Ajax Patterns and Best Practices

■ ■ ■

Christian Gross

Apress®

Ajax Patterns and Best Practices

Copyright © 2006 by Christian Gross

ISBN-13 (pbk): 978-1-59059-616-6

ISBN-10 (pbk): 1-59059-616-1

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

Lead Editor: Jonathan Hassell
Technical Reviewer: Paul Tyma
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editor: Sharon Wilkey
Assistant Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Susan Glinert
Proofreader: Elizabeth Berry
Indexer: Broccoli Information Management
Artist: Kinetic Publishing Services, LLC
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 2560 Ninth Street, Suite 219, Berkeley, CA 94710. 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` in the Source Code section.

CHAPTER 3

Content Chunking Pattern

Intent

The Content Chunking pattern makes it possible to incrementally build an HTML page, thereby allowing the logic of an individual HTML page to be distributed and the user to decide the time and logic of the content that is loaded.

Motivation

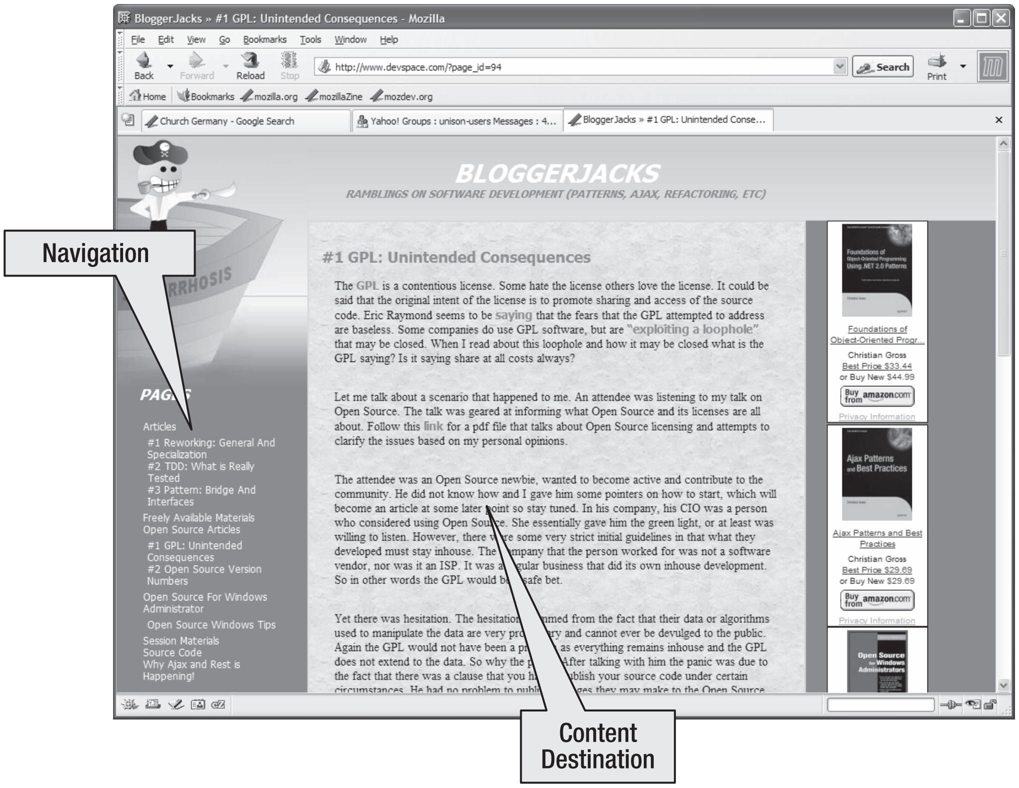
Originally, when the Web was in its infancy, HTML content designers created documents that were incomplete. The incomplete pages were made complete by using document links. The completeness of a document was the sum of the pages in the document tree.

Think of it as follows: instead of creating a book in which you follow through the content in a sequential manner, for the Web you would paste materials together like a bunch of magazine articles. But unlike a magazine that required you to go through one page after another, the Web allowed you to click a link and jump to different content. As time passed, websites moved away from this distributed web structure to a strictly hierarchical self-contained structure.

An example of a strictly hierarchical self-contained website is illustrated in Figure 3-1.

In Figure 3-1, the website is split into two areas: blue-background navigation and brown-background content. When a user clicks on a navigational link, the content is changed. But the problem is that the entire page is reloaded even though the user is only interested in the brown-background content. One way to get around this problem is to use HTML frames so that the navigational area is one frame, and the content area is another frame. When a link in the navigational area is clicked, only the frame containing the content is altered. However, as time has shown, although frames solve the problem of loading content individually, they are problematic from a navigational and user interface perspective. Thus websites have used fewer and fewer frames.

Ideally, what a website developer wants is the ability to alter the content that needs to be altered and to leave the rest of the content as is. After all, untouched content is content that stays the same and works.


BloggerJacks » #1 GPL: Unintended Consequences - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://www.devspace.com/?page_id=94
Search
Print
Home Bookmarks mozilla.org mozillaZine mozdev.org
Church Germany - Google Search
Yahoo! Groups : unison-users Messages : 4...
BloggerJacks » #1 GPL: Unintended Conse...
BLOGGERJACKS
RAMBLINGS ON SOFTWARE DEVELOPMENT (PATTERNS, AJAX, REFACTORING, ETC)
Navigation
PAGES
Articles
#1 Reworking: General And Specialization
#2 TDD: What is Really Tested
#3 Pattern: Bridge And Interfaces
Freely Available Materials
Open Source Articles
#1 GPL: Unintended Consequences
#2 Open Source Version Numbers
Open Source For Windows Administrator
Open Source Windows Tips
Session Materials
Source Code
Why Ajax and Rest is Happening!
Foundations of Object-Oriented Progr...
Christian Gross
Best Price $33.44
or Buy New $44.99
Buy from amazon.com
Privacy Information
Ajax Patterns and Best Practices
Christian Gross
Best Price $29.69
or Buy New $29.69
Buy from amazon.com
Privacy Information
Open Source for Windows Administrators
Content Destination

#1 GPL: Unintended Consequences

The GPL is a contentious license. Some hate the license others love the license. It could be said that the original intent of the license is to promote sharing and access of the source code. Eric Raymond seems to be saying that the fears that the GPL attempted to address are baseless. Some companies do use GPL software, but are "exploiting a loophole" that may be closed. When I read about this loophole and how it may be closed what is the GPL saying? Is it saying share at all costs always?

Let me talk about a scenario that happened to me. An attendee was listening to my talk on Open Source. The talk was geared at informing what Open Source and its licenses are all about. Follow this link for a pdf file that talks about Open Source licensing and attempts to clarify the issues based on my personal opinions.

The attendee was an Open Source newbie, wanted to become active and contribute to the community. He did not know how and I gave him some pointers on how to start, which will become an article at some later point so stay tuned. In his company, his CIO was a person who considered using Open Sou ce. She essentially gave him the green light, or at least was willing to listen. However, there w re some very strict initial guidelines in that what they developed must stay inhouse. The mpany that the person worked for was not a software vendor, nor was it an ISP. It was a gular business that did its own inhouse development. So in other words the GPL would b safe bet.

Yet there was hesitation. The hesitation mmed from the fact that their data or algorithms used to manipulate the data are very pro ry and cannot ever be devulged to the public. Again the GPL would not have been a pr as everything remains inhouse and the GPL does not extend to the data. So why the p After talking with him the panic was due to the fact that there was a clause that you h blish your source code under certain circumstances. He had no problem to publi ges they may make to the Open Source

Figure 3-1. *Strict hierarchical structure of a website*

Applicability

Use the Content Chunking pattern in the following contexts:

- When it is not known what the HTML page should look like because of the nature of the website. In Figure 3-1, there is a blue-background navigational area and a brown-background content area. The content of each area is unknown, but what is known is the area where the content is destined.
- When the content to be downloaded is too large and would cause an excessive wait for the user. For example, doing a search and waiting for all found elements to be collected as a result set is not an option because the user would have to wait too long. A better approach would be to keep a search executing while displaying whatever is found.

- When the displayed content is not related. Yahoo!, MSN, and Excite are portal applications displaying content side-by-side with other content that has no relation to it. If the content is generated from a single HTML page, the server-side logic would have to contain a huge decision block to know which content is loaded and not loaded. A better approach would be to consider each block of content as a separate piece that is then loaded separately.

Associated Patterns

The Content Chunking pattern is a core pattern to any Ajax application. You could even make the assertion that the Content Chunking pattern is implicit to Ajax. Be that as it may, it is still necessary to identify and define the context of the Content Chunking pattern. What makes the Content Chunking pattern unique is that it always follows the same steps: generated event, request, response, and chunk injection. The other patterns covered in this book are similar, but do take deviations such as sending a request and not getting an immediate response (for example, the Persistent Communications pattern).

Architecture

The architecture of the Content Chunking pattern is relatively simple. A URL is called by the client. The server responds with some content that is received and processed by the client. An implementation of the Content Chunking pattern always follows these steps:

1. An event is generated that could be the result of a button being clicked or of an HTML page being loaded.
2. The event calls a function that is responsible for creating a URL used to send a request to the server.
3. The server receives the request and associates the request with some content. The content is sent to the client as a response.
4. The client receives the response and injects the response in an area of the HTML page.

Implementing Order in a Web Application

Looking back at Figure 3-1, the strict hierarchical nature of the website is not a bad thing. With respect to HTML, the result of the strictness is to generate the content in one step, and this all-in-one generation causes problems. Traditional applications do not function in such a manner, as illustrated in Figure 3-2.

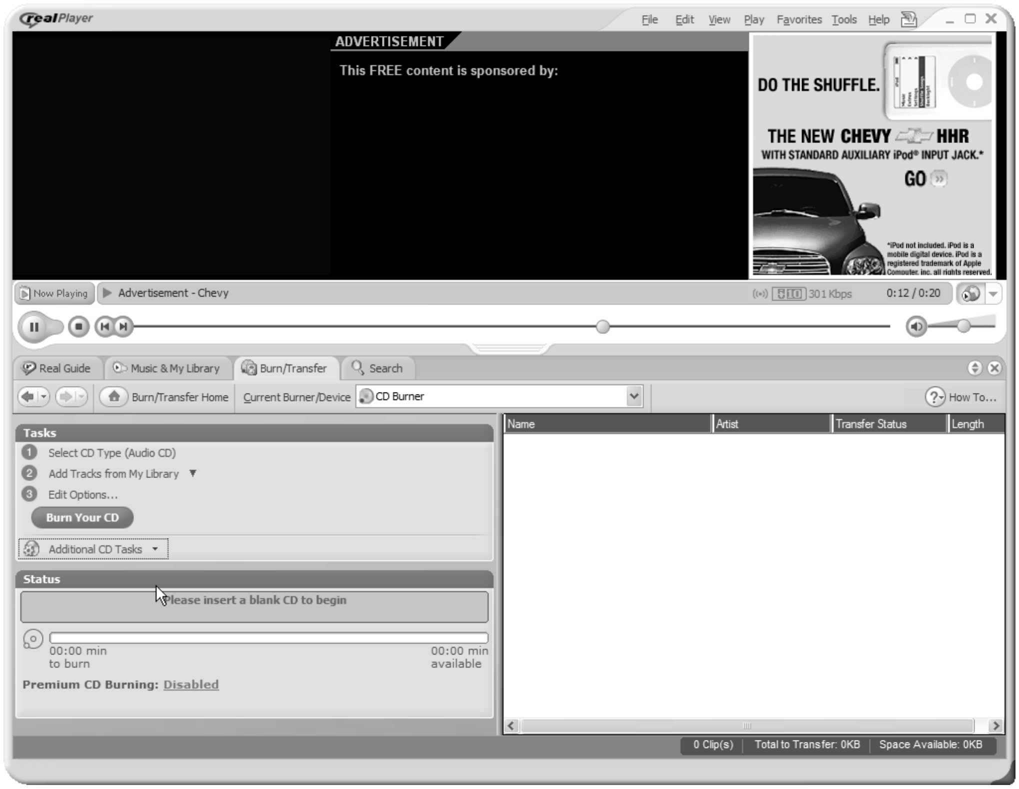


Figure 3-2. *Traditional client application*

In Figure 3-2, the RealPlayer is an example of a traditional client application that mixes newer HTML-type technologies with traditional user interface elements. Clicking the Burn Your CD button causes RealPlayer to burn your CD but does not affect the advertisement that is running at the top half of the application. The logic associated with the advertisement and the logic associated with burning the CD are two separate, distinct pieces of logic that happen to be sharing the same window area.

Figure 3-3 dissects the web application of Figure 3-1 into distinct pieces of logic.

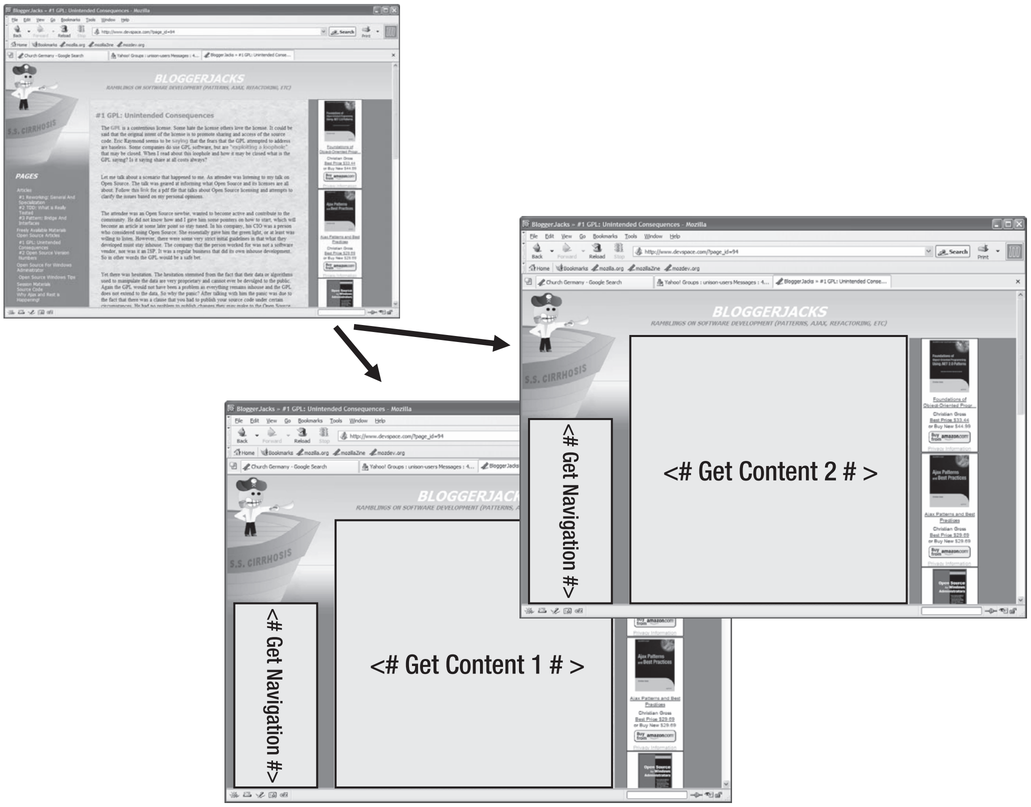


Figure 3-3. *Website architecture*

In Figure 3-3, the original HTML page has links to two other pages that represent an example blog and article content. The example content has two execution blocks: `Get Navigation` and `Get Content (1,2)`. The logic used to generate `Get Content 1` is distinct from the logic used to generate `Get Content 2`. In the context of generating an HTML page, when either `Get Content 1` or `Get Content 2` is executed, the logic `Get Navigation` is executed. This means the logic `Get Navigation` is executed multiple times, generating the same data each time. Some readers might argue that different data is generated by `Get Navigation` (e.g., different folders are opened), but in fact it is the same data formatted a different way. In a nutshell, there is an inherent data-generation redundancy that should be avoided.

The solution is to distribute the logic so that an HTML page is generated, by using an architecture similar to Figure 3-4.

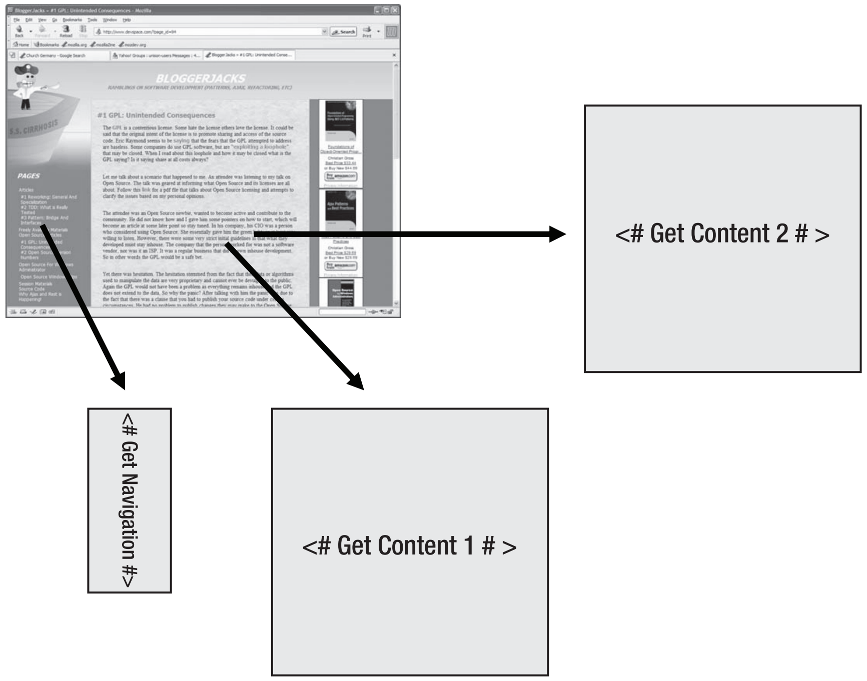


Figure 3-4. *Improved website architecture*

In Figure 3-4, the HTML page is the result of multiple pieces of server-side logic. When the main outline of the HTML page has been loaded, the `XMLHttpRequest` object retrieves the content blocks `Get Navigation`, `Get Content 1`, and `Get Content 2`. When and how the individual content blocks are retrieved depends on the events and links created by the content blocks. Each content block is a separate request that needs to be called by the `XMLHttpRequest` type.

The proposed architecture has the following advantages:

- The client downloads only what is necessary, when it is necessary. There is no need to re-retrieve a content block unless necessary.
- The architecture is separated into different code blocks that can be assembled dynamically in different contexts.
- The architecture resembles that of a traditional client in that only those elements that pertain to the event are manipulated.
- The overall look and feel is not affected because the generated code blocks delegate the look and feel to the parent HTML page retrieving the content blocks.

Figure 3-4 shows how the Content Chunking pattern got its name: a single HTML page is the sum of its chunks of content, which are referenced and loaded separately.

Defining the Content Within a Content Chunk

The content chunks referenced by the `XMLHttpRequest` object can be in any form that both the client and server can understand. Whatever the server sends must be understood by the client. In Figure 3-4, the content chunks would be in HTML because the chunks would be injected directly into the HTML page. HTML, though, is not the only format that can be sent to and from the server.

The following formats are covered in this chapter:

- *HTML:* The server can send HTML to the client directly. The received HTML would not be processed, but injected directly into the HTML page. This is a blind processing approach in that the client has no idea what the HTML does, and knows only that it should be injected into a certain area of the HTML document. Injecting HTML directly is a very simple and foolproof way of building content. The client has to do no processing and needs to know only the destination area of the HTML content. If processing is necessary, the received content (if it is XML compliant) would also be available as an instantiated object model. Using the instantiated object model, it is possible to manually manipulate the received HTML content. It is advised that the HTML content sent to the client be XHTML compliant (HTML that implements a particular XML schema) or at least XML compliant.

- *Images:* It is not possible to directly send images because images are binary, and the `XMLHttpRequest` object cannot process binary data. Typically, image references are sent as HTML tags that are injected into the HTML document, resulting in the remote image to be loaded. It is possible to download and reference binary data if the data has been encoded and decoded by using Base64 encoding. However, manipulating binary data directly is not recommended because that will create more problems than it solves.

- *JavaScript:* The server can send JavaScript to the client that can be executed by using the JavaScript `eval` statement, and the client can send persisted JavaScript objects to the server for further processing. A first impression may be that executing arbitrary JavaScript presents a security problem. It is not typically a problem because the JavaScript engines in all browsers use the same origin and sandbox policies. Sending arbitrary JavaScript to execute could be a security problem if there is a bug in the JavaScript engine. Sending JavaScript is desirable if you want to dynamically execute and add logic on the client that was not loaded when the initial HTML page was loaded. It is a very powerful method of enhancing the functionality of a client without the client having to be aware of that. For example, let’s say an HTML form element needs validation. Because different users have different validations, it would not be desirable to send all validation implementations to the client. A solution would be to let the user decide which HTML form element they are presented with, and then dynamically download the validation of the form element as a content chunk. Be forewarned, though, that sending JavaScript chunks could open up your application to hackers. So think before using this technique.

- *XML:* The preferred approach is to send and receive XML. The XML can be transformed or parsed on the client side by manipulating the XML object model, or an Extensible Stylesheet Language Transformations (XSLT) library can be used to transform the XML into another object model such as HTML. The reason XML is preferred is that XML is a known technology and the tools to manipulate XML are well defined, working, and stable. XML is a very well established technology that you can search, slice, dice, persist, and validate without having to write extra code. Some do consider XML heavy because of the angle brackets and other XML character tokens. The advantage, though, is that when a server-side application generates XML, it can be processed by a web-browser-based client or a non-GUI-based browser. The choice of how to parse the XML and what information to process depends entirely on the client, so long as the client knows how to parse XML. XML is flexible and should be used. Throughout this book, XML will be used extensively and is considered the premier data exchange format.

There are other data exchange formats, such as JavaScript Object Notation (JSON).[1] However, I advise when those formats are chosen that you carefully consider the ramifications. It is not that I find them badly designed or improper. What concerns me about these other data exchange formats is that they do not provide as extensive an environment as XML for processing, searching, validating, and generating. For example, using XPath I search for specific elements in XML without having to parse the entire XML document. Granted, XML might in certain conditions not have the same performance levels as, let's say, JSON. For those readers who do not care whatsoever for the diversity of XML and are sure that they will never need it, JSON might be the right technology. However, I do not cover other technologies such as JSON in the scope of this pattern or in the rest of the book.

Now that you understand the architecture, you're ready to see some implementations that demonstrate how that architecture is realized.

Implementation

When implementing the Content Chunking pattern, the sequence of steps outlined earlier needs to be followed (event, request, response, and injection). The logic is easily implemented by using the `Asynchronous` type, because the `Asynchronous` type can be called by an HTML event and there is an explicit response method implementation. The example implementations that follow will illustrate how to generate the events by using HTML, call the functions, generate requests by using `XMLHttpRequest`, and process responses by using Dynamic HTML and JavaScript techniques.

Implementing the HTML Framework Page

The implementation of the Content Chunking pattern requires creating an HTML page that serves as the framework. The idea behind the framework page is to provide the structure into which content can be chunked. The framework page is the controller and provides a minimal amount of content.

The following HTML code is an example HTML framework page that will dynamically inject HTML content into a specific area on the HTML page:

1. `http://www.crockford.com/JSON/index.html`

```
<html>
<head>
<title>Document Chunk HTML</title>
<script language="JavaScript" src="/lib/factory.js"></script>
<script language="JavaScript" src="/lib/asynchronous.js"></script>
<script language="JavaScript" type="text/javascript">
var asynchronous = new Asynchronous();
asynchronous.complete = function(status, statusText, responseText, responseXML) {
    document.getElementById("insertplace").innerHTML = responseText;
}
</script>
</head>
<body onload="asynchronous.call('/chap03/chunkhtml01.html')">
<table>
    <tr><td id="insertplace">Nothing</td></tr>
</table>
</body>
</html>
```

In the HTML code, the class `Asynchronous` is instantiated and the `asynchronous.complete` property is assigned a function callback. How the `Asynchronous` class works and which properties need to be assigned was discussed in Chapter 2. The instantiation of `asynchronous` occurs as the HTML page is loading. After the page has loaded and is complete, the event `onload` is executed—which is the event step of the pattern implementation. The `onload` event calls the `asynchronous.call` method to execute an `XMLHttpRequest` request to download an HTML chunk—which is the request step of the pattern implementation.

After the request has completed, a response is generated that when received by the client results in the method `asynchronous.complete` being called. The received response is the response step of the pattern implementation. In the example, the method `asynchronous.complete` is assigned an anonymous JavaScript function. In the implementation of the anonymous function, the method `getElementById` is called to insert the `XMLHttpRequest` results into an HTML element. The HTML element is located by the identifier `insertplace`, which happens to be the HTML tag `td`. The referencing of the Dynamic HTML element and its assignment using the `innerHTML` property is the HTML injection—which represents the injection step of the pattern implementation.

In the example, it is odd that after the HTML page is downloaded, processed, and considered complete, another piece of logic is called. The other piece of logic is used to retrieve the rest of the content in the form of a chunk. The server-side code could have generated the complete page in the first place. However, it was illustrated in this fashion to show how simple the implementation of the Content Chunking pattern can be. The example illustrated reacting to the `onload` page event, but any event could be used. For example, examples in Chapter 2 used the button `onclick` event. A script could even simulate events by using the `Click()` method.

This example illustrated separation of the HTML page's appearance from its logic. The framework HTML page could be realized by an HTML designer. For the area where content is injected, the HTML designer would need only to add a placeholder token identifier such as `Nothing`. A server-side web application programmer creates the generated content that replaces the placeholder. The HTML designer would not need to be concerned with any server programming technology because the framework HTML page would contain only client-side instructions.

The server-side web application programmer would not need to be concerned with the look of the HTML page, because the generated content does not contain any information that affects the look and feel. For testing purposes, the web application programmer focuses on logic, whereas the HTML designer focuses on look and workflow.

Injecting Content by Using Dynamic HTML

The magic of the example is the ability of Dynamic HTML to dynamically insert content in a specific location. Before Dynamic HTML, you would have to use frames or server-side logic to combine the multiple streams. In recent years, Dynamic HTML has been formally defined by the World Wide Web Consortium (W3C) in the form of the HTML Document Object Model (DOM). The W3C HTML Document Object Model is not as feature rich as the object models made available by Microsoft Internet Explorer and Mozilla-derived browsers. For the scope of this book, the object model used is a mixture of the W3C HTML Document Object Model and functionality that is available to most browsers (for example, Mozilla-derived browsers and Microsoft Internet Explorer).

Going back to the previous example, the attribute `id` uniquely identifies an element in the HTML page. Using the uniquely identified element, a starting point is described from where it is possible to navigate and manipulate the HTML object model. The other way to find a starting point is to explicitly retrieve a type of tag and then find the HTML element that provides the starting point. Regardless of which approach is used, one of these two ways must be used to retrieve a starting point. Some readers may say that you could use other properties or methods, but those properties and methods are considered non-HTML-DOM compliant and hence should be avoided.

The following HTML code illustrates how to find a starting point using the two approaches:

```
<html>
<head>
<title>Document Chunk HTML</title>
<script language="JavaScript" src="/lib/factory.js"></script>
<script language="JavaScript" src="/lib/asynchronous.js"></script>
<script language="JavaScript" type="text/javascript">
var asynchronous = new Asynchronous();
asynchronous.complete = function(status, statusText,
    responseText, responseXML) {
    document.getElementsByTagName("table")[ 0].rows[ 0].cells[ 0].innerHTML
        = responseText;
    document.getElementById("insertplace").innerHTML = responseText;
}
</script>
</head>
<body onload="asynchronous.call('/chap03/chunkhtml01.html')">
<table>
    <tr><td>Nothing</td></tr>
    <tr><td id="insertplace">Nothing</td></tr>
</table>
</body>
</html>
```

In the implementation of the anonymous function for the `asynchronous.complete` method, two methods (`getElementsByTagName`, `getElementById`) are used to inject content into a Dynamic HTML element. The two methods retrieve an element(s) that represents a starting point.

The method `getElementsByTagName` retrieves all HTML elements of the type specified by the parameter to the method. In the example, the parameter is `table`, which indicates to search and retrieve all `table` elements in the HTML document. Returned is an instance of `HTMLCollection` of all HTML elements, and in the case of the example contains all of the `table` elements. The class `HTMLCollection` has a property, `length`, that indicates how elements have been found. The found elements can be referenced by using the JavaScript array notation (square brackets), where the first element is the zeroth index.

In the example, right after the method identifier `getElementsByTagName("table")` is a set of square brackets (`[0]`) used to retrieve the first element from the collection. The zeroth index is arbitrarily referenced, meaning the first found table is referenced. In the example, some index was used. The correct index is referenced because the example HTML page only has a single table; therefore, the zeroth index will always be the correct index, meaning that the correct table, row, and cell are referenced. However, imagine a scenario of multiple tables. Then, referencing an arbitrary index may or may not retrieve the correct table. Even worse, if the Content Chunking pattern were called multiple times, the order of the found element collection could change and reference elements that were not intended to be referenced.

The method `getElementsByTagName` is best used when operations are executed on all found elements without trying to identify individual elements. Examples of such operations include the addition of a column in a table and modification of a style. The method `getElementById` is best used when an individual element needs to be manipulated.

It is possible when using the method `getElementsByTag` to retrieve all elements in the HTML document, as illustrated in the following example:

```
var collection = document.getElementsByTag("*");
```

When the method `getElementsByTag` is called with an asterisk parameter, it means to return all elements of the HTML document. Some may note that using the property `document.all` does the exact same thing. Although this is true, it is not DOM compliant and will generate a warning by any Mozilla-based browser.

Focusing on the following code from the example:

```
document.getElementsByTagName("table")[ 0].rows[ 0].cells[ 0].innerHTML
```

The identifiers after the square brackets of the method `getElementsByTagName` represent a series of properties and methods that are called. These properties and methods relate directly to the object retrieved, which in this case is a table that contains rows, and the rows contain cells. Had the retrieved element not been a table, the calling of the properties and methods would have resulted in an error.

Again from the example source code, let's focus on the following:

```
document.getElementById("insertplace").innerHTML = responseText;
```

The method `getElementById` retrieves an HTML element with an `id` attribute identical to the parameter of the method. The `id` attribute and parameter are case-sensitive. The result of the method `getElementById` is to retrieve the `td` tag with the `id` attribute value `insertplace`. When using the method `getElementById`, if there are multiple items with the same identifier on the HTML page, then only the first found element is retrieved. The other elements are not returned nor accessible because the method getElementById returns only a single HTML element instance. Unlike the getElementsByTagName method, the returned element is not guaranteed to be a certain type other than having the parameter identifier equal to the id attribute. As a result, the object model referenced after the getElementById method may or may not apply to the found element. In the case of the property innerHTML, that is not a problem because virtually all visible elements have the innerHTML property. What could be more problematic is if the identifier assumed the retrieved element were a table when in fact the element is a table cell. At that point, the object model referencing would result in an exception.

When writing JavaScript code that dynamically retrieves an HTML element(s), it is a good idea to test the found element before manipulating it. As a rule of thumb, when using getElementsByTag, you know what the HTML elements are but do not know where they are or what they represent. When using getElementById, you know what the found HTML element represents and where it is, but do not know the type and hence the object hierarchy.

Understanding the Special Nature of innerHTML

The property innerHTML is special in that it seems simple to use but can have devastating consequences. To illustrate the problem, consider the following HTML code:

```
<html>
<head>
<title>Document Chunk HTML</title>
<script language="JavaScript" type="text/javascript">
function GoodReplace() {
    document.getElementById("mycell").innerHTML = "hello";
}
function BadReplace() {
    document.getElementById("mytable").innerHTML = "hello";
}
function TestTable() {
    window.alert(document.getElementsByTagName(
        "table")[ 0].rows[ 0].cells[ 0].innerHTML);
}
</script>
</head>
<body>
<button onclick="GoodReplace()">GoodReplace</button>
<button onclick="BadReplace()">BadReplace</button>
<button onclick="TestTable()">TestTable</button>
<table id="mytable" border="1">
    <tr id="myrow"><td id="mycell">Nothing</td><td>Second cell</td></tr>
</table>
</body>
</html>
```

In this example, there are three buttons (GoodReplace, BadReplace, and TestTable), and the HTML elements table, table row, and row cell have added identifiers. The GoodReplace button will perform a legal HTML injection. The BadReplace button will perform an illegal HTML injection. And the TestTable button is used to test the validity of an object model. The TestTable button is used as a way of verifying the result of the HTML injection performed by either GoodReplace or BadReplace. Downloading the HTML page and presenting it in the browser results in something similar to Figure 3-5.

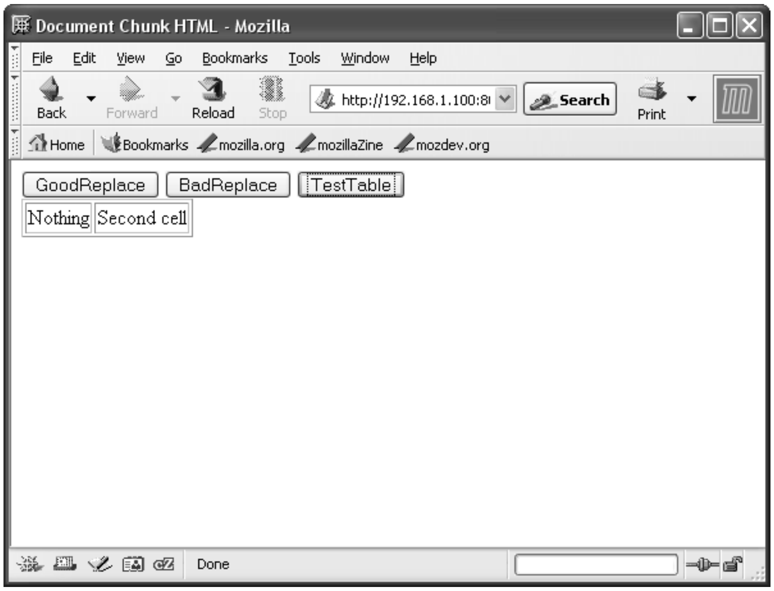

Document Chunk HTML - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://192.168.1.100:8
Search
Print
Home Bookmarks mozilla.org mozillaZine mozdev.org
GoodReplace BadReplace TestTable
Nothing Second cell
Done

Figure 3-5. *Initial generation of the HTML page*

To check that the HTML page is in a valid state, the button TestTable is clicked. Clicking the button calls the function `TestTable`, which tests whether the content within a table cell exists by outputting the content in a dialog box. The generated output appears similar to Figure 3-6.

The dialog box in Figure 3-6 confirms that the table cell contains the value Nothing. This means our HTML page is in a stable state. If the GoodReplace button is clicked, the function `GoodReplace` is called, which changes the table cell contents from Nothing to Hello. To verify that the HTML page is still valid, the TestTable button is clicked. If the HTML page is valid, a dialog with the text Hello should appear, and it does, as is illustrated in Figure 3-7.

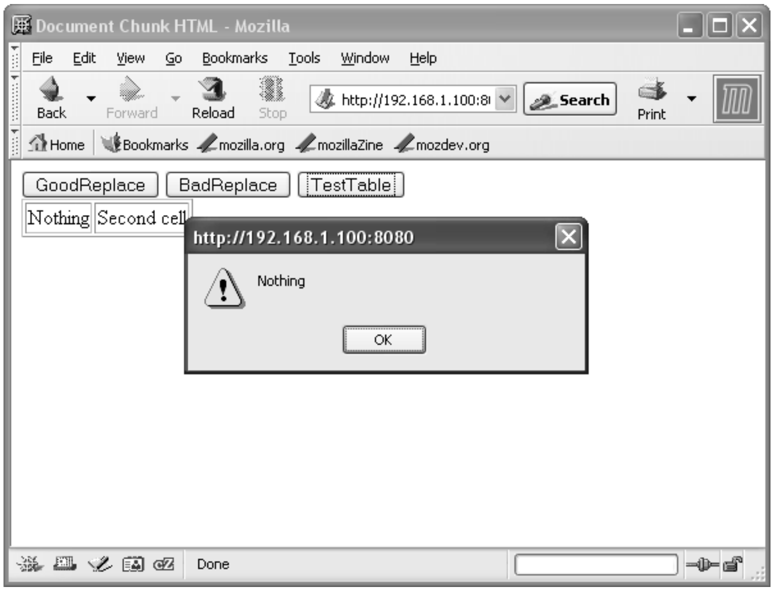

Document Chunk HTML - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop http://192.168.1.100:8l Search Print
Home Bookmarks mozilla.org mozillaZine mozdev.org
GoodReplace BadReplace TestTable
Nothing Second cell
http://192.168.1.100:8080
Nothing
OK
Done

Figure 3-6. *Displaying the contents of the cell* `mycell`

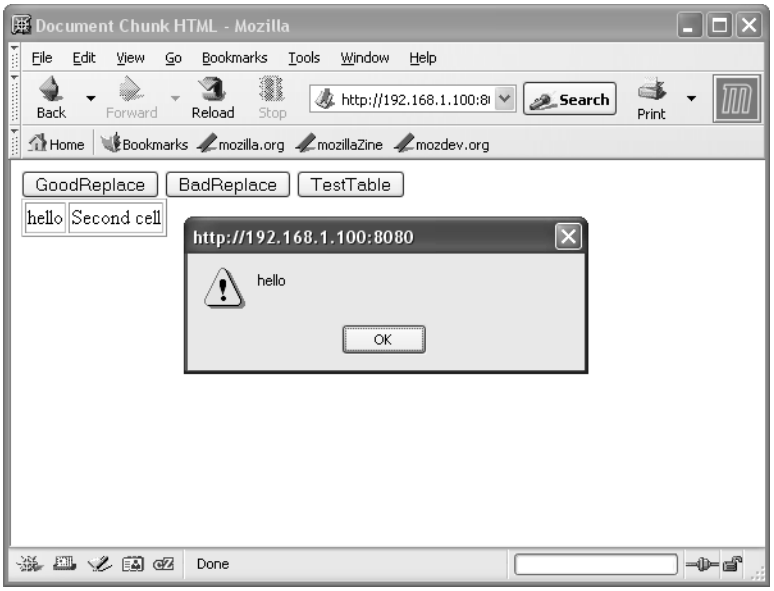

Document Chunk HTML - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop http://192.168.1.100:8l Search Print
Home Bookmarks mozilla.org mozillaZine mozdev.org
GoodReplace BadReplace TestTable
hello Second cell
http://192.168.1.100:8080
hello
OK
Done

Figure 3-7. *Modified contents of the cell*

For interest, let's add some complications by clicking the button BadReplace. Clicking BadReplace calls the function `BadReplace`, and that assigns the property `innerHTML` of the HTML table with other text. This means that the HTML content `<table><tr><td>...</table>` is changed to `<table>Nothing</table>`. The changed HTML is not legal and is displayed as shown in Figure 3-8.

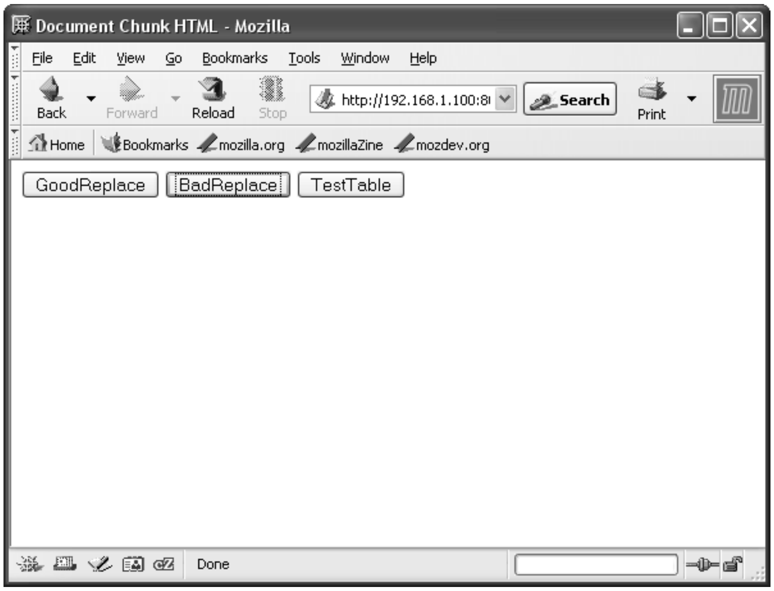
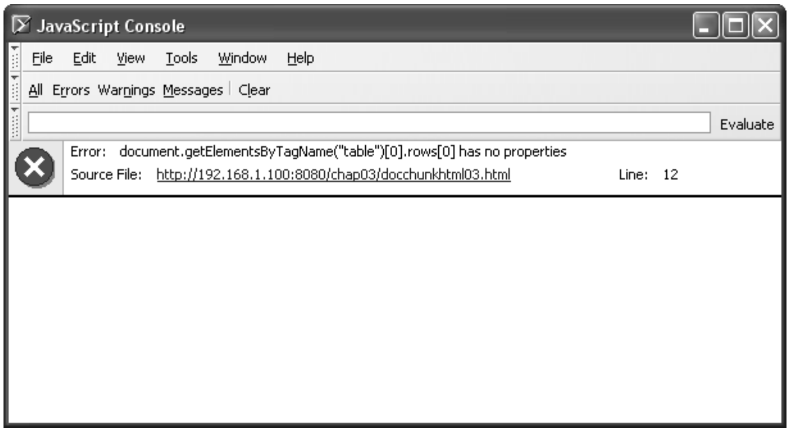

Document Chunk HTML - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://192.168.1.100:8
Search
Print
Home Bookmarks mozilla.org mozillaZine mozdev.org
GoodReplace
BadReplace
TestTable
Done

Figure 3-8. *Modified contents of the table after replacing the rows and cells*

Figure 3-8 illustrates that the table rows have been replaced with nothing. If the TestTable button is clicked to validate the state, an error is generated, as illustrated in Figure 3-9.


JavaScript Console
File Edit View Tools Window Help
All Errors Warnings Messages Clear
Evaluate

```
Error:   document.getElementsByTagName("table")[0].rows[0] has no properties
Source File:  http://192.168.1.100:8080/chap03/docchunkhtml03.html        Line:  12
```

Figure 3-9. *Object model exception*

The exception is important and relates to how the property `innerHTML` operates. When HTML content is assigned using the `innerHTML` property, the data is text based. When retrieving the value of the `innerHTML` property, child elements are converted into a text buffer. Assigning the `innerHTML` property means replacing the child elements with the HTML text defined by the assignment. Then that new HTML text is converted into a series of HTML elements that are presented to the user. The functions `GoodReplace` and `BadReplace` are examples of manipulating the innerHTML property. However, things can run amok if the innerHTML property is manipulated when it should not be manipulated or when doing so will violate the structure of the HTML. For instance, as illustrated in the example you cannot create a table without rows or cells.

Another way to interact with the HTML Document Object Model is to use individual elements that are instantiated, manipulated, and deleted. Using the Document Object Model, it is much harder to mess up because this model supports only certain methods and properties. When using the HTML Document Object Model, it is not as simple to arbitrarily remove all the rows and replace them with text. There are no methods on the table object model to create a construct, as illustrated in Figure 3-8.

It is important to remember that entire chunks of HTML content are replaced when using the Content Chunking pattern. So even though the property innerHTML is powerful and flexible, replacing the wrong chunk at the wrong time will result in an incorrectly formatted HTML page. What you need to remember is that when referencing HTML elements in the context of the pattern, only framework HTML elements used to contain content chunks should be referenced. As a pattern rule, script in the HTML framework page should not directly reference injected elements, as that would create a dynamic dependency that may or may not work. If such a dependency is necessary, encapsulate the functionality and call a method on the injected elements. JavaScript allows the assignment of arbitrary functions on HTML elements.

Identifying Elements

It was previously mentioned that when finding elements by using a tag type, it is not possible to know the identifier; and when finding elements using the identifier, it is not possible to know the tag type. Regardless of how the elements have been found, they are considered a starting point from which manipulations can happen. Based on the starting point, a script can navigate the parent or the child elements by using a number of standard properties and methods.

These standard properties and methods are available on virtually all HTML elements, and script writers should focus on using them when navigating a hierarchy, modifying the look and feel, or attempting to identify what the element is. Table 3-1 outlines properties that are of interest when writing scripts.

Table 3-1. *HTML Element Properties Useful for Writing Scripts*

Property Identifier	Description
`attributes[]`	Contains a read-only collection of the attributes associated with the HTML element. An individual attribute can be retrieved by using the method `getAttribute`. To assign or overwrite an attribute, the method `setAttribute` is used. To remove an attribute, the method `removeAttribute` is used.
`childNodes[]`	Is an instance of `NodeList` that most likely is referenced by using an array notation, but the array is read-only. To add a child node to the current element, the method `appendChild` is used. To remove a child node, the method `removeChild` is used; and to replace a child node, `replaceChild` is used.
`className`	Assigns a stylesheet class identifier to an element. A class type is very important in Dynamic HTML in that the look and feel of the element can be dynamically assigned.
`dir`	Indicates the direction of the text, either left to right (`ltr`) or right to left (`rtl`).

Table 3-1. *HTML Element Properties Useful for Writing Scripts*

Property Identifier	Description
`disabled`	Enables (`false`) or disables (`true`) an element. Useful when the script does not want a user to click a certain button or other GUI element before completing a required step.
`firstChild`, `lastChild`	Retrieves either the first child node or the last child node.
`id`	Is the identifier of the element used to find a particular element. For example, this property is referenced when a script calls the method `getElementById`.
`nextSibling`, `previousSibling`	Retrieves either the next or previous sibling. When used in combination with `firstChild` and `lastChild`, can be used to iterate a set of elements. This approach would be used to iterate a list in which the element is responsible for indicating what the next element should be—for example, when implementing a Decorator pattern or similar structure.
`nodeName`	Contains the name of the element, which in HTML means the tag (for example, `td`, `table`, and so on).
`nodeType`	Contains the type of element but is geared for use when processing XML documents. With respect to HTML, this property has very little use.
`nodeValue`	Contains the value of the data in the node. Again, this property has more use when processing XML documents. With respect to HTML, this property cannot be used as a replacement for `innerHTML`.
`parentElement`	Retrieves the parent element for the current element. For example, can be used to navigate to the table that contains a row cell.
`style`	Identifies the current style properties associated with the element and is an instance of `CSSStyleDeclaration` type.
`tabIndex`	Defines the tab stop of the element with respect to the entire HTML document.
`tagName`	Identifies the tag of the current element. Use this property when attempting to figure out the element type after the element has been retrieved via the method `getElementById`.

Binary, URL, and Image Chunking

Chunking binary or images in their raw form using the `XMLHttpRequest` object is rather complicated because the data that is read turns into gibberish. The `XMLHttpRequest` properties `responseText` and `responseXML` expect either text or XML, respectively. Any other data type is not possible. Of course there is an exception: Base64-encoding binary data that is encoded as text, and then retrieving the text by using the `XMLHttpRequest` object. Another solution is not to manage the binary data but to manage the reference of the binary data. In the case of the `img` tag, that means assigning the `src` attribute to where an image is located.

Images are downloaded indirectly. To understand how this works, consider an application that uses `XMLHttpRequest` to retrieve a document containing a single line. The single line is a URL to an image file.

Here is the implementation of the example program:

```
<html>
<head>
<title>Document Chunk Image HTML</title>
<script language="JavaScript" src="/lib/factory.js"></script>
<script language="JavaScript" src="/lib/asynchronous.js"></script>
<script language="JavaScript" type="text/javascript">

var asynchronous = new Asynchronous();
asynchronous.complete = function(status, statusText, responseText, responseXML) {
    document.getElementById("image").src = responseText;
}

</script>
</head>
<body>
<button onclick="asynchronous.call('/chap03/chunkimage01.html')">Get Image</button>
<br>
<img id="image" />
</body>
</html>
```

The `img` tag is used to reference an image. The `img` tag is in most cases defined by using a `src` attribute that references an image location. In the example, the `src` attribute does not exist, and instead an `id` attribute exists. When the HTML page is downloaded and presented, a broken image is displayed because there is no image associated with the `img` tag. To make the `img` tag present an image, the Get Image button is clicked to make a request to retrieve the single-line file containing the URL of the image. When the `XMLHttpRequest` has downloaded the single-line file, the function implementation for `complete` is called and the attribute/property `src` is assigned to the URL of the remote image. Thus the browser updates itself, loading the image and displaying it.

The single-line file is stored at the URL `/chap03/chunkimage01.html`, and its content is defined as follows:

```
/static/patches01.jpg
```

When the previously outlined HTML page with an undefined `src` attribute is loaded, Figure 3-10 is generated.

Figure 3-10 shows a small box below the Get Image button that indicates a broken `img` tag, because there is no loaded image. When the Get Image button is clicked, the link of the image is downloaded and assigned to the `img` tag, which causes the image to be loaded. Figure 3-11 is the regenerated HTML page.

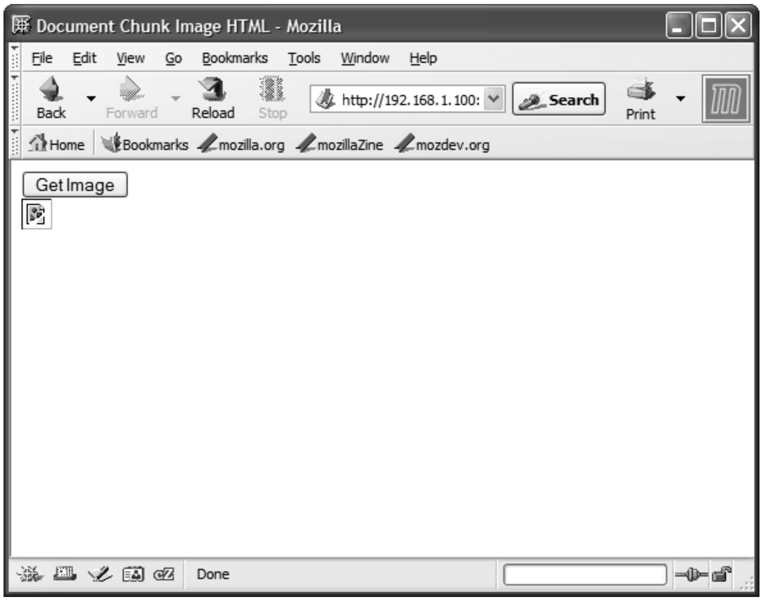

Document Chunk Image HTML - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://192.168.1.100:
Search
Print
Home Bookmarks mozilla.org mozillaZine mozdev.org
Get Image
Done

Figure 3-10. *Initial HTML page generated without an image*

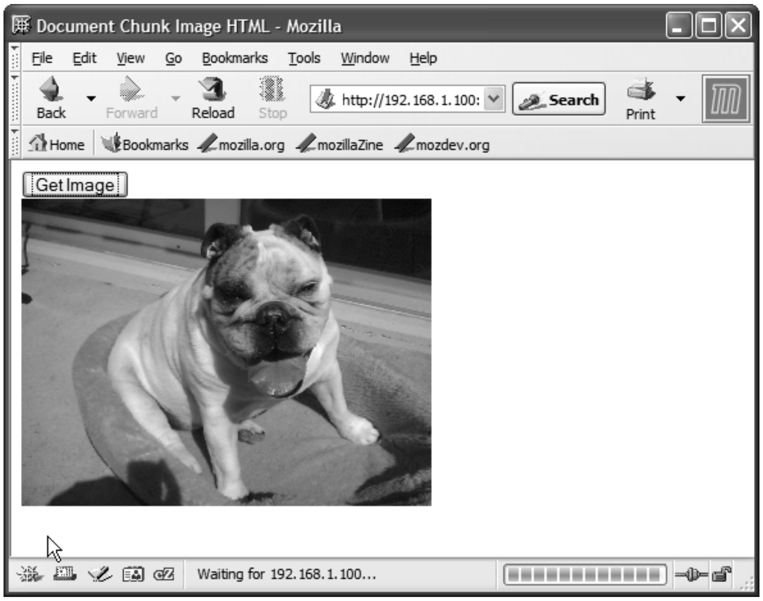

Document Chunk Image HTML - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://192.168.1.100:
Search
Print
Home Bookmarks mozilla.org mozillaZine mozdev.org
Get Image
Waiting for 192.168.1.100...

Figure 3-11. *The HTML page after the image has been downloaded*

It seems a bit odd to download and assign links that are then processed by the web browser. This indirect approach is done not to illustrate how complicated a web application can be made. The indirect technique is necessary because directly downloading binary data is not possible. But all is not lost, because of the way that the browser caches images. If an image is referenced and downloaded, the image stays in the browser's cache. If the image is referenced a second time, the image is retrieved from the cache. Of course this happens only if the HTTP server implements caching. There is a downside: If a request is made for a URL that references an image, two HTTP requests are required: one to download the content that contains the URL of the image, and the image itself. If both requests are using HTTP 1.1, which most likely is the case, the requests will be inlined using a single connection.

Another variation of the illustrated strategy is to download not a URL but the entire HTML to create an image. The strategy does not save a request connection, but provides a self-contained solution that involves no additional scripting. The following HTML code snippet illustrates how the entire `img` HTML tag is downloaded:

```
<img src="/static/patches01.jpg" />
```

When injecting both the `img` tag and its appropriate `src` attribute, the browser will dynamically load the image as illustrated in the previous example. The advantage of injecting the HTML is that the server side could inject multiple images or other types of HTML. Additionally, by injecting the entire `img` tag, there is no preliminary stage where a broken image is generated. However, either approach is acceptable, and which is used depends on the nature of the application. When injecting HTML, there might be a flicker as the HTML page resizes. When you assign the `src` property, there is no flicker, but an empty image needs to be defined or the image element needs to be hidden.

JavaScript Chunking

Another form of chunking is the sending of JavaScript. Sending JavaScript can be very effective because you don't need to parse the data but only execute the JavaScript. From a client script point of view it is very easy to implement. For reference purposes, do not consider downloading JavaScript faster than manually parsing and processing XML data and then converting the data into JavaScript instructions. JavaScript that is downloaded needs to be parsed and validated before being executed. The advantage of using the JavaScript approach is simplicity and effectiveness. It is simpler to execute a piece of JavaScript and then reference the properties and functions exposed by the resulting execution.

Executing JavaScript

Consider the following HTML code that will execute some arbitrary JavaScript:

```
<html>
<head>
<title>JavaScript Chunk HTML</title>
<script language="JavaScript" src="/lib/factory.js"></script>
<script language="JavaScript" src="/lib/asynchronous.js"></script>
<script language="JavaScript" type="text/javascript">
```

```
var asynchronous = new Asynchronous();
asynchronous.complete = function(status, statusText, responseText, responseXML) {
    eval(responseText);
}

</script>
</head>
<body>
<button onclick="asynchronous.call('/chap03/chunkjs01.html')">Get Script</button>
<table>
    <tr><td id="insertplace">Nothing</td></tr>
</table>
</body>
</html>
```

When the user clicks the Get Script button, an `XMLHttpRequest` request is made that retrieves the document `/chap03/chunkjs01.html`. The document contains a JavaScript chunk that is executed by using the `eval` function. The following chunk is downloaded:

```
window.alert("oooowweee, called dynamically");
```

The example chunk is not very sophisticated and pops up a dialog box. What would concern many people with arbitrarily executing JavaScript code is that arbitrary JavaScript code is being executed. An administrator and user might be concerned with the security ramifications because viruses or Trojans could be created. However, that is not possible because JavaScript executes within a sandbox and the same origin policy applies. Granted, if a developer bypasses the same origin policy, security issues could arise.

When receiving JavaScript to be executed, a simple and straightforward implementation is to dynamically create a JavaScript chunk that executes some methods. The JavaScript chunks make it appear that the web browser is doing something. For example, the JavaScript chunk downloaded in the previous example could be used to assign the `span` or `td` tag as illustrated here:

```
document.getElementById("mycell").innerHTML = "hello";
```

The generated script is hard-coded in that it expects certain elements to be available in the destination HTML page.

Generating a JavaScript That Manipulates the DOM

Earlier you saw the image generation solution in which an image was broken and then made complete by downloading a valid link. It is also possible to download an image by modifying the Dynamic HTML object model. You modify the object model by using a JavaScript chunk to insert the `img` tag. The following is an example image JavaScript chunk that creates a new `img` tag and chunks it into the HTML document:

```
var img = new Image();
img.src = "/static/patches01.jpg";
document.getElementById("insertplace").appendChild(img);
```

In this example, the variable `img` is an instance of an `Image`, which cross-references to the HTML tag `img`. The property `src` is assigned the URL of the image. The last line of the code chunk uses the method `appendChild` to add the instantiated `Image` instance to the HTML document. Not associating the variable `img` with the HTML document will result in an image that is loaded but not added to the HTML document, and hence not generated. The resulting generated HTML page is shown in Figure 3-12.

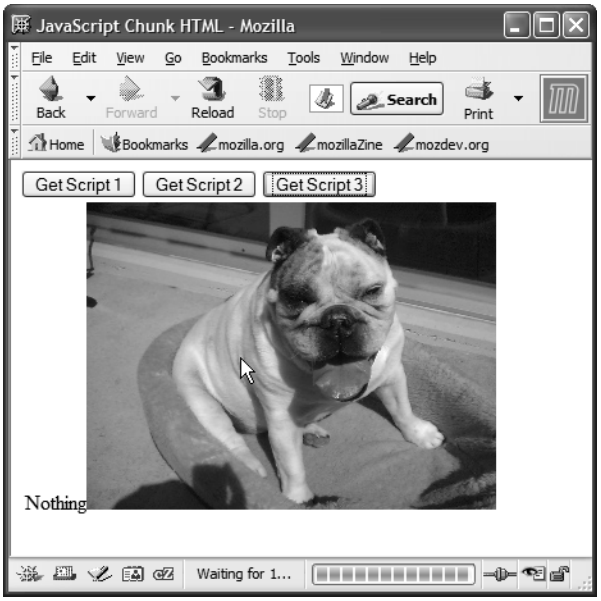

JavaScript Chunk HTML - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop Search Print
Home Bookmarks mozilla.org mozillaZine mozdev.org
Get Script 1 Get Script 2 Get Script 3
Nothing
Waiting for 1...

Figure 3-12. *Generated HTML page after image has been inserted*

Figure 3-12 is not that spectacular because it illustrates yet again how an image can be added to an HTML page. What is of interest is that the text Nothing has remained and is not replaced as in previous examples. The reason is that the method `appendChild` was used (and not `replaceChild` or `removeChild`, and then `appendChild`).

The advantage of using the Dynamic HTML object model approach is that it enables images or arbitrary actions to be downloaded in the background that can at the script's choosing be displayed.

Instantiating Objects

Another type of JavaScript chunk that can be downloaded are object states. By using an object state, you can add a level of indirection, allowing functionality to be added during the execution of the HTML page. In all of the past example HTML code pieces, the initial HTML page had to have all the scripts and know the URLs of the resources that were retrieved. Using an indirection, the JavaScript on the client side does not need to know the specifics of a URL or data structure. The client references a general piece of code that is executed. The general piece of code is managed by the server, and contains specific instructions to do something that the client was not programmed to do. Using indirection, it is possible to add functionality to the client that the client did not possess at design time.

Consider the following example HTML page:

```
<html>
<head>
<title>JavaScript Chunk HTML</title>
<script language="JavaScript" src="/lib/factory.js"></script>
<script language="JavaScript" src="/lib/asynchronous.js"></script>
<script language="JavaScript" type="text/javascript">
var asynchronous = new Asynchronous();
asynchronous.complete = function(status, statusText, responseText, responseXML) {
    eval(responseText);
    dynamicFiller.makeCall(document.getElementById("insertplace"));
}
</script>
</head>
<body>
<button onclick="asynchronous.call('/chap03/chunkjs04.js')">Start Process</button>
<table>
    <tr><td id="insertplace">Nothing</td></tr>
</table>
</body>
</html>
```

As in previous examples, a variable of type `Asynchronous` is instantiated. The `button` is wired to make an asynchronous method call with the URL `/chap03/chunkjs04.js`. When the request receives the JavaScript chunk, it is executed via the `eval` statement. After the `eval` statement has returned, the method `dynamicFiller.MakeCall` is made. The call to the method `dynamicFiller.MakeCall` is a general piece of code. In the implementation of the `dynamicFiller.MakeCall` method is the specific code managed by the server. Referencing the `dynamicFiller.MakeCall` method is done using an incomplete variable; that is, the initial script includes no definition of the variable `dynamicFilter`. Of course, a loaded and processed script cannot reference an incomplete variable because that would generate an exception. But what a script can do is load the implementation just before an incomplete variable is used. That is what the example HTML page has illustrated. For those wondering, there is no definition of `dynamicFilter` in the files `factory.js` or `asynchronous.js`. Incomplete variables, types, and functions are possible in JavaScript, allowing a script to be loaded and processed without generating an exception.

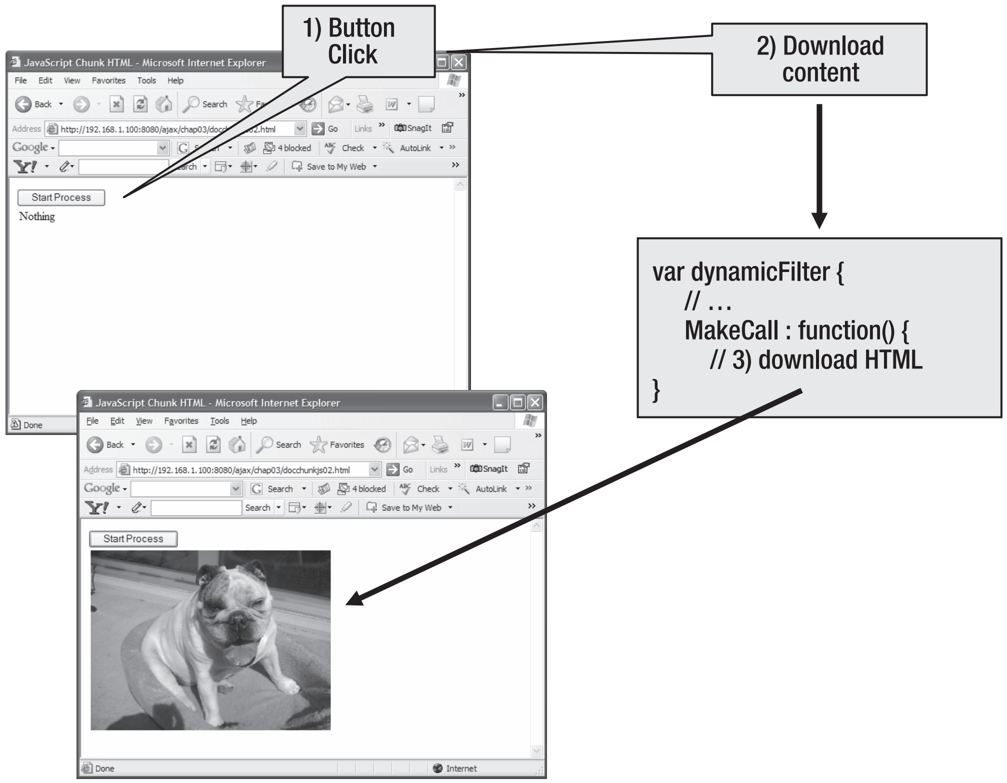
The following source code implements the incomplete `dynamicFiller` variable:

```
var dynamicFiller = {
    generatedAsync : new Asynchronous(),
    reference : null,
    complete : function(status, statusText, responseText, responseXML) {
        dynamicFiller.reference.innerHTML = responseText;
    },
    makeCall : function(destination) {
        dynamicFiller.reference = destination;
        dynamicFiller.generatedAsync.complete = dynamicFiller.complete;
        dynamicFiller.generatedAsync.call('/chap03/chunkjs05.html');
    }
}
```

The example JavaScript source code is formatted using object initializers. An object initializer is the persisted form of a JavaScript object. You should not equate an object initializer with a JavaScript class definition; they are two entirely separate things. When an object initializer is processed, an object instance is created, and the identifier of the object instance is the variable declaration. In the example, the variable `dynamicFiller` is the resulting object instance.

The variable `dynamicFiller` has two properties (`generatedAsync` and `reference`) and two methods (`complete` and `makeCall`). The property `generatedAsync` is an instantiated `Asynchronous` type and is used to make an asynchronous call to the server. The property `reference` is the HTML element that will be manipulated by the method `complete`. The method `makeCall` is used to make an `XMLHttpRequest`, and the parameter destination is assigned to the property reference.

Putting all the pieces together, the HTML framework code contains general code that references an incomplete variable. To make a variable complete, the JavaScript content is downloaded and executed. The complete variable contains code to download content that is injected into the framework page. Figure 3-13 illustrates the execution sequence of events.


1) Button Click
2) Download content
JavaScript Chunk HTML - Microsoft Internet Explorer
Start Process
Nothing
Done
JavaScript Chunk HTML - Microsoft Internet Explorer
http://192.168.1.100:8080/ajax/chap03/docchunkjs02.html
Start Process
Done
Internet

```
var dynamicFilter {
    // …
    MakeCall : function() {
        // 3) download HTML
}
```

Figure 3-13. *Sequence of events when downloading and executing JavaScript*

In Figure 3-13, the initial page is downloaded by clicking the button. The downloaded content is JavaScript, and the initial page has no idea what the content does. When the content has been downloaded, it is executed. The HTML framework page has coded the referencing of the variable `dynamicFilter` and the calling of the method `MakeCall`. The `MakeCall` method does not exist when the HTML framework page is executed, and is available when the downloaded content is executed. The downloaded content that is executed downloads yet another piece of content that is injected into the HTML page. The result is the loading of an image where the text Nothing was.

The role of the HTML framework page has changed into a bootstrap page that loads the other code chunks. The other code chunks are purely dynamic and contain references and code that the HTML framework page does not know about. The advantage of this implementation is that the document can be loaded incrementally by using pieces of dynamic code that are defined when they are loaded. The Content Chunking pattern defines the loading of content dynamically. But the additional use of JavaScript makes it possible to dynamically define the logic that is used by the HTML framework page.

Pattern Highlights

The following points are the important highlights of the Content Chunking pattern:

- An HTML page is the sum of an HTML framework page and content chunks.
- The HTML framework page is responsible for organizing, referencing, and requesting the appropriate chunks. It should act as a mediator for the individual chunks. The HTML framework page delegates the processing of the chunks to another piece of code.
- The content chunks are uniquely identified by a URL. Content chunks that are distinct do not use the same URLs. Content chunks are used to implement functionality that is determined by the user.
- Content chunks should be one of three types: XML (preferred), HTML (preferred XHTML), or JavaScript. There are other formats, but they are not covered in this book and their use should be carefully considered.