe to assume that JavaScript is turned on, or it's acceptable to require users to turn it on before using the application.

We feel that if your application meets these prerequisites, a rich Internet application should be preferred. Assuming you have such an application (at least in mind), let's look at the different approaches that are available for developing these applications.

Different Approaches to Building RIAs

To build these rich Internet applications, we need an easy way to develop the code that runs in the browser and a way to call the server remotely. Let's briefly look at the different approaches that are available. Note that this isn't meant as an exhaustive listing of all frameworks and solutions, but just a highlight of the different approaches.

Handwritten JavaScript

The first and probably most widely used way to develop Ajax applications is by taking a normal web application and adding Ajax capabilities to it by writing some JavaScript. You would usually use preexisting libraries that allow you to focus on the important stuff while reusing features such as remoting and convenience classes for manipulating the interface. In general, there are four main concerns with this approach:

- **JavaScript is not Java**—developing applications using JavaScript is totally different from developing applications in Java. It's not only a completely separate language with its own concepts and constructs, but it also requires a different tool set. And although it's nice for developers to learn a new language, that might not be the best choice from a business perspective. The obvious solution is to hire professional JavaScript developers; but really skillful JavaScript developers are hard to find.

- **Browser quirks**—the biggest challenge and probably the most time-consuming activity when developing Ajax applications is coding an application that handles all the differences between browsers and even between different versions of the same browser. A good JavaScript developer should know all these browser quirks by heart.

- **Too many libraries**—assuming you don't want to reinvent the wheel, you start looking for libraries that give you the basic features you need. But where do you start and which one do you choose for a particular project? At the time of this writing, AjaxPatterns.org listed 42 JavaScript Multipurpose frameworks and 51 Java Ajax frameworks, so in all, you have to choose from 93 frameworks for building your application.

- **Libraries only solve part of the puzzle**—choosing a library is generally not enough; they typically solve only part of the puzzle in that they only handle a certain aspect of building Ajax applications. One might contain basic functionality for remoting, while another provides widgets, and a third handles visual effects. So you end up with a number of different frameworks that you have to tie together into one application.

This approach is workable when adding limited Ajax functionality to a traditional web application. But taking into account these concerns, we don't suggest using this approach to develop the rich Internet applications that we discussed earlier.

Flex

Adobe Flex is a set of tools for developing rich Internet applications on the proprietary Adobe Flash platform. Flash itself offers a lot more than you could get from HTML in terms of real-time interactivity. However, developing rich applications in Flash is daunting and unintuitive for core developers. The Flash development tool is geared for designers, and developing on a timeline is a strange concept for applications that aren't driven by time but by user actions. Flex removes that barrier to entry by providing a programmatic way for developing RIAs. Although the Flex platform was originally developed by Macromedia in 2004 as a commercial product, Adobe made parts of it open source in 2007 after its acquisition of Macromedia.

When developing rich Internet applications using Flex, you use the tools provided by Adobe. The good thing about this is that Flex comes with a lot of built-in functionality, such as widgets, drag and drop, vector graphics, and animations. This will greatly speed up your development, especially if you're already familiar with Flash and ActionScript.

The main advantage of using Flex is that you develop your application to run inside the controlled environment of the Flash plug-in. So you don't have to take browser quirks into account. On the other hand, the client must have the plug-in installed on the client computer, which might be not the case in some locked-down environments such as schools and government agencies. Some purists also feel that Flex isn't a real Ajax application framework, as it doesn't adhere to the standards normally used for building Ajax applications, such as XHTML and JavaScript.

In February 2008, Adobe released its Adobe AIR runtime, which allows applications developed in Flex to be deployed as local desktop applications while still enjoying the benefits of Ajax, such as seamless installation and updating. The main benefit of AIR is that it gives application developers the best of both worlds: you can use the deployment model of a web application while still getting desktop integration if you need it. More in-depth coverage of Flex, and a guide on how to get up to speed, can be found in *The Essential Guide to Flex 3* by Charles E. Brown (Friends of ED, 2008).

Adobe Flex, especially since the release of Adobe AIR, is a good choice for developing rich Internet applications. Especially when you can't do without desktop integration features such as user notification, startup integration, background processes, and (most important) access to the local file system, Flex is a great option. However, remember that even though it's mostly open source, it still requires users to install a proprietary plug-in before using your applications developed in Flex.

More information about Flex can be found at `http://www.adobe.com/products/flex` and more about Adobe AIR can be found at `http://www.adobe.com/products/air`.

Java Applets and JavaFX

Another approach to building rich Internet application is using Java applets, or more recently, JavaFX. The most interesting part of JavaFX for our purposes is JavaFX Script, a scripting language for easily creating rich media and interactive content. However, it's built on the Java development platform, so if you want to deploy JavaFX applications on the Web, you're in essence deploying a Java applet, only one created with JavaFX. Therefore, we'll discuss Java applets in this section and just note that it also covers applications written in JavaFX.

The main benefit of a Java applet is that it's a Java program that runs on the client side. Although there are some obvious security restrictions when running from the browser, you can do anything you want with a Java applet, for example make connections to the originating server and spawn threads. You can even drop all security restrictions by digitally signing the application and asking the user for additional permissions, such as accessing the local file system. So if you want to build a rich Internet application, this appears to be the ideal solution. More in-depth information can be found in *JavaFX Script: Dynamic Java Scripting for Rich Internet/Client-side Applications* by James L. Weaver (Apress, 2007).

The main problem with Java applets and JavaFX is the need for a Java Runtime Environment (JRE) to be installed on the client. The JRE has a lower installed base than for instance Flash Player. Another issue is that depending on the requirements and development time, you may need a different version of the JRE. Furthermore, in contrast to the Flash player, the JRE has a substantial download size of approximately 14Mb for offline installation on the Windows platform at the time of writing. Table 1-1 provides an overview of the market penetration and download size of some of the packages discussed in this section.

More information on Java applets and JavaFX can be found at `http://java.sun.com/javafx`.

Silverlight

Silverlight is Microsoft's recent attempt to provide a platform similar to Flash. It's restricted to a subset of the .NET framework and should therefore be more suitable for developing rich Internet applications. Although Silverlight is starting to get some attention, it has a long way to go before it can call itself a serious alternative to Flash. Currently the biggest hurdle for Silverlight is enlarging its installed base. At the time of this writing, Silverlight's market penetration is negligible. As it has taken Adobe many years to reach its current market share, it will probably be a long time before Silverlight can even begin to challenge it.

Another issue with developing rich Internet applications using Silverlight is that runtimes are currently available for the Windows platform only. For the foreseeable future, no Mac and Linux users will be able to use applications developed in Silverlight. Also, the download size of the Silverlight runtime is large compared to Flash.

For these reasons, we don't consider Silverlight to be a serious option for developing rich Internet applications at the time of this writing, at least not until it acquires a considerable installed base and Microsoft releases runtimes for Linux and Mac users.

More information on Silverlight can be found at `http://www.silverlight.net`.

Table 1-1. *Installation Solutions' Market Penetration and Download Size*

Product	Market Penetration (Rounded)	Download Size (in Mb)
Flex/Flash	99%	1.5
Java/JavaFX	85%	14
Silverlight	0%	10

OpenLaszlo

Another product that we want to highlight is OpenLaszlo, released by Laszlo Systems under the Common Public License. OpenLaszlo takes the approach to define your UI in Laszlo's own LZX programming language. LZX is basically XML with embedded JavaScript snippets describing application logic. OpenLaszlo then takes the UI definitions and generates a rich Internet application. Previous versions only allowed the application to be translated into a fully functional Flash application. This runs into the same problems as the Flex approach. But starting with version 4 of OpenLaszlo, it now supports translating the UI into a DHTML application. The latest version also supports cross-compilation to embedded platforms such as mobile phones and other devices.

The main problem with OpenLaszlo is that it requires all application logic to be written in JavaScript. This leads to the same problems we discussed in the section about handwritten JavaScript. However, OpenLaszlo claims to eliminate a number of cross-browser compatibility issues, because the compiler is aware of them (see Figure 1-5).

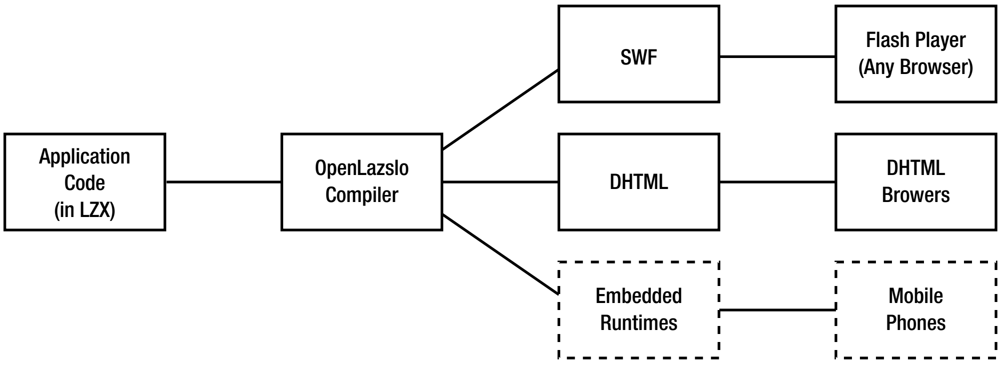

Figure 1-5. *OpenLaszlo application code compiled to different output formats*

More information about OpenLaszlo can be found at `http://www.openlaszlo.org`.

Echo2

The Echo Framework is yet another approach to developing rich Internet applications. The stable version at the time of writing is Echo2. Echo2 takes a nice approach to building these applications by allowing you to write them in Java. In Echo2, you write a servlet that handles all user requests and creates an application for each unique user. An application is created by extending the `ApplicationInstance` class. This application instance determines the UI of the application and handles user events. Because a new instance of the application is created for each user, it also holds all state. Internally, the application instance is stored in the user session.

The Echo framework takes a revolutionary approach to developing rich Internet applications. But in our opinion, there are two main problems with Echo2:

- **(Almost) every user action results in a call to the server**—because of the way Echo2 works internally, keeping state on the server in the user session, every action that the user performs needs to be communicated to the server. In our opinion, a rich Internet application should communicate with the server only when needed. In other words, the application should run entirely in the browser and only communicate to the server when it needs to retrieve data or perform a server-side action, or both.
- **Keeps all state on the server**—by nature, a rich Internet application maintains most, if not all, state on the client. This makes it possible to offload the server and scale better. One thing we've learned over the years in developing Java applications that handle large number of users is that you should keep as little state on the server as possible. Building rich client-side applications makes that possible to a great extent; but we feel that Echo2 hinders that by keeping most state on the server.

More information about Echo2 can be found at `http://echo.nextapp.com/site/echo2`.

GWT

As GWT is the subject of this book and is introduced in detail in the next chapter, we won't go into much detail here. But looking at GWT in contrast to the previously discussed solutions, there are two main things to note. First of all, GWT looks a lot like OpenLaszlo in the sense that the application you develop in GWT will eventually be compiled into a DHTML application and run entirely in the browser using XHTML and JavaScript.

But the main advantage of GWT over, for instance, OpenLaszlo is that instead of defining your UI in XML and JavaScript, applications are developed in Java. So for Java developers, this is a natural fit. You can reuse your existing expertise and best practices, favorite IDE, and development tools. The benefit of using GWT over other tools such as Echo2 is that GWT code compiles to a fully client-side JavaScript application and not just a proxy to the server-side Java application.

The most serious competitor GWT is Flex. But in our opinion, GWT has two advantages: first it's completely open source, whereas Flex is just partly open source; and second, a compiled GWT application runs inside any modern browser (assuming JavaScript is turned on) while Flex applications run only on systems where the Flash player is installed.

Summary

In this chapter, we introduced the concept of a rich Internet application (RIA) and illustrated where it came from by giving some historic background. We went from the mainframe application with its terrible user experience and limited cost effectiveness to rich Internet applications that are superior in both respects.

Next, we introduced Ajax, discussing every letter in the original acronym and showing why it's no longer an acronym. Most importantly, we discussed the advantages and disadvantages of using Ajax for developing RIAs, and laid down some guidelines on which applications are best suited to Ajax.

Last, we gave our take on several approaches to building Ajax applications, introducing some different strategies together with their strengths and weaknesses. This revealed why we believe GWT is the best solution currently available for developing RIAs, and why the rest of this book deals with GWT and not one of the other technologies.

The next chapter will introduce GWT in more detail, showing a typical GWT application layout. It will also introduce the sample application that we're going to gradually build in the course of this book.