Various client-server communication mechanisms in an Ajax-based web application

Yoav Rubin
Gal Shachor

June 22, 2010

A key part of any Asynchronous JavaScript and XML (Ajax)-based web application is the communication layer between the client and the server. To implement this layer, you need to understand the various communication mechanisms that browsers provide, as well as each mechanism's pros and cons. In this article, learn to make the correct match between the specific communication needs of an application and the appropriate mechanism. Detailed examples show you how to create a communication layer that can meet these different client-server communication needs.

Introduction

Modern web applications are all based on various Ajax-related concepts. The use of Ajax techniques led to an increase in interactive or dynamic interfaces on web pages. The Ajax revolution began with the notion that web applications can retrieve data from the server asynchronously in the background, and interaction between the web page and the server is not limited to the moment when the page is fetched. The web page concept extended into a long-living web application that interacts with the user through ongoing communication with the application's back end. A few examples for what this ongoing communication allows are:

- Sending and receiving of information
- Ad-hoc input validation (for example, password strength)
- Auto-completion of user input based on rules and analysis done on the server

To perform the tasks related to the client-server interaction, an application needs an optimal communication layer that provides the proper communication mechanism for each communication task.
In this article, learn about the issues to consider when constructing a communication layer and explore the different mechanisms you should build in it.

**Doing it old school**

In the olden days, communication between a web page and the server was considered a hack. It was possible only by using different HTML elements in ways they were not intended to be used. The key aspect of these elements, which allowed for their usage (or abuse), is that they are intended to fetch a file from a server. Then, the browser is responsible for interpreting the file based on the type of element. The elements are:

- **img**  
  To fetch an image file
- **script**  
  To fetch a JavaScript™ file
- **iframe**  
  To display and possibly fetch an HTML file

These elements are intended to be part of a website's markup. Yet, by using JavaScript to do some DOM manipulation, you can inject the elements in a dynamic fashion and interact with a server as part of the page's life cycle. The next three sections describe how to use these elements.

**Fire and forget: Abusing the <img> element**

The main use of the `img` element is to fetch an image from a server and display it. To fetch the image, the browser creates a GET request with the URL that is the value of the element's `src` attribute. This value is not limited by the "same origin policy" of the browsers; therefore, the image's URL is not limited to the domain of the page that displays the image.

**What you can and cannot do**

It's possible to use the `img` element to perform GET calls from any URL. You can then, logically, invoke a service on a different server.

**Beware of caching**

Remember, GET calls can be cached either on the browser or on any network node along the way. If you use an `<img>` element to invoke a call, you might need to create a URL that cannot be cached (for example, by adding a random URL parameter to it).

However, the browser assumes that the response for that GET call is an image file and handles it as such (by presenting it on the screen). If the response is not in fact an image file, you can simply not add the img element to the DOM tree, because the GET request is created when the `src` attribute is set (regardless of whether the img element was added to the page).

You can use the `img` element mainly for “fire-and-forget” type of services, in which the response that the server returns is of no interest to the client. It's suited for a service that follows the rule of “what happens in the server, stays in the server.”

Listing 1 shows how to perform the fire-and-forget call.
Various client-server communication mechanisms in an Ajax-based web application

Listing 1. Invoke a service with fire and forget

```javascript
function fireAndForgetService(targetUrl){
    // first we create an img object
    var imgNode = document.createElement("img");
    // then we set its src attribute to the url of the service we'd like to invoke
    // when the next code line is executed the browser creates a GET request
    // and sends it to targetURL
    imgNode.src = targetUrl;
}
// calling the function with any url – cross site scripting is possible here
fireAndForgetService("http://www.theTargetUrl.com/doSomething?param1Name:param1Value");
</script>

Fetch and execute: Using the <script> element

Almost everything previously mentioned about the img element would work the same if you use the script element instead. This communication mechanism is also not limited by the "same origin policy" of the browsers.

One way in which the img and script elements differ, though, is that when a script element is used, the browser expects to receive the JavaScript code that can be executed. This is a very powerful mechanism that shouldn't be used lightly. The fetched script is invoked and has access to everything that the page can access, including cookies, DOM, history, and so forth.

What you can and cannot do

You should define some sort of a protocol between the code that runs in the browser and the fetched script. If you're fetching a script from your own domain, then most likely this code is familiar with the various functions, utilities, and constraints of the application's code. The protocol can basically be an integration of two components of the same application. The server can receive, as part of the request, part of the code that should be executed in the browser and might do various manipulations on it, such as handling internationalization restrictions or other user-specific adaptations.

JSON

JavaScript Object Notation (JSON) is a way to describe a JavaScript object. This is a very powerful data format, which can be quickly transformed into a JavaScript object in the browser. Related topics has more information about JSON.

Things get more interesting when you need to fetch code from another server. In this case (and if that domain is trustworthy), the fetched script is unfamiliar with the constructs of the application, so static data is simply all that can be fetched. To handle this data in your application, you need to somehow integrate the fetched content with the application's code. You can use the JSONP concept, which basically states that the fetched code is a JSON object that is wrapped (or padded) with a function call that accepts this object. (This call is known as the callback.) The name of that function is sent as a URL parameter as part of the fetched script's URL, and it is up to the recipient's domain to provide the JSON object wrapped with the name of that function.

Listing 2 demonstrates how to perform a client-server communication based on JSONP.
Listing 2. JSONP approach

```
<script type="text/javascript">
// the next function receives the following arguments:
// targetUrl - the url from which data is fetched and handled later as
// jsonp
// jsonpName - the name of the url parameter that the target url accepts and
// knows to read from the callback name
// callbackName - the name of the function that will handle the returned
// json object
function invokeJSONP(targetUrl, targetDomainJsonpName, callbackName) {
    // first we create a script object
    var scriptNode = document.createElement("script");
    // set its type so upon return it would be executed
    scriptNode.type = "text/javascript";
    // set its src attribute to the url of the fetched script
    // and add to the url the callback name
    scriptNode.src = targetUrl + "?" + targetDomainJsonpName + "=" + callbackName;
    // adding the script to the page to get it up and running
    document.getElementsByTagName("head")[0].appendChild(scriptNode);
}
// calling the function with any url - cross site scripting is possible here
function handleJsonp(infoObject) {
    validateInfoObject(infoObject);
    handleInfoObject(infoObject);
}
invokeJSONP("http://targetUrl.com/provideJsonpData","jsonpCallback", "handleJsonp");
</script>
```

One tiny difference is worth mentioning. When you use the `<img>` element, you don’t need to add it to the DOM tree. In contrast, when you do script fetching, the GET request will not be created unless the `<script>` element is added to the DOM tree.

It is up to the target domain to publish its APIs for JSONP calls, especially the parameter name that is in the application, which should provide the name of the callback. The other part of the APIs should include the structure of the JSON object that is sent in the response body.

In Listing 2, the target domain accepts a URL parameter called `jsonpCallback` (the name might vary in different domains), which expects the domain to return a script that is a call to `handleJsonp`. This script might look like Listing 3.

Listing 3. JSONP as it is returned from a target domain

```
handleJsonp( {
    'height':185,
    'units':'cm',
    'age': 30,
    'favoriteFruit':'apple',
    'likesDogs': true
});
```
Ask the middleman for help: Using the <iframe> element

An iframe is an element that lets you to embed pages within pages. If two pages come from the same domain, it is possible for them to communicate with and transfer information between each other.

In an Ajax application, it's common to use this paradigm to separate the roles of user interaction and client-server communication by using an iframe element that resides within the main page. This element would be hidden from the user, and would not interfere with any of the user interaction activities of the application.

The joint work lets you get any kind of content from the server, and lets you do form submissions without page refreshes—all behind the scenes.

What you can and cannot do

Unlike the previous mechanisms, which are all elements that construct a page, an iframe element is a page by itself. It is not limited to a specific kind of content (such as an image) or process (execution of the received contents). Therefore, any kind of data can be retrieved from, and sent to, any server. And because any content can be received, you can perform interactions based on data with any possible format, which gives you more flexibility on the server side. Sending data to the server can be based on form submission. You can also use POST requests in addition to GET. Multipart requests are possible, so files from the client's machine can be uploaded to the server. (And remember, all page refreshes block the rest of the interaction with the user.)

The use of iframe is not flawless. Because cross-site calls are valid, the security of the application might be vulnerable. Additionally, any interaction based on this mechanism is pushed into the page's history object, which may confuse the users that navigate with the Back/Forward actions.

Using iframe

Hiding the iframe

There are several ways to hide an iframe, and almost all would work, from placing it outside of the screen to reducing its size to zero. You can also set the visibility style to hidden.

However, you cannot set its display style to none; if you do so, the GET request cannot be created and the needed content cannot be fetched from the server.

Using the iframe element to fetch content from a server with a programmatic approach is similar to using the script element—with one important difference. After you create the script element and connect it to the page, you will need to hide it so that users won't be confused by its presence on their screen.

You can use the iframe element as a target for a form submission, and thus prevent the page refresh that results from a form submission.

Listing 4 shows how to upload files using an iframe element as part of the application's markup, instead of performing client-server communication in a programmatic fashion.
Listing 4. Uploading a file using an <iframe>

```html
<!--a hidden iframe that is the target of the form that would be used to upload a file -->
<iframe id="IFrame" name="IFrame"
    style="width:0px; height:0px; border:0px"
    src="blank.html"
</iframe>

<!--the form is connected to the previous iframe by the target attribute, thus basically
reloading that hidden frame upon form submission -->
<form name="UploadFile"  target="IFrame" method="POST"
    action="http://myServer/fileUploadServiceURL"
    enctype="multipart/form-data">
    <input type="file" name="uploadFileNameId"/>
    <input type="submit" value="Upload" name="submit"/>
</form>
```

Doing it the Ajax way

Ajax-based web applications usually act as a client to an application that runs on a server,
resulting in a communication-intensive application that sends data back and forth to that server.
A mechanism to provide data transport is needed. As a key component of the application, that
mechanism should be as lightweight and secure as possible. Luckily, all modern browsers are
equipped with an object you can use for exactly this purpose: XMLHttpRequest (XHR). XHR,
which is in fact lightweight, creates a request that is limited to the server the page was retrieved
from. It removes all cross-site scripting problems, and can convey only text.

What you can and cannot do

Use of the XHR object allows applications to send information to their server and fetch data from it.
This mechanism has several advantages over the previously described mechanisms:

Any type of HTTP method can be used

Because the most common modern server architectures are based on REST principles, you
need to use GET, PUT, POST, and DELETE requests. Using the XHR object at the core of the
client-server communication enables RESTful server architecture.

Notification upon completion

The XHR object can exist in several states, from creation to response fully loaded. Upon each
state change, an event is fired, and you can define a callback to be called upon future state
changes.

This event handler lets you ensure that code that relies on data fetched from the server is
executed when this data is available.

No intervention in the page history

XHR calls are not reflected in the page's history object, so it does not deviate from the
common usage of the Back and Forward actions of the browser.

Cross-Origin Resource Sharing

Cross-site Ajax calls are possible using the Cross-Origin Resource Sharing (CORS)
mechanism. However, CORS is still in its early days, is supported by a limited list of
browsers, and requires additional server-side coding.
No cross-site scripting (XSS) is allowed
The application is more secure because it lacks the ability to perform XSS.

Blocking or not
You can define whether the request-response cycle is either:
• Synchronous, where no code is executed while the browser waits for the response
• Asynchronous, where a callback can be executed when the response has arrived

XML or any other format
In cases where the response is in XML format, a complete DOM tree is created for that response data. The raw textual content of the response is also available (and can be handled when the response arrives in another format, such as JSON).

Still, not everything is sunshine and flowers with the XHR object. Because only text can be sent, you still need to use iframe to upload files to the server. In addition, the creation of this object in different browsers is done differently.

Summary
Client-server interaction is the backbone of any modern web application. In this article, you learned about the common mechanisms you can use for your application interaction. Not every application needs all of the mechanisms discussed here. It's up to you to decide which you need and how to use them.

Many JavaScript frameworks come with either partial or complete implementation of the set of Ajax communication mechanisms. Consider this when you decide which framework to use and which mechanisms to implement.
Related topics

- Learn more about JSON, the lightweight data-interchange format.
- This JSON Schema lets you to formalize the structure of a JSON object.
- Read the W3C XMLHttpRequest specification, which defines an API that provides scripted client function for transferring data between a client and a server.
- The W3C Cross-Origin Resource Sharing specification defines a mechanism to enable client-side cross-origin requests. Specifications that enable an API to make cross-origin requests for resources can use the algorithms defined by this specification.
- Cross-domain Ajax with Cross-Origin Resource Sharing is an excellent blog post about CORS.
- Learn more about Ajax on Wikipedia.
- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

© Copyright IBM Corporation 2010
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)