Build Ajax applications using the first real Ajax server: Aptana Jaxer

Learn to build Jaxer applications with HTML/DOM/JavaScript support on both the client and server side

Ken Ramirez

August 26, 2008

Get acquainted with Jaxer, the first true Asynchronous JavaScript + XML (Ajax) server. Jaxer makes it possible to execute JavaScript code, Document Object Model (DOM), and HTML on the server side as well as giving you the ability to access server-side functions asynchronously from the client side. This article describes the features of Jaxer and shows the great potential that Jaxer has to offer, even in its infancy.

Traditionally, to provide richer client features to Web-based clients, you had to create Web-based applications that are comprised of a homogeneous system from various technologies, which may include:

- Server-side Web or application server, such as Apache HTTP Server, Microsoft® Internet Information Services (IIS), Sun Java™ Web Server, IBM® WebSphere®, or BEA WebLogic
- Server-side scripting or processing language, such as Java, PHP, JavaServer Pages™ (JSP), or Active Server Pages (ASP)
- Client-side scripting and formatting, such as HTML, Cascading Style Sheets (CSS), JavaScript, or DOM
- HTTP communication protocol or application program interface (API), such as XMLHttpRequests or JavaScript Serialized Object Notation (JSON)

Now, however, you can use Jaxer, a new Ajax server that not only integrates all of these technologies into one deployable server, but also provides server-side scripting and processing using some of the same client-based technologies (such as JavaScript code, DOM, and more). Jaxer is free open-source code that you can use as is, or extend further using its JavaScript framework.

Imagine being able to use JavaScript code directly in your HTML pages and simply stating that the code should execute on the server side before returning the HTML on the client side. This would allow further communication with the server directly from the client without refreshing the page. Furthermore, the resulting HTML could be based on JavaScript code executed on the server side.

This would lessen the number of technologies and amount of code you have to write, providing a better overall experience for both the developer and the user.

For the user, you can now provide Ajax-based functions that are closer to a rich native application. Jaxer makes it possible because it is actually the first true Ajax server. You don't need to determine what browser your Ajax code is running in. You also don't have to write protocol code to perform the server communication. By calling simple Jaxer APIs, you provide robust Web applications with minimal fuss. Even more importantly, you no longer have to expose all of your code using embedded JavaScript. You can actually use Jaxer to unify client code with server code, hiding strategic code securely behind your firewall while allowing it to still be accessible from your client.

Installation

Before you can develop any Web-based applications that use Jaxer, you must first install it on your machine or in your development environment. There are three choices. Jaxer is available for Microsoft Windows®, Mac OS X, or Linux®. The Jaxer installation is a self-contained, stand-alone Apache/Jaxer server. However, you can also install it as a module in an existing Apache or Jetty environment. Aptana reports plans to support IIS in the near future.

For the purpose of this article, and because most developers write code on Windows and then deploy to a Windows or *NIX environment (such as UNIX® or Linux), I've chosen to install the stand-alone Windows version. Installing on Windows is very easy. You simply go to the Aptana Jaxer download page (see Resources for a link to this page) and download the compressed (.zip) file for the Windows stand-alone version. At the time of this article, the latest version was build 0.9.7.2472.

After you download and open the compressed file, copy the Aptana Jaxer folder to your hard drive. I copied mine directly to my C: drive. Thus, I can access my root folder for Jaxer by going to C:\Aptana Jaxer.

There are a number of files and folders inside the Aptana Jaxer folder. The root folder contains the following files:

- ConfigureFirewall.exe
- LICENSE.TXT
- README.TXT
- StartServers.bat

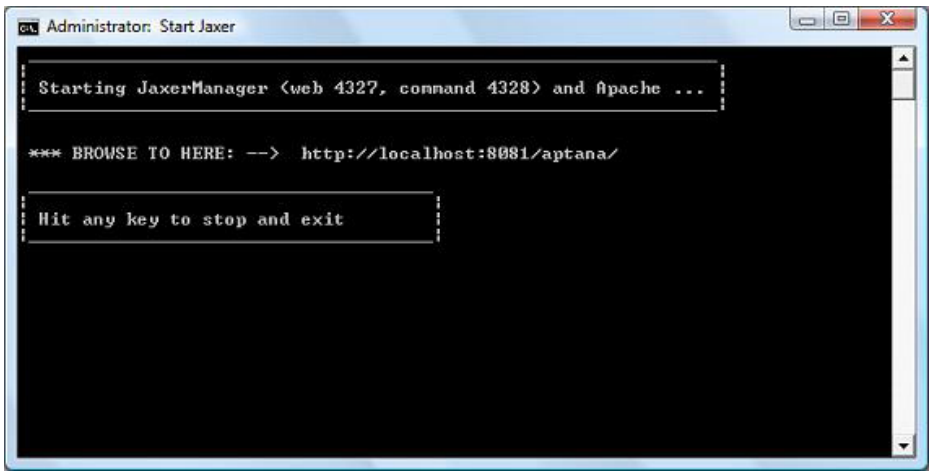
It also contains the following folders:

- Apache22
- data
- jaxer
- local_jaxer
- logs
- public
- tmp

Starting and testing your Jaxer server

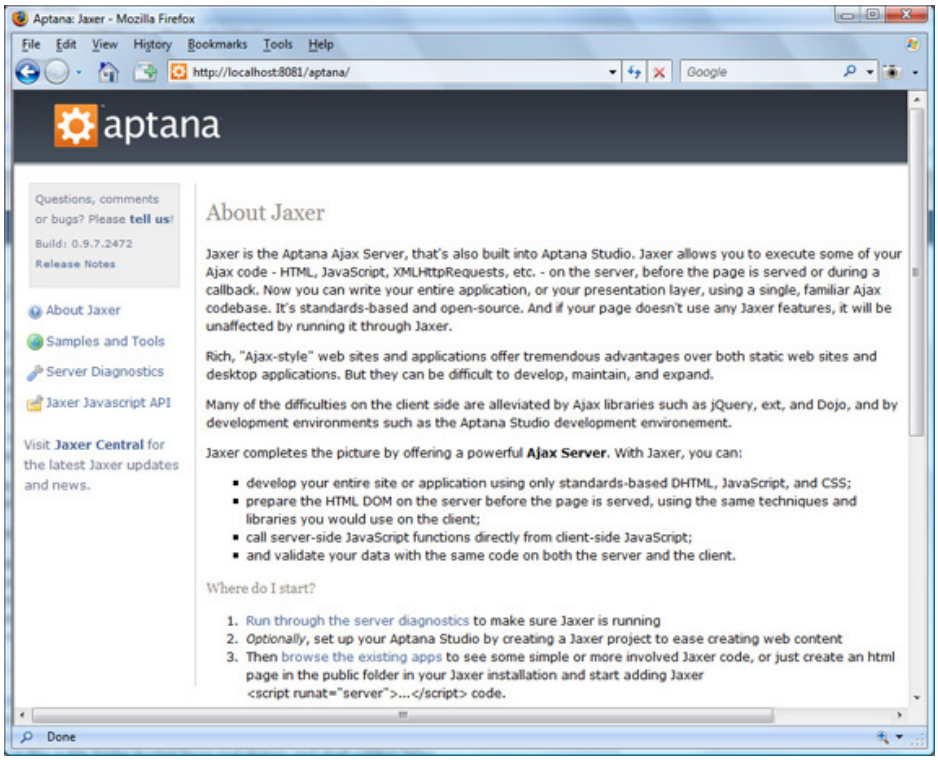
Starting the server is even easier than the original installation. Simply run the StartServers.bat file, and the startup window shown in Figure 1 is displayed.

Figure 1. Jaxer startup window



You must keep this window open as long as you intend to use and test applications in the Jaxer environment. You can test that your Jaxer server is running correctly by opening a browser window and navigating to http://localhost:8081/aptana. The About Jaxer page shown in Figure 2 should display in your browser.

Figure 2. About Jaxer page



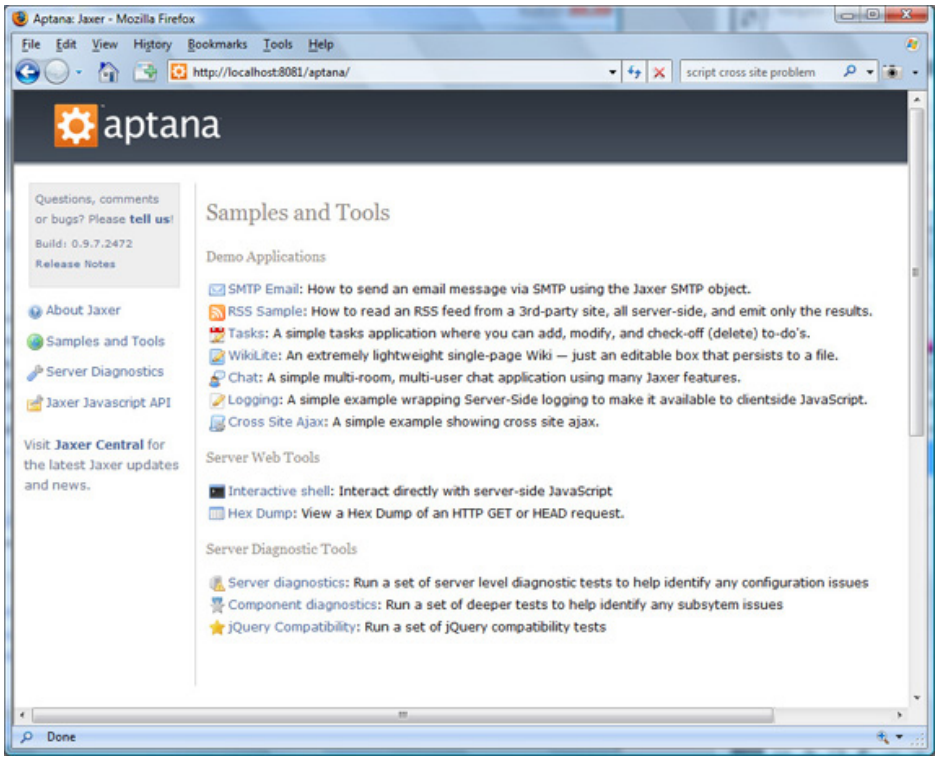
There are other Jaxer-based Web applications available in the Samples folder, which is located in <jaxer_install_dir>\jaxer\aptana\samples. You'll find several applications, including those described in Table 1.

Table 1. Applications included in the Jaxer Samples folder

Application	Description
chat	Demonstrates the sending and receiving of chat messages without having to refresh the entire page.
csajax	Uses Ajax to provide a solution to the problem normally faced by Web applications trying to communicate with other domains from the client browser using JavaScript code. Instead, the browser-based client uses Jaxer to communicate with the server and then allows the server to communicate with the other domain, essentially providing a solution to the cross-site problem. For more information about this problem, see Resources.
logging	Shows how to wrap server-side logging and access it with client-side JavaScript.
rss-sample	Demonstrates how the page can be divided to show a list of news stories in one portion of the page while showing a selected story in another portion of the page. The story is shown within its section, without refreshing the entire page; all done using Ajax and Jaxer.
smtp-email	Allows you to use the Jaxer Simple Mail Transfer Protocol (SMTP) object for sending e-mail asynchronously.
tasks	Allows you to add, modify, or delete items in a simple to-do or task list using asynchronous communication.
wikilite	Allows the user to provide text for a simple wiki in edit mode, save the text, and then view it later.

You can access these samples by clicking the **Samples and Tools** link shown in Figure 2. The resulting Samples and Tools page is shown in Figure 3.

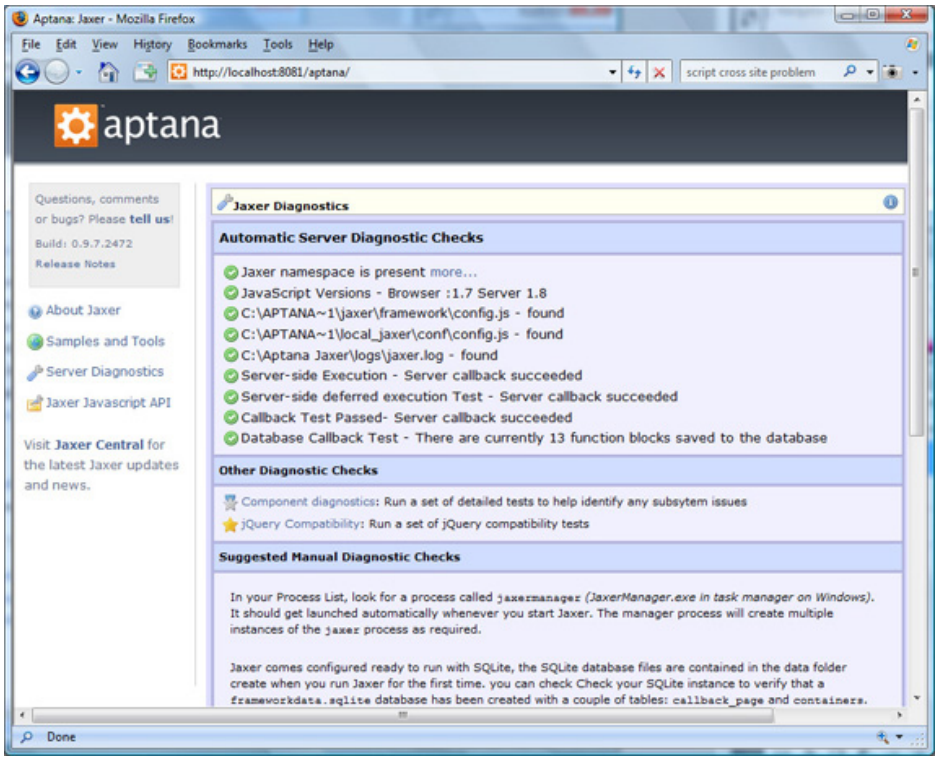
Figure 3. Samples and Tools page



You'll be using the tasks sample application to learn more about Aptana Jaxer later in this article.

As you might have noticed, there are also several Web and diagnostic tools you can use to interact with Jaxer and diagnose issues when they occur. For example, clicking the **Server Diagnostics** link returns the Jaxer Diagnostics page shown in Figure 4.

Figure 4. Jaxer Diagnostics page



Adding content

When I described the Jaxer root folder, I mentioned that there are several subfolders within it. One of these folders is named public. You can use this folder to provide your own content. If you do so, it will be called from the browser if you simply specify the host:port address in the browser's address line. For example, http://localhost:8081/ returns whatever index file you provide there.

Even better, you can create several projects within the public folder and load the project by name. For example, create a project named my_project. The complete path to this folder should be C:\<jaxer_root>\public\my_project.

Now create an index file named index.html and place it in the my_project folder. You can load the file using the Jaxer server by entering its path in the browser. In this case, you enter `http://localhost:8081/my_project`. You can place any HTML, images, JavaScript files, or CSS files in this folder or any of its subfolders.

In my version of my_project, I chose to create an index.html file that borrows from the quick start code available on the Aptana Web site. This code, which demonstrates server-side DOM, JavaScript code, and callback, is shown in Listing 1.

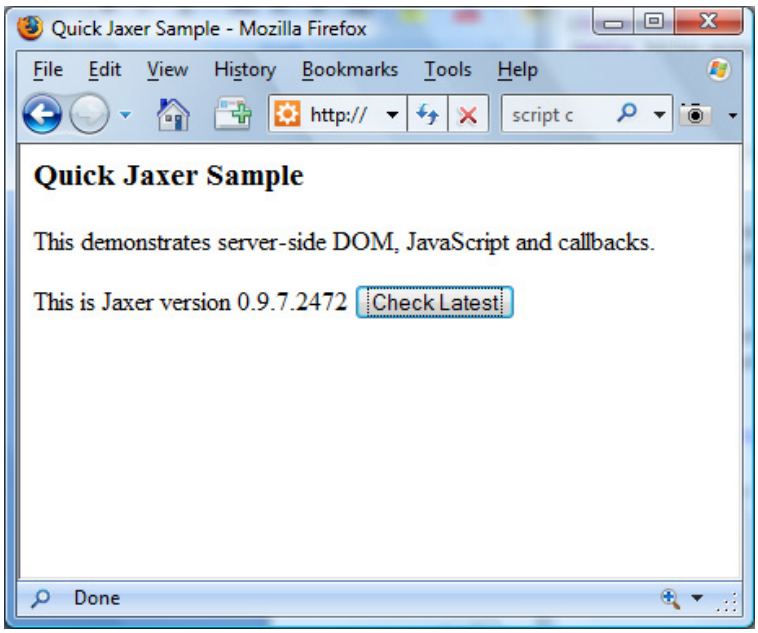
Listing 1. Sample code to test Jaxer

```
<html>
<head>
<title>Quick Jaxer Sample</title>
</head>
```

```
<body>

<h3>Quick Jaxer Sample</h3>
<p>This demonstrates server-side DOM, JavaScript and callbacks.</p>

<script runat="server-proxy">
  document.write("This is Jaxer version " + Jaxer.buildNumber);
  function getLatestVersion() {
    var url = "http://update.aptana.com/update/jaxer/win32/version.txt";

    try {
      var versionString = Jaxer.Web.get(url);
    }
    catch (e) {
      throw "Could not retrieve version number from " + url;
    }

    var matches = versionString.match(/\"([\.\d]+)\"/);
    return (matches && matches.length > 1) ? matches[1] : "(unknown)";
  }
</script>
<input type="button" value="Check Latest" onclick="alert('Latest version: '
  + getLatestVersion())">

</body>
</html>
```

When executed, the code produces the page shown in Figure 5.

Figure 5. Quick start code used in my_project



If you view the actual HTML source code sent to the browser, you find that it is quite different from the original code. Take a look at Listing 2.

Listing 2. Jaxer-generated HTML code returned to the browser

```
<html><head>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1">
<script src="\jaxer\framework\clientFramework_compressed.js?
   version=0.9.7.2472"></script>
```

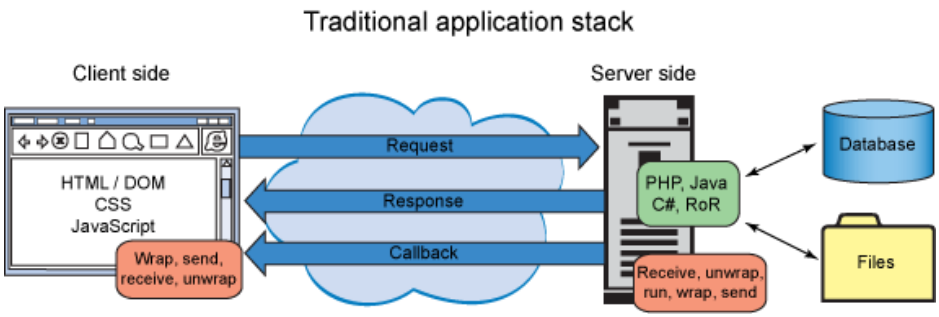
```
<script>Jaxer.Callback.pageSignature = -1837648436;
  Jaxer.Callback.pageName = 'localhost:8081/my_project';
  Jaxer.CALLBACK_URI = '/jaxer-server/callback';
  Jaxer.ALERT_CALLBACK_ERRORS = true;</script>

<title>Quick Jaxer Sample</title>
</head><body>

<h3>Quick Jaxer Sample</h3>
<p>This demonstrates server-side DOM, JavaScript and callbacks.</p>

<script>
function getLatestVersion() {return Jaxer.remote("getLatestVersion", arguments);}
function getLatestVersionAsync(callback) {return Jaxer.remote(
  "getLatestVersion", arguments, callback);}
</script>This is Jaxer version 0.9.7.2472

<input value="Check Latest" onclick=
  "alert('Latest version: ' + getLatestVersion())" type="button">

</body></html>
```

Notice that the original code didn't need to include any external JavaScript files. Next, take a look at the JavaScript section wrapped by the `<script>` block. The only difference between the JavaScript code you normally write for a browser and this is that the code uses the `runat` attribute to specify that the code should be executed using the `server-proxy`.

Jaxer high-level view

Before looking at the Jaxer architecture, first take a look at the architecture of a traditional application, as shown in Figure 6.

Figure 6. Traditional application stack

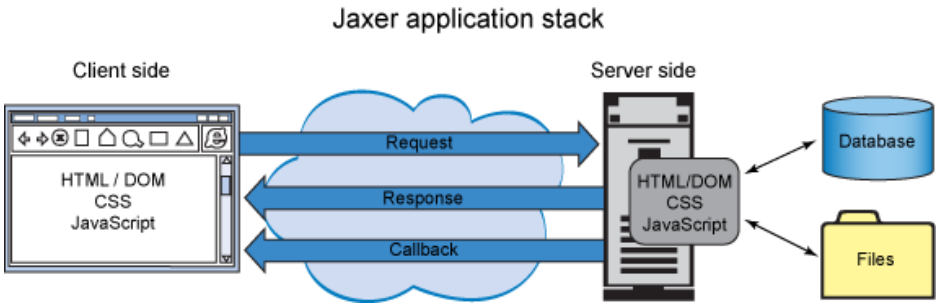


All stacks begin life with a request from the client to the server. In the traditional case, the server side responds by processing the request using one or more technologies, which might include PHP, Java code, C#, Ruby On Rails, or any number of other scripting/object technologies. The server side can perform database or file access, process the data, then format the data and return it back to the browser. These technologies are separate components that must be brought together by connecting code.

If the Web-based application is to use Ajax, it must take steps to process the client requests, wrap the call/parameters, and send the requests. On the server side, code must unwrap the request to determine the call/parameters, then execute the call, receive the response, rewrap the response, and return it to the client. The client then must unwrap the response and finally process it. The

wrapping/unwrapping can be done using JSON, and the actual request/response/callback can be done using XMLHttpRequest. In contrast, take a look at what a Jaxer stack looks like, as shown in Figure 7.

Figure 7. Jaxer application stack



Notice the stack is much simpler. Also notice that there is no need for external scripting; it's all handled within the actual HTML pages. The JavaScript code simply targets where it should run by specifying the location using the `runat` attribute.

When client code needs to call the server side, Jaxer seamlessly places JavaScript wrapper code into the HTML returned to the browser. This is the reason why Listing 2 was slightly different from the original code in Listing 1.

In case you're wondering, Jaxer's core is actually written in C++. It also integrates the Mozilla engine to provide parsing and APIs for HTML, CSS, and JavaScript code. From a programmer's perspective, what you see is pure JavaScript code provided as a framework that you use to accomplish server-side JavaScript code and client-side Ajax.

Jaxer provides services to support the development of applications that require the following technologies:

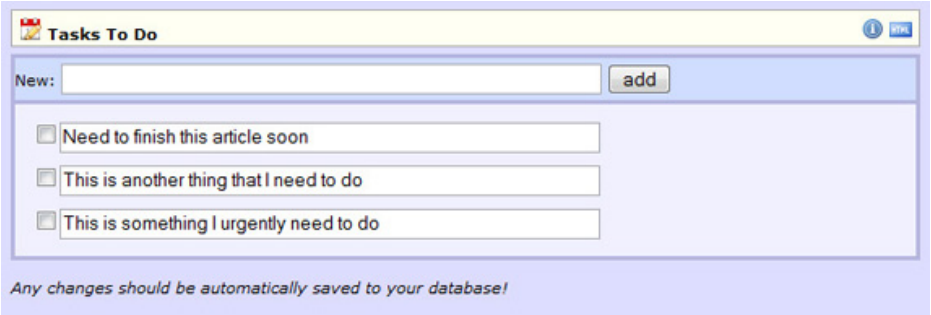
- Ajax
- Database communication from the client or server
- Client- or server-side logging
- Session manipulation
- Client- or server-side network API connectivity
- File objects
- the list goes on...

Additionally, you can integrate Jaxer with traditional technologies such as Java code to help solve cross-platform issues.

Example

To better understand how a Jaxer application is built, I'd like to walk you through one of the sample sites shipped with Jaxer. You'll be looking at the tasks application, which you can access by going to http://localhost:8081/aptana/samples/tasks. The page shown in Figure 8 is displayed. I've already added a few tasks.

Figure 8. Tasks sample main page



The tasks application is simple from a user's point of view. You simply type new tasks in the **New** box and click **add**. To remove items, simply click the checkboxes next to the items you want to delete.

Take a look at Listing 3, which shows some of the JavaScript code embedded in the tasks HTML page.

Listing 3. Tasks JavaScript code that runs on both the client and the server

```
<script type="text/javascript" runat="both">
  /*
   * Easy access to a named element in the DOM
   */
  function $(id)
  {
    return document.getElementById(id);
  }

  /*
   *
   */
  function addTaskToUI(description, id)
  {
    var newId = id || Math.ceil(1000000000 * Math.random());
    var div = document.createElement("div");
    div.id = "task_" + newId;
    div.className = "task";

    var checkbox = document.createElement("input");
    checkbox.setAttribute("type", "checkbox");
    checkbox.setAttribute("title", "done");
    checkbox.setAttribute("id", "checkbox_" + newId);
    Jaxer.setEvent(checkbox, "onclick", "completeTask(" + newId + ")");
    div.appendChild(checkbox);

    var input = document.createElement("input");
    input.setAttribute("type", "text");
    input.setAttribute("size", "60");
    input.setAttribute("title", "description");
    input.setAttribute("id", "input_" + newId);
    input.value = description;
    Jaxer.setEvent(input, "onchange",
      "saveTaskInDB(" + newId + ", this.value)");

    div.appendChild(input);

    $("tasks").insertBefore(div, $("tasks").firstChild);

    if (!Jaxer.isOnServer)
    {
```

```
      saveTaskInDB(newId, description);
    }
  }
</script>
```

First, notice that there are various JavaScript blocks. The first one has its `runat` attribute set to both, which means that the code within will be made available to both the client and server side. The functions in Listing 3 provide general functions accessible by other functions executed on both the client and server side. The `$` function simply returns an element based on the provided ID. The `addTaskToUI` provides the task on the client side using the `createElement` and `insertBefore` methods to create the user interface.

There aren't too many calls to any Jaxer-specific APIs here, only the definition of the events attached to the input and checkboxes, and the logical comparison against the `Jaxer.isOnServer` property, which is set to true if the JavaScript code is currently executing on the server side.

The `Jaxer.setEvent` is used to establish the communication proxies used to make calls from the client to the server. Stay with me, this will become clear in a few paragraphs.

The next segment of JavaScript code is slated to run on the server, based on the value passed to the `runat` attribute, as you can see in Listing 4.

Listing 4. Tasks server-side JavaScript code

```
<script type="text/javascript" runat="server">

  /*
   * The SQL to create the database table we'll use to store the tasks
   */
  var sql = "CREATE TABLE IF NOT EXISTS tasks " +
    "( id INTEGER NOT NULL" +
    ", description VARCHAR(255)" +
    ", created DATETIME NOT NULL" +
    ")";

  // Execute the sql statement against the default Jaxer database
  Jaxer.DB.execute(sql);

  /*
   * Set the 'onserverload' property to call our function
   * once the page has been fully
   * loaded server-side. We could have also set this attribute
   * on the <body> tag of our
   * page and had it call a function by name.
   */
  window.onserverload = function()
  {
    var resultSet = Jaxer.DB.execute("SELECT * FROM tasks ORDER BY created");
    for (var i=0; i<resultSet.rows.length; i++)
    {
      var task = resultSet.rows[i];
      addTaskToUI(task.description, task.id);
    }
  }

  /*
   * Save a task directly into the database
   */
  function saveTaskInDB(id, description)
```

```
{
  var resultSet = Jaxer.DB.execute(
    "SELECT * FROM tasks WHERE id = ?", [id]);
  if (resultSet.rows.length > 0) // task already exists
  {
    Jaxer.DB.execute("UPDATE tasks SET description = ? WHERE id = ?",
      [description, id]);
  }
  else // insert new task
  {
    Jaxer.DB.execute("INSERT INTO tasks (id, description, created) " +
      "VALUES (?, ?, ?)",
      [id, description, new Date()]);
  }
}
// Because we want this function callable from the client, we set its proxy
// value to true
saveTaskInDB.proxy = true;

/*
 * Delete a task from the database
 */
function deleteSavedTask(id)
{
  Jaxer.DB.execute("DELETE FROM tasks WHERE id = ?", [id]);
}
// Because we want this function callable from the client, we set its proxy
// value to true
deleteSavedTask.proxy = true;</script>
```

When you run this sample, take a look at the source code produced by Jaxer in your browser. Notice that the code from Listing 4 is missing. Conveniently, the defined functions have been converted to simple shells of the real functions. In their place, simple Jaxer remoting has been created.

Listing 4 provides some interesting JavaScript code. Although, it is all JavaScript code, notice the power that comes with having the ability to define and run JavaScript code on the server side. In the first portion of this segment, note that a Structured Query Language (SQL) table is created (if it doesn't exist). Next, an event similar to `onload` on the browser side is established, except this one is on the server side. The function defined and assigned to the `onserverload` is called after Jaxer loads the entire file and makes note of the all of the global server functions and variables.

In this case, the methods defined include `saveTaskInDB` and `deleteSavedTask`. To have the functions callable from the client side, a proxy must be defined. This is done by setting the functions' `proxy` property to true.

After Jaxer loads the file, the functionality assigned to `onserverload` is executed. In this case, the function executes a SQL call to retrieve the present tasks from the database. Next, it proceeds to iterate over all of the tasks and calls the `addTaskToUI` method. If you remember correctly, the method inserts checkboxes and input boxes into the HTML document. But wait! That would mean that the server is actually pushing those checkboxes and input boxes into the document in this case. When this code is executed on the server side, the only thing that doesn't execute within the `addTaskToUI` is the code wrapped within the logical check to see if `isOnServer` is true or false. That piece of code is only needed for the client side to perform a remote call to the server for adding the tasks to the database.

Listing 4 has Jaxer calls sprinkled throughout the functions, mostly to deal with the database interaction. As you can see, again, no external components or facades or Data Access Objects (DAOs) are needed. It uses simple calls to the Jaxer API to perform the SQL function. Keep in mind that there is nothing to stop you from wrapping these simple calls with more specific JavaScript components that appear to be more familiar with the actual details of your database, essentially providing a faÃ§ade over the SQL database.

The rest of the JavaScript code in the tasks sample simply deals with the events that occur as you interact with the form controls, as shown in Listing 5.

Listing 5. Tasks remaining JavaScript code, executed only on the client side.

```
<script type="text/javascript">

  /*
   * This client function sets a task as completed and
   * calls the server-side function
   * 'deleteSavedTask' to remove it from the database
   */
  function completeTask(taskId)
  {
    var div = $("task_" + taskId);
    div.parentNode.removeChild(div);
    deleteSavedTask(taskId);
  }

  /*
   * Create a new task and add it to the user interface
   */
  function newTask()
  {
    var description = $('txt_new').value;
    if (description != '')
    {
      addTaskToUI(description);
      $('txt_new').value = '';
    }
  }

  /*
   * Create a new task if the enter key was hit
   */
  function newKeyDown(evt)
  {
    if (evt.keyCode == 13)
    {
      newTask();
      return false;
    }
  }
  </script>
```

This code is plain old JavaScript client code, and it only executes on the client side. Notice, however, that these functions call the functions wrapped in the server-side JavaScript code. The HTML-generated code provides proxy methods for these functions, and these methods call the remote methods on the server side. The server side methods' actual function or code is never exposed to the client side.

Summary

My hope is that you are able to see the great potential that Jaxer has to offer, even at this early stage. Yes, it is in its infancy and probably has some maturing to do. Nonetheless, it shows a lot of promise. After all, just the fact that it can run JavaScript code on the server side, Ajax on the client side, and is built on many of the existing and matured technologies is a major plus. It is easily set up and can run alongside an existing Web server environment with little or no issues. I tried this on my Windows machine, which is currently running Apache along with PHP, MySQL, and so forth, without a hitch. It never brought down my other Apache instance and was able to communicate with the Jaxer instance on port 8081 with no problems. The only possible problem I can see is if you are currently using port 8081 for something else (such as Tomcat).