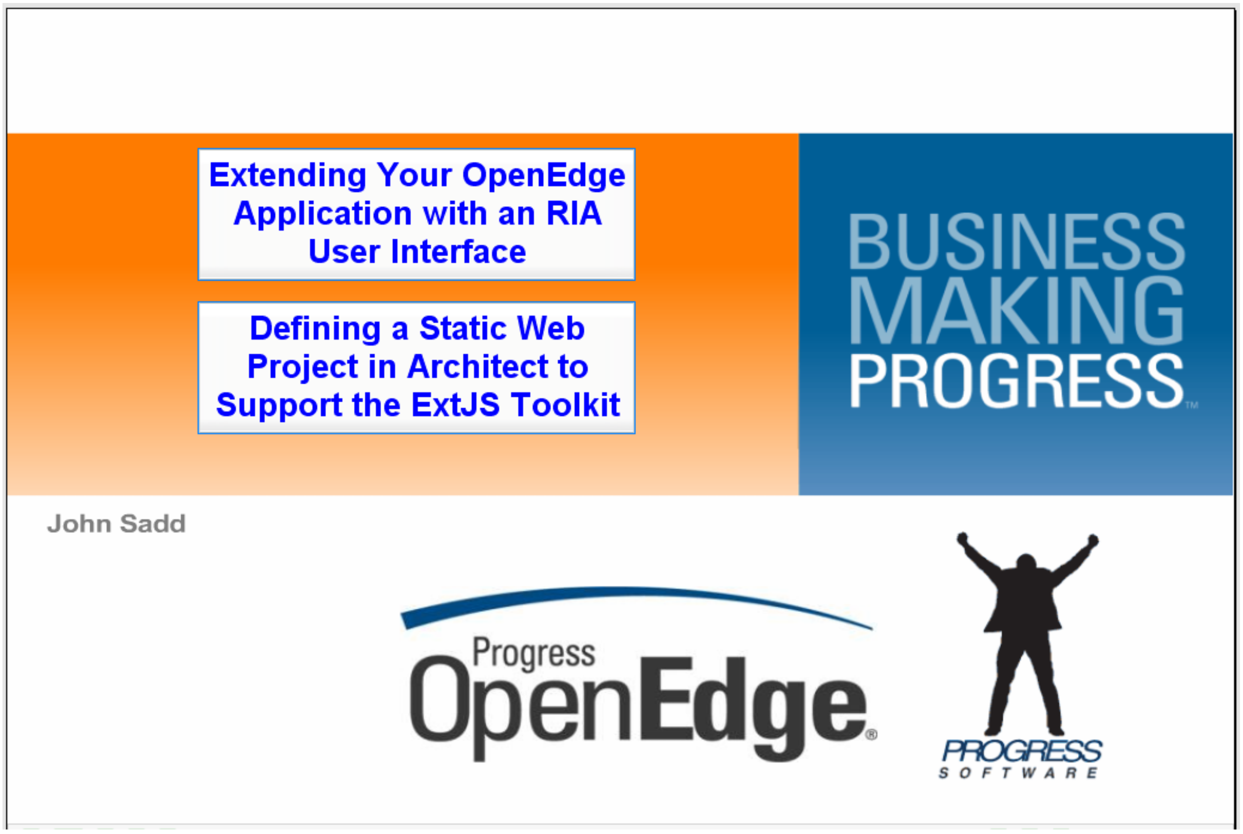
USING A STATIC WEB PROJECT FOR AN EXTJS SAMPLE

John Sadd
Fellow and OpenEdge Evangelist
Document Version 1.0
August 2010



Extending Your OpenEdge Application with an RIA User Interface

Defining a Static Web Project in Architect to Support the ExtJS Toolkit

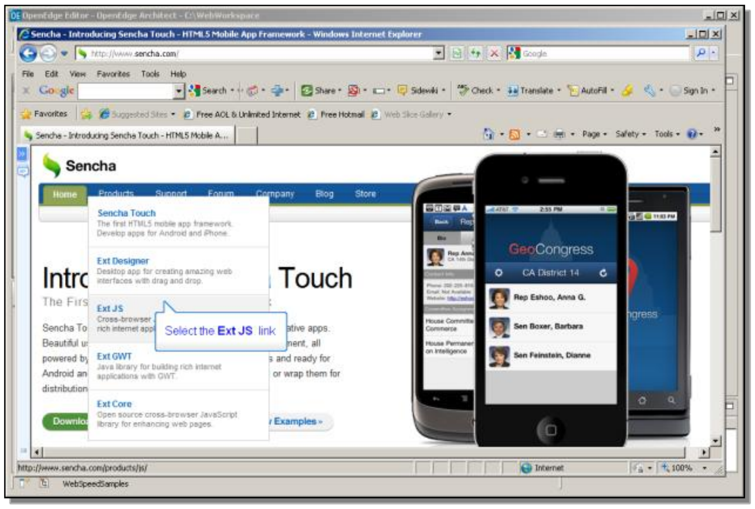
John Sadd

BUSINESS MAKING PROGRESS
Progress OpenEdge
PROGRESS SOFTWARE

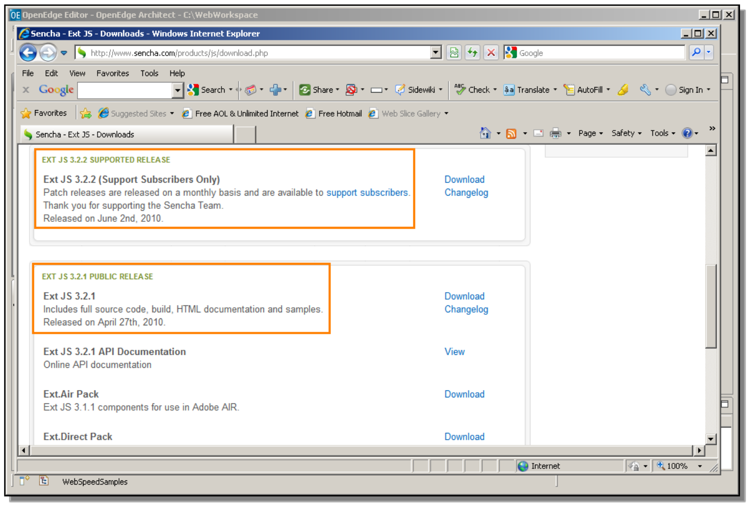
DISCLAIMER

This paper accompanies two videos in a series of presentations that set up a foundation for building a rich user interface on the desktop to communicate with an OpenEdge ABL application running on the server. The first of the two videos is entitled ***Defining a Static Web Project in Architect to Support the ExtJS Library***. It's the first of two parts that finally get you all the way to creating the beginnings of a user interface in a non-OpenEdge open source library. The library chosen for this example is **ExtJS**, one of many that you can choose to build a browser-based UI that can get data and execute business logic in an ABL application. In the first video and the first part of this paper I show you where to download the library, and then define a Web project in Architect to hold the content you build using that library.

The company that makes ExtJS also markets an HTML5-based framework for mobile devices, and they've changed their name accordingly, to Sencha, so you'll find their website at **`www.sencha.com`**.


Sencha
Home
Products
Support
Forum
Company
Blog
Store
Sencha Touch
The first HTML5 mobile app framework. Develop apps for Android and iPhone.
Ext Designer
Desktop app for creating amazing web interfaces with drag and drop.
Ext JS
Select the Ext JS link
Ext GWT
Java library for building rich internet applications with GWT.
Ext Core
Open source cross-browser JavaScript library for enhancing web pages.
Touch
GeoCongress
CA District 14
Rep Eshoo, Anna G.
Sen Boxer, Barbara
Sen Feinstein, Dianne
http://www.sencha.com/products/js/

The product I'm going to introduce you to is their extended JavaScript library for browser-based UI development, called ExtJS, which you can find in their product list. ExtJS is what we would term a lightweight AJAX library. Lightweight doesn't mean that it doesn't do much. In fact, ExtJS, as the name implies, is a set of JavaScript extensions that provide support libraries for a large number of very powerful user interface controls, designed to be very full-featured as well as browser-independent, so they can save you a tremendous amount of time coding a rich browser-based UI. We categorize it as lightweight because the empasis is on the UI controls, and not on providing a substantial framework for structuring your client-side application code. In any case, you can see that there is both a commercially supported version of ExtJS, and also a no-charge public release for it:


OpenEdge Editor - OpenEdge Architect - C:\WebWorkspace
Sencha - Ext JS - Downloads - Windows Internet Explorer
http://www.sencha.com/products/js/download.php
File Edit View Favorites Tools Help
Search Share Sidewiki Check Translate AutoFill Sign In
Favorites Suggested Sites Free AOL & Unlimited Internet Free Hotmail Web Slice Gallery
Sencha - Ext JS - Downloads
Page Safety Tools
Internet 100%
WebSpeedSamples

EXT JS 3.2.2 SUPPORTED RELEASE

Ext JS 3.2.2 (Support Subscribers Only)
Patch releases are released on a monthly basis and are available to support subscribers.
Thank you for supporting the Sencha Team.
Released on June 2nd, 2010.

Download
Changelog

EXT JS 3.2.1 PUBLIC RELEASE

Ext JS 3.2.1
Includes full source code, build, HTML documentation and samples.
Released on April 27th, 2010.

Download
Changelog

Ext JS 3.2.1 API Documentation
Online API documentation

View

Ext.Air Pack
Ext JS 3.1.1 components for use in Adobe AIR.

Download

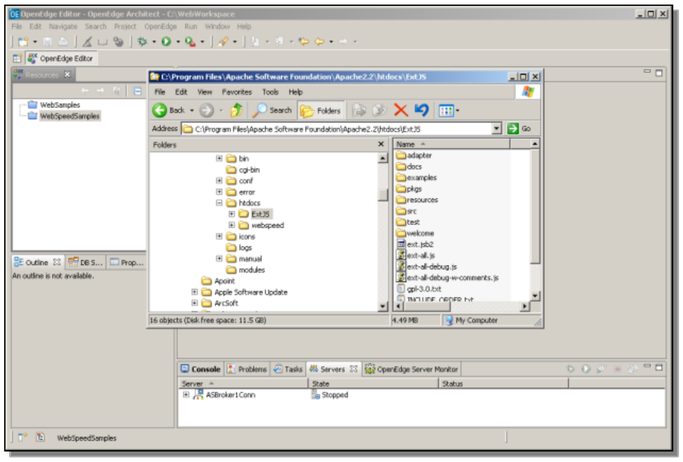
Ext.Direct Pack

Download

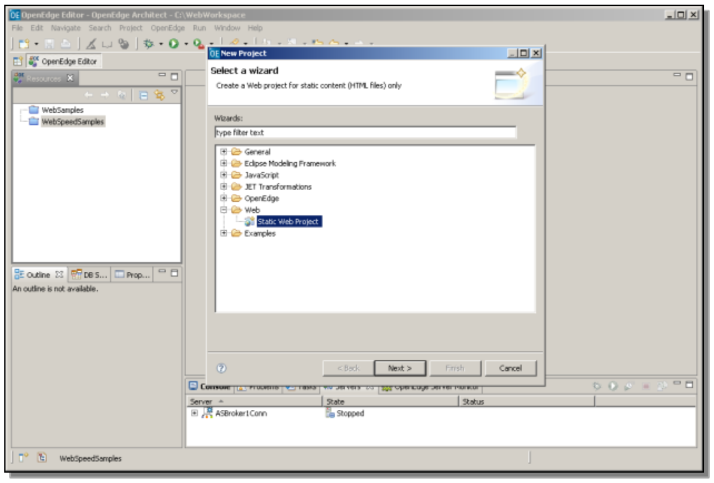
The fact that there's a supported release may be of interest to you as you make a decision among the many libraries that are available. The free public release means that you can try out the library without any initial investment. There's also an ExtJS user interface design tool available (for a fee) to help you get started building a UI and generating the supporting JavaScript code for it, which could also be a consideration. In any case, you can download the **Public Release** from this page.

Once that's done, you have a zip file with the name ExtJS-3.2.1, or whatever the latest release is at the time you do this. You need to unzip that file. Then, to make all the support code it contains available to your work, you can copy or move the top-level folder to the directory where your Web server looks for static application files. In the talks in this series on configuring WebSpeed, I installed the Apache HTTP server. In that case, the target directory for static content is the Apache **`htdocs`** directory. I also renamed the top-level installation folder to just **`ExtJS`** without the version number, so that I can easily add this as part of the pathname of any references I make to the library in my client-side application.

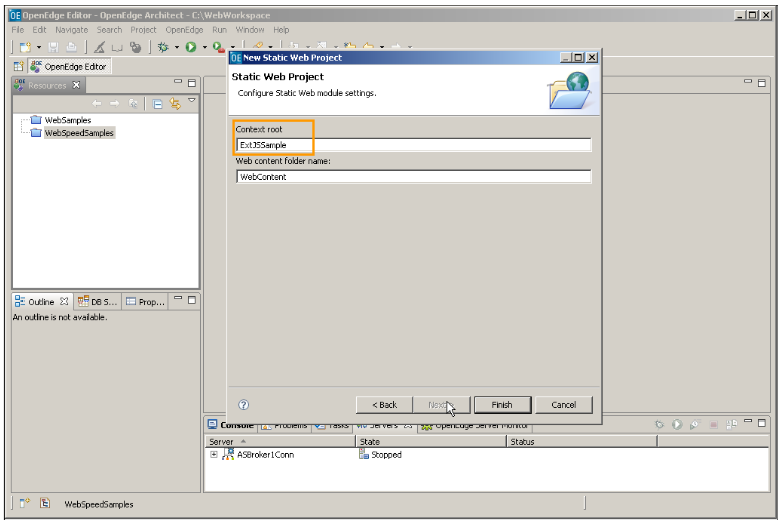
So below you can see the top level of all the support code that's installed for ExtJS. There's extensive online documentation and examples which you can refer to.


OpenEdge Editor - OpenEdge Architect - C:\WebWorkspace
C:\Program Files\Apache Software Foundation\Apache2.2\htdocs\ExtJS
Address C:\Program Files\Apache Software Foundation\Apache2.2\htdocs\ExtJS
WebSamples
WebSpeedSamples
adapter
docs
examples
pkgs
resources
src
test
welcome
ext.jsb2
ext-all.js
ext-all-debug.js
ext-all-debug-w-comments.js
gpl-3.0.txt
ASBroker1Conn
Stopped

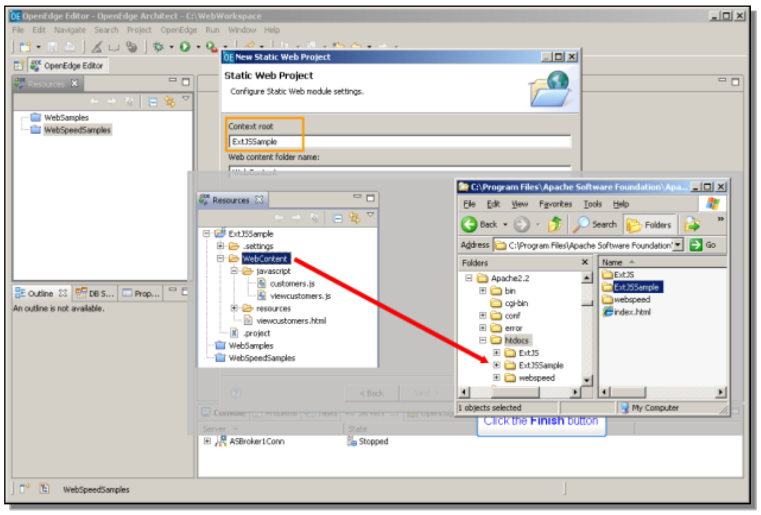
Now I have an RIA library installed to provide support for a new user interface. If you want to use a library such as this one, you need to create a special type of project in Architect to hold the code you build for that interface. Under **`File -> New -> Project`**, you need to create a non-OpenEdge project, since it's not going to contain any ABL code. It's called a **Static Web Project**. It will hold HTML and supporting JavaScript files with static content, that is, HTML that is not generated on the fly as much WebSpeed content is, hence the term static web project.



What's distinctive about this project type is that you can associate it with an HTTP server for it to use. For this example I just call the project **`ExtJSSample`**.

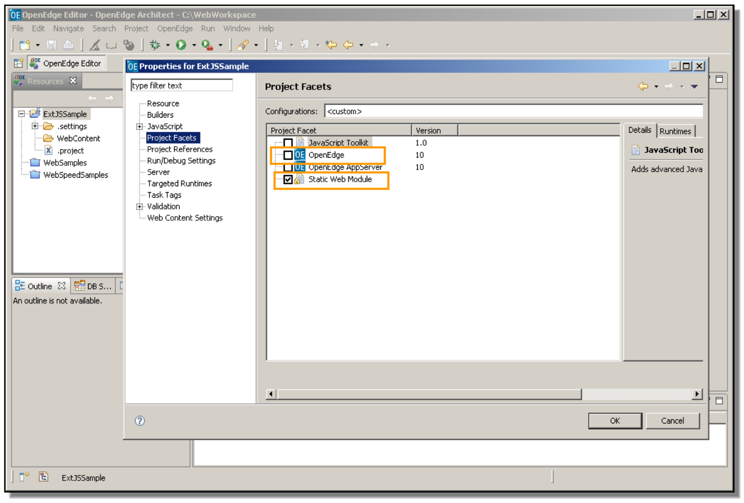


The **context root** on this page of the wizard is the target directory that content will be published to. If you watched the videos on Architect support for AppServer, it's very parallel to that. You develop content in Architect and then some portion of it gets copied – published – to a directory where the AppServer agents in that case, or the Web server in this case, will be able to find it to execute. This is illutsrated in this screenshot:



So **ExtJSSample** – the default name is just the name of the project I chose -- becomes a subdirectory under **htdocs**, just like the **ExtJS** product folder. The **Web content folder name** is the source that Architect publishes from – the folder in the project where you develop static web content. That's all you have to define for a new static web project.

Now if you take a look at the properties of the new project, you can select **Project facets**. A facet is an aspect of a project type, in effect a unit of functionality that defines an aspect of the project's behavior. Here you can see that the **Static web module** facet is checked on because this was defined this as a static web project:



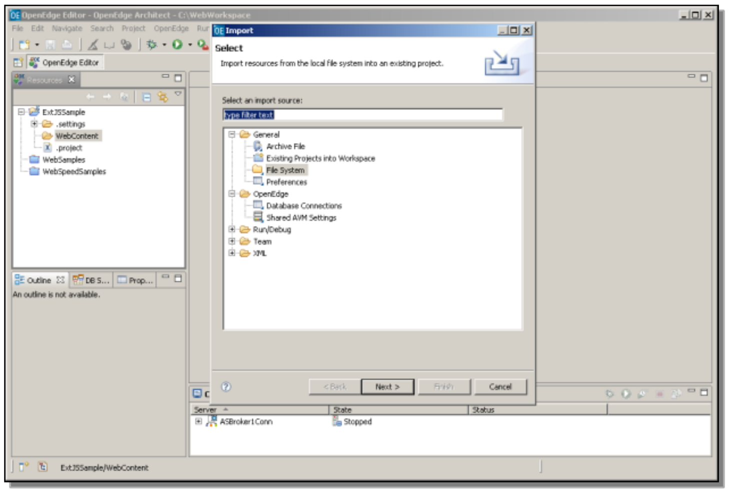
Project Facets

Configurations: <custom>

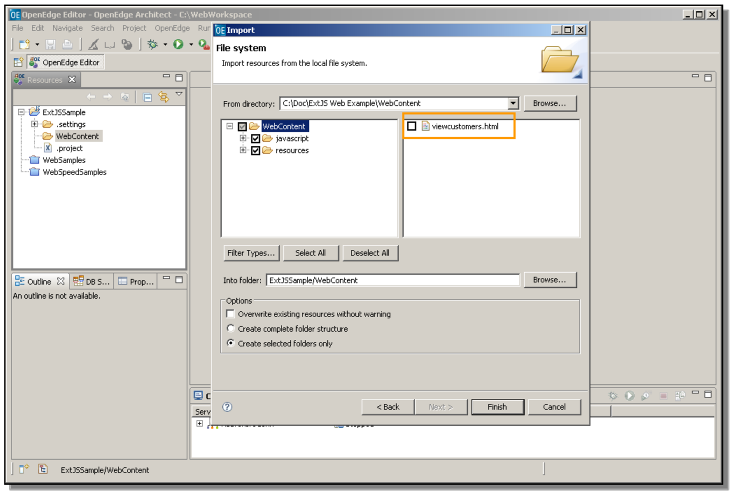
Project Facet	Version
☐ JavaScript Toolkit	1.0
☐ OpenEdge	10
☐ OpenEdge AppServer	10
☑ Static Web Module	

That facet defines this as a project with content that can be published as a static web module. Note that this isn't an OpenEdge project, so this isn't where you put ABL procedures and classes. The OpenEdge checkbox here is enabled, by the way, but it probably shouldn't be, and don't think about checking that on, because it could cause unwanted behavior.

Now we've got a Web content folder in the project to put code into that will be published to the Web server. For this example I'm going to import a simple ExtJS sample into the folder so you can get a small taste of what's involved in coding with ExtJS. So under the **File** menu I **Import** from the **File System**.



And then I specify a directory I set up where I have the sample code. Below you can see first that there's a **`javascript`** folder. ExtJS provides support for building advanced user interfaces through coding in JavaScript, and using a set of libraries that define the kinds of controls you want to put into your UI. This is where that code would go. Then there's a **`resources`** folder. This is where other supporting files such as css files – Cascading Style Sheets – would go. Then you define HTML files to load into the browser to make everything happen.



From directory: C:\Doc\ExtJS Web Example\WebContent

Into folder: ExtJSSample/WebContent

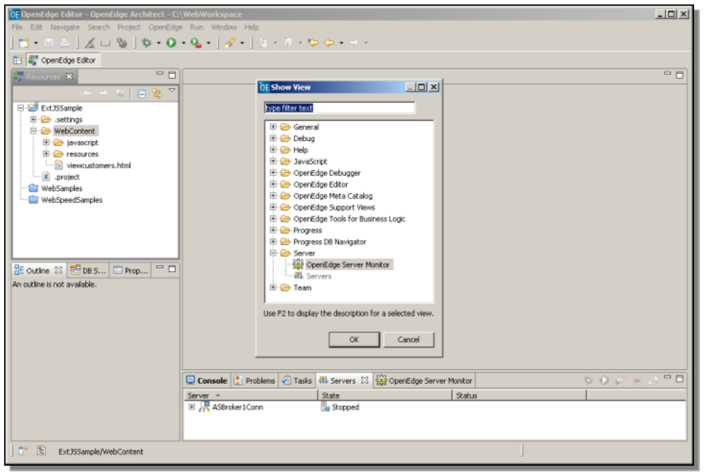
Options
- ☐ Overwrite existing resources without warning
- ○ Create complete folder structure
- ● Create selected folders only

`Viewcustomers.html` is a simple HTML file that wraps the extended JavaScript code to retrieve and display customers from an ABL procedure running on the server. We'll take a look at the code for all of this a little later.

The **Import** has brought the sample files into my project.

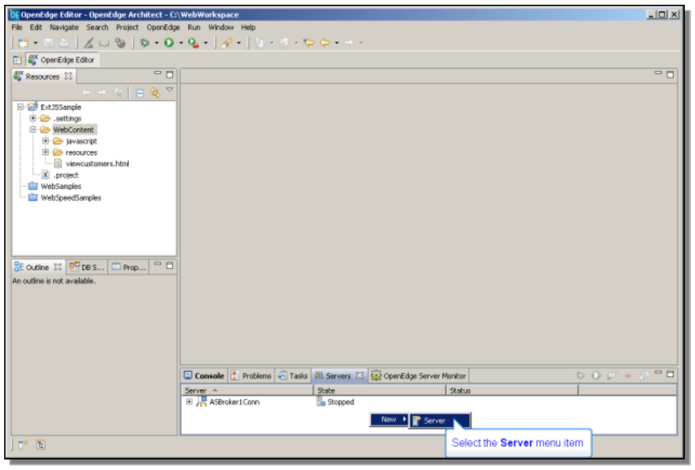
It's worth mentioning here that I didn't bring all of the ExtJS support code that I installed into the project as well. All I did was to move it by hand to the same **`htdocs`** folder where the Web server will find my project's **`WebContent`**. I won't be editing any of that installation support code, so there's no need to make it part of the project. That would only slow down the development process when Architect is doing a build or searching for content to publish.

The next thing to do is to define a new server to associate with the static web project. Now in my environment I happen to have the **Servers View** open already, because in another of these sessions I use it to define an AppServer connection, but if it isn't already open in your session, you can open it by selecting **Show View** from the **Window** menu, and then under **Other**, and then **Server**, select the **Servers View**:

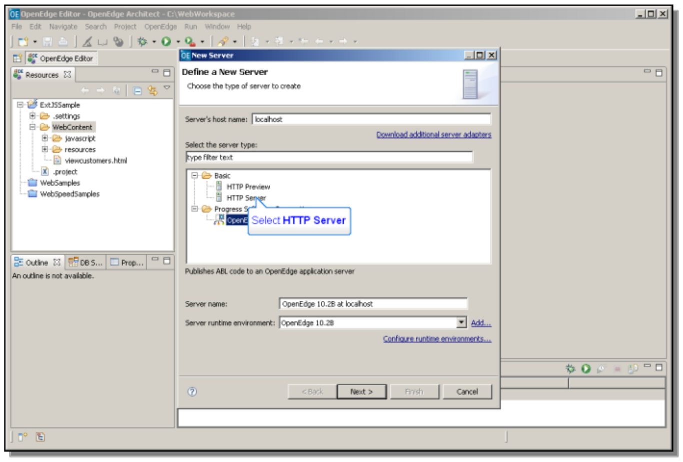


Server	State	Status
ASBroker1Conn	Stopped	

The screenshot below shows that view with the AppServer connection that I defined elsewhere. In the same way as I did for the AppServer, I can right-click in the View, and select **New ->Server**.



In this case I don't want an AppServer connection. Instead I expand the **Basic** node, and select **HTTP Server**. This is going to be a definition of my Apache HTTP Server for Architect to associate with the static web project.



Server's host name: localhost

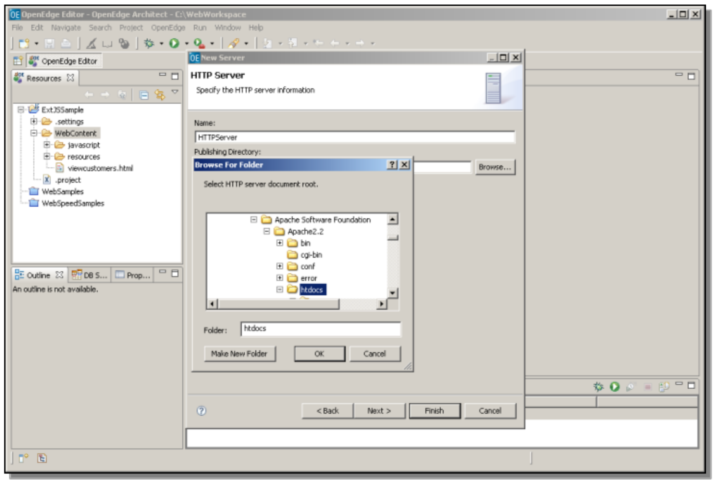
Select the server type: type filter text

Publishes ABL code to an OpenEdge application server

Server name: OpenEdge 10.2B at localhost

Server runtime environment: OpenEdge 10.2B

I just set the **Server name** to **`HTTPServer`**, and the way I associate it with my Apache server is to point it at the **`htdocs`** directory that server uses, which is here under the Apache install location:


HTTP Server

Specify the HTTP server information

Name: HTTPServer

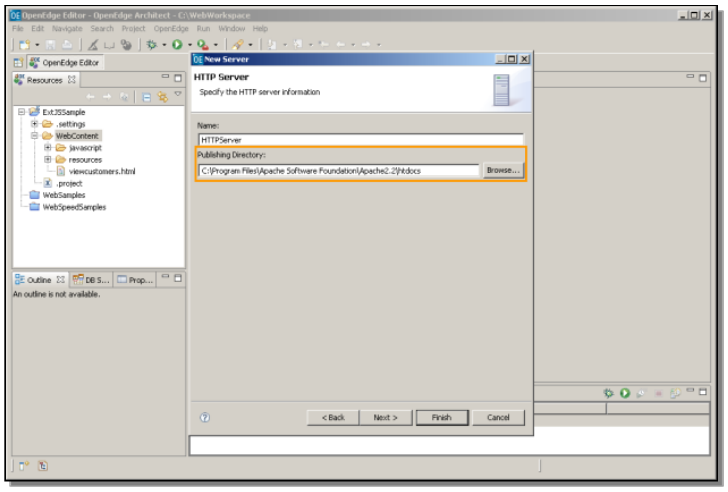
Publishing Directory:

Browse For Folder

Select HTTP server document root.

Folder: htdocs

Remember that this is the same location where I moved the ExtJS install, so it can be found by the server even though it's not part of the published content in the project. This next screenshot shows the full pathname to that publishing directory that you specify in the **HTTP Server** wizard:

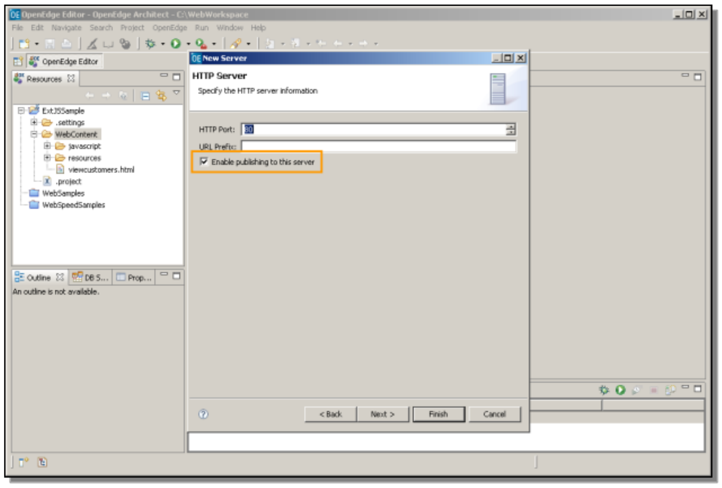

HTTP Server

Specify the HTTP server information

Name: HTTPServer

Publishing Directory: C:\Program Files\Apache Software Foundation\Apache2.2\htdocs

In the following wizard page, the **HTTP Port number** default of `80` is correct, and I don't need a **URL Prefix**. Note that I leave the option checked to enable publishing to this server. That will copy code from **`WebContent`** to the new folder under **`htdocs`** whenever code in the source folder changes:



HTTP Server

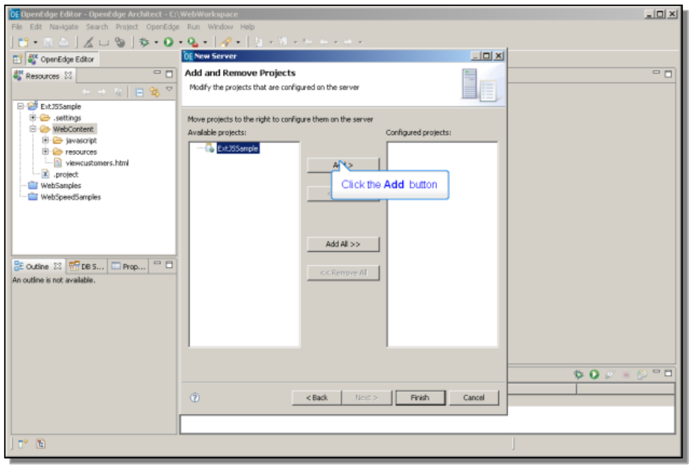
Specify the HTTP server information

HTTP Port: 80

URL Prefix:

☑ Enable publishing to this server

If you watched the AppServer videos, you've seen this next page of the new server wizard before. For an AppServer, it looks for AppServer-enabled projects, those with that facet checked on. In this case, an HTTP server looks for projects with the static web module facet checked on, so it finds my new sample project. I add that to the list of projects associated with this server, and the server definition is complete:



Add and Remove Projects

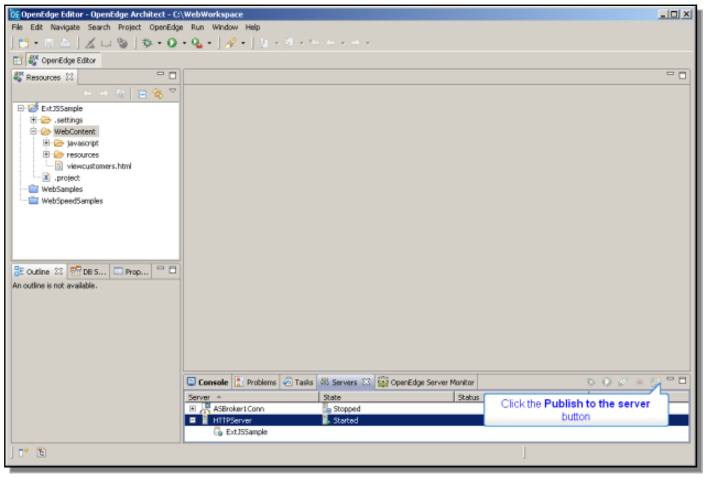
Modify the projects that are configured on the server

Move projects to the right to configure them on the server

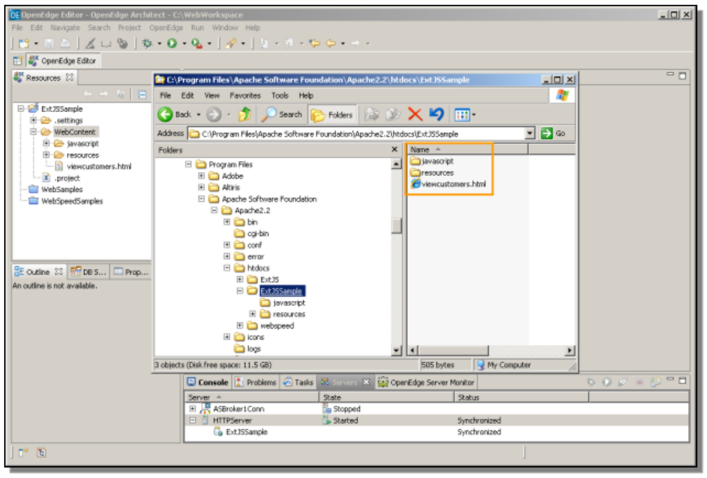
Available projects: ExtJSSample

Configured projects:

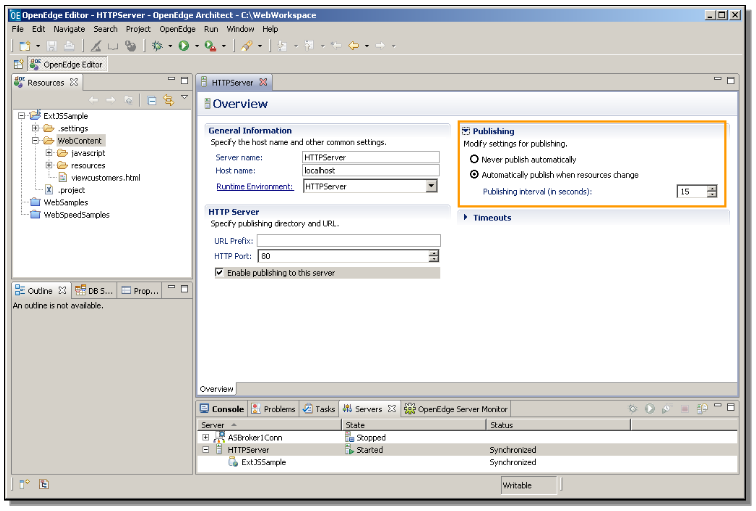
Below you can see the new server in the **Servers** View. I can expand it, and see the one project that I have associated with the server. The last button in the **Servers** View toolbar over on the right is the **Publish** button:



Pressing the **Publish** button immediately publishes anything it finds in the **`WebContent`** project folder. Now that the initial publish is done, I can go back to the **`htdocs`** folder in the Apache install, and see the **`ExtJSSample`** folder under it. This is what I named as the context root back in the project definition. And here you can see that the files in the project's **`WebContent`** folder have been copied here:



One more thing to point out is that if you double-click on a server in the **Servers** View, an **Overview** display comes up with a summary of all the settings that have been defined, including ones that got defaulted. Here you can adjust whether new and modified content is published automatically, for instance, and how frequently Architect should check for it:



General Information
Specify the host name and other common settings.

Server name: HTTPServer
Host name: localhost
Runtime Environment: HTTPServer

HTTP Server
Specify publishing directory and URL.

URL Prefix:
HTTP Port: 80
[✓] Enable publishing to this server

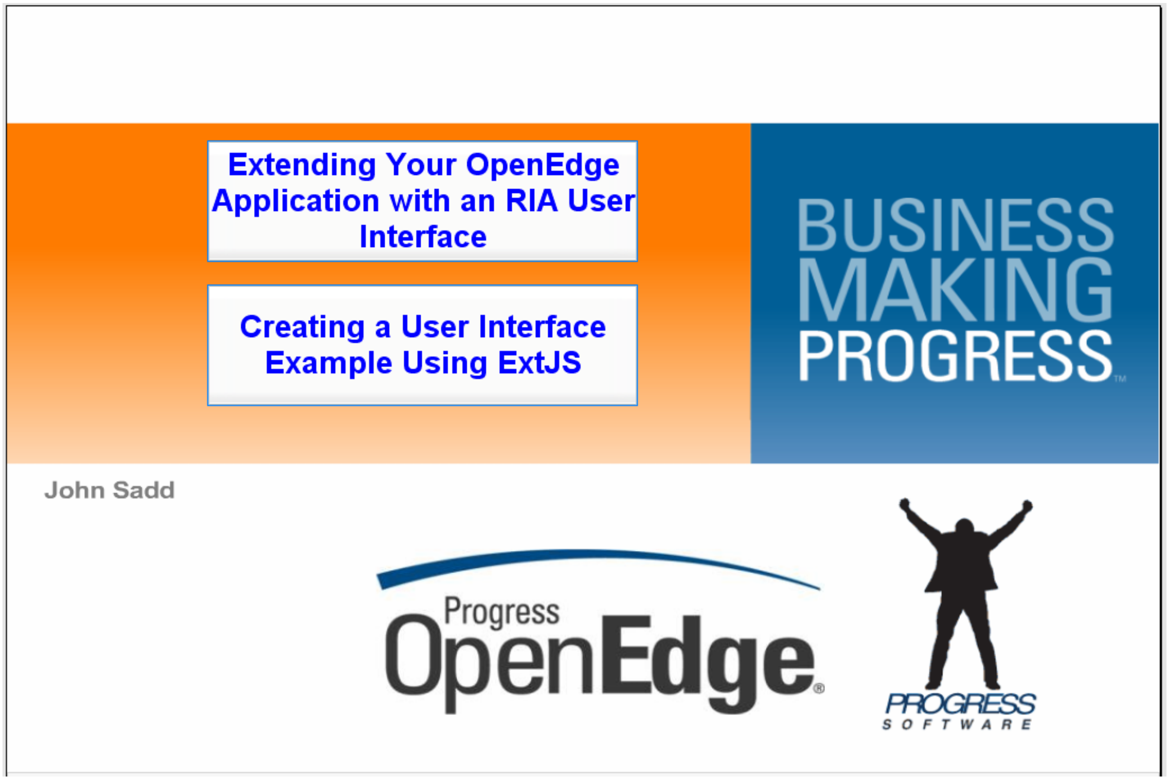
Publishing
Modify settings for publishing.

Never publish automatically [✓ Automatically publish when resources change]
Publishing interval (in seconds): 15

Timeouts

Server	State	Status
ASBroker1Conn	Stopped	
HTTPServer	Started	Synchronized
ExtJSSample		Synchronized

This brings us to the end of the content covered in the first part of this two-part video session on preparing to use ExtJS. The second video session walks you through a few of the highlights of the kind of code you need to write to use a library like ExtJS, and shows you the output from a simple example.



Extending Your OpenEdge Application with an RIA User Interface

Creating a User Interface Example Using ExtJS

John Sadd

BUSINESS
MAKING
PROGRESS
Progress OpenEdge
PROGRESS SOFTWARE

In this part of the presentation on setting up to use the ExtJS library, I examine the code for a very simple example and show you the output that results from running it.

The first code example shown below is **viewcustomers.html**. This HTML file is what actually gets loaded into the browser. In reality it serves as a wrapper around the extended JavaScript code that defines the UI, which could be embedded here, but is better to reference as a separate file. I'll say just a few things about this example.

```
<html>
<head>

<title>View Customers</title>
<link rel="stylesheet" type="text/css"
       href="/extjs/resources/css/ext-all.css">

<link rel="stylesheet" type="text/css" href="resources/stylesheets/myapp.css">
<script type="text/javascript" src="/extjs/adapter/ext/ext-base.js"></script>
<script type="text/javascript" src="/extjs/ext-all.js"></script>
<script type="text/javascript" src="javascript/viewcustomers.js"></script>

</head>

<body>
<div id="customergrid"></div>
</body>

</html>
```

The **title** is the title of the HTML page; that will show up as the label for the tab where output is displayed if you run it from Architect.

Next is a link to one of the install files, **ext-all.css**, where standard support for cascading style sheets is located. Notice that the pathname to this file starts with **extjs**. That's the installation folder that I renamed **ExtJS** and moved to **htdocs**, so all pathnames here are relative to that directory.

Then there's a reference to a specific stylesheet of my own, which in this case is just a placeholder.

The next two references to **ext-base.js** and **ext-all.js** are a standard part of ExtJS definitions, to JavaScript support code the library uses.

Following this there's a reference to my own JavaScript file, **viewcustomers.js**, which you've seen in the **WebContent** javascript directory.

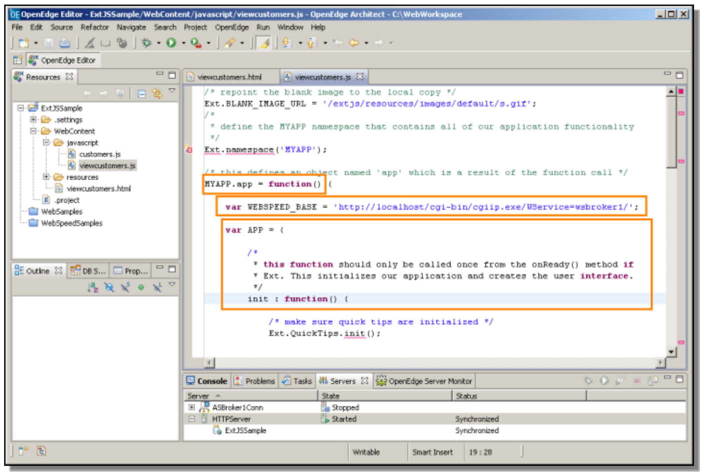
Finally, the **div** tag is the HTML definition of where the customer grid defined in the JavaScript gets placed in the overall UI. And that's it. The rest of the work is done in the JavaScript.

The next blocks of code show **viewcustomers.js**, where all the work of defining the controls and the data access is done.

I'll just point out a few of the elements of the JavaScript. There's an example of a reference to the extended libraries, named **Ext**. The **namespace** function lets me define a namespace for all the elements of my application.

```
/* repoint the blank image to the local copy */
Ext.BLANK_IMAGE_URL = '/extjs/resources/images/default/s.gif';
/*
 * define the MYAPP namespace that contains all of our application functionality
 */
Ext.namespace('MYAPP');
```

When I open the code in the Architect editor, an error icon appears to the left of the line that references **Ext.namespace**. Why is this here?



```
/* repoint the blank image to the local copy */
Ext.BLANK_IMAGE_URL = '/extjs/resources/images/default/s.gif';
/*
 * define the MYAPP namespace that contains all of our application functionality
 */
Ext.namespace('MYAPP');

/* this defines an object named 'app' which is a result of the function call */
MYAPP.app = function() {

    var WEBSPEED_BASE = 'http://localhost/cgi-bin/cgiip.exe/WService=wsbroker1/';

    var APP = {

        /*
         * this function should only be called once from the onReady() method if
         * Ext. This initializes our application and creates the user interface.
         */
        init : function() {

            /* make sure quick tips are initialized */
            Ext.QuickTips.init();
```

```
Server          State      Status
ASBroker1Conn   Stopped
HTTPServer      Started    Synchronized
  ExtJSSample              Synchronized
```

Well, one of the strange things about programming in JavaScript that takes some getting used to for ABL developers, or developers in most other languages for that matter, is that a great deal of the definition of a JavaScript class is done on the fly, dynamically, in executable statements encountered as the file is evaluated. Here in the code segment shown above, for instance, is the beginning of the definition of a function in the **MYAPP** space called **app**, which defines two variables. The first defines the URL to the webspeed broker, the same as I defined in Architect when I configured **wsbroker1**. That's how the JavaScript will communicate with the WebSpeed broker that executes ABL procedures. The second variable **APP** has a function called **init**, which actually defines the grid control. This definition all gets executed dynamically, so the editor is really not able to help you determine whether your JavaScript is valid or not. That's just one of the realities about programming in JavaScript.

```
/* this defines an object named 'app' which is a result of the function call */
MYAPP.app = function() {

        var WEBSPEED_BASE = 'http://localhost/cgi-
bin/cgiip.exe/WService=wsbroker1/';

        var APP = {

                /*
                 * this function should only be called once from the onReady()
method if
                 * Ext. This initializes our application and creates the user
interface.
                 */
                init : function() {

                        /* make sure quick tips are initialized */
                        Ext.QuickTips.init();
```

Further down, you can see the name of an ABL procedure that I'm going to invoke on the webspeed broker's URL, so I append the procedure name to the URL.

```
                        var customerURL = WEBSPEED_BASE + 'getcustomers.p';
```

Then there's an **HTTPproxy** definition, which will establish the connection to WebSpeed, using its URL together with the name of the procedure to run.

```
                        /* define a proxy to our customers url */
                        var customerProxy = new Ext.data.HttpProxy( {
                                url : customerURL
                        });
```

Then there's a variable that represents a **JsonStore**, a place to hold data in the client in the JSON format – JavaScript Object Notation. OpenEdge supports generating this format directly from a temp-table or ProDataSet, as you'll see when we look at **getcustomers.p**. The **root** corresponds to what is identified in the ABL as the **serialize-name**, basically a name for the top-level node in the data tree. And the **fields** map to fieldnames in the temp-table I'll populate with customers.

```
                        /* define a local store for holding onto our customers */
                        var customerStore = new Ext.data.JsonStore( {
                                proxy : customerProxy,
                                root : 'customers',
                                fields : [ 'Name', 'CustNum', 'Address', 'City',
'State' ]
                        });
```

Finally we get to the one UI control in the application. It's an ExtJS **GridPanel**, a grid control. The **renderTo** property is set to **customerGrid**. That name is the id in the **div** tag we looked at back in the **html** file. So this definition will create an instance of a grid control that will get displayed where that simple HTML definition says to display it. The **store** property is the **customerStore** defined just above.

```
                        var customerGrid = new Ext.grid.GridPanel( {
                                renderTo : 'customergrid',
                                height : 300,
                                width : 500,
                                store : customerStore,
```

Then there's a series of **column** definitions for the grid. Each **id** maps to one of the **field** names in the **store**, which in turn matches a field in the temp-table over in the ABL we'll look at in a moment.

```
                                columns : [ {
                                        id : 'Name',
                                        header : 'Customer Name'
                                }, {
                                        id : 'CustNum',
                                        header : "Customer Number"
                                }, {
                                        id : 'Address',
                                        header : "Address"
                                }, {
                                        id : 'City',
                                        header : 'City'
                                }, {
                                        id : 'State',
                                        header : 'State'
                                } ],
```

Then there are a couple of specialized property definitions, such as making the grid single selection.

```
                                sm : new Ext.grid.RowSelectionModel( {
                                        singleSelect : true
                                }),
                                viewConfig : {
                                        forceFit : true
                                }
                        });
```

Finally here, the **load** function tells the store to load its data, which it does by invoking the procedure at the end of the URL.

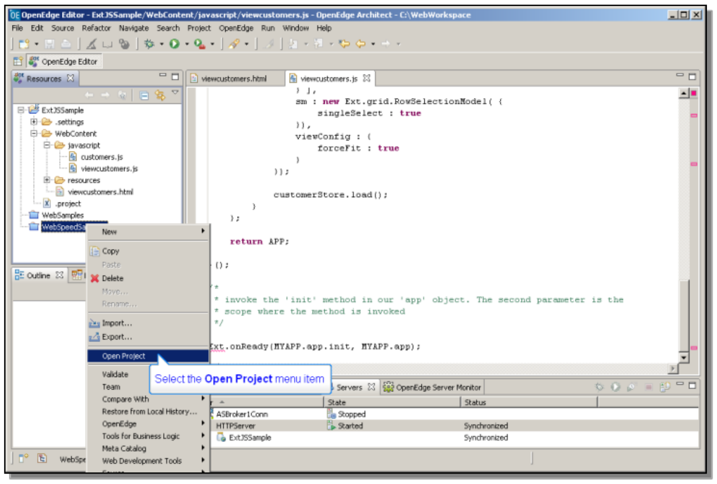
```
                        customerStore.load();
                }
        };

        return APP;

}();
```

That's the end of the **init** function in **app**. **Ext.onReady** gets called to create the UI, and it invokes this **init** function to get everything rolling.

```
/*
 * invoke the 'init' method in our 'app' object. The second parameter is the
 * scope where the method is invoked
 */

Ext.onReady(MYAPP.app.init, MYAPP.app);
```

So that's a simple but complete ExtJS UI definition.

Now I've been referring to WebSpeed here and the ABL procedures that get run in that context, so it's time to take a look at that. I open the **WebSpeedSamples** project that I created in another of these presentations.



```
                        } ],
                        sm : new Ext.grid.RowSelectionModel( {
                                singleSelect : true
                        }),
                        viewConfig : {
                                forceFit : true
                        }
                });

                customerStore.load();
        }
    };

    return APP;

}();

/*
 * invoke the 'init' method in our 'app' object. The second parameter is the
 * scope where the method is invoked
 */

Ext.onReady(MYAPP.app.init, MYAPP.app);
```

	State	Status
ASBroker1Conn	Stopped	
HTTPServer	Started	Synchronized
ExtJSSample		Synchronized

My static web project gives me a place to develop HTML and JavaScript code and publish that code to the Web server. This WebSpeed project represents the other half of the job. This is where I define ABL code that gets executed in the WebSpeed broker. The **wsbroker1** URL that I defined in the JavaScript example connects the two together at runtime, but at development time they give me two different Architect environments to work in, with different characteristics.

Here's the ABL procedure that the JavaScript will invoke in WebSpeed:

```
{src/web2/wrap-cgi.i}

output-content-type ("text/json").

define temp-table ttCust like Sports2000.Customer.

temp-table ttcust:serialize-name = "customers".

for each Sports2000.Customer where Customer.CustNum < 10:
    create ttcust.
    buffer-copy Customer to ttcust.
end.

temp-table ttCust:write-json("stream", "webstream", true).
```

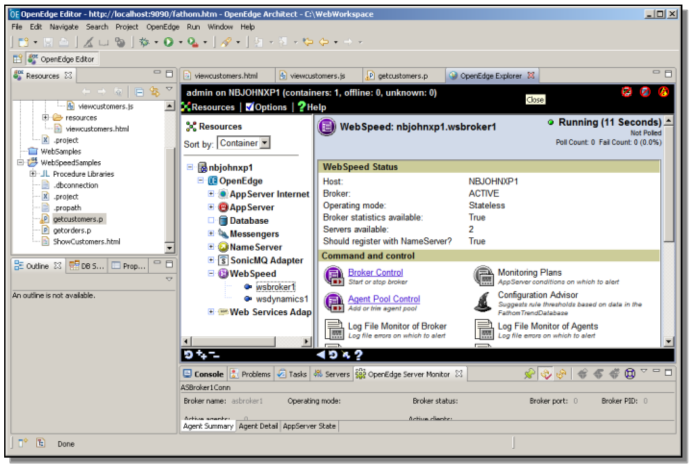
It uses the standard CGI wrapper include file that's part of WebSpeed.

It defines the output content type as being Json.

Then it defines and populates a temp-table with a few customer records. Remember the **root** property of the **customerStore** back in the JavaScript? That corresponds to the **serialize-name** here in the ABL, a header for the data in the temp-table as it gets converted to JSON.

Finally, the **write-json** built-in ABL function executes on the temp-table, and in a single statement, converts the data in the temp-table to JSON format and writes it to the webstream where it is returned to the **customerStore** defined in **viewcustomers.js**. So once you get the support pieces set up, the process is really very straightforward and well supported by the features of OpenEdge 10.

In my testing environment I also have the **Server Monitor** View open from my work with AppServer, and in that view, which you could also open from the **Open View** option on the **Window** menu, there's a button to provide access to OpenEdge Explorer from within Architect. You can drill down in Explorer to the **WebSpeed** node, select the default WebSpeed broker, **wsbroker1**, then select **Broker Control**, and **Start WebSpeed**.



WebSpeed: nbjohnxp1.wsbroker1

Running (11 Seconds)

Not Polled

Poll Count: 0 Fail Count: 0 (0.0%)

WebSpeed Status

Host:	NBJOHNXP1
Broker:	ACTIVE
Operating mode:	Stateless
Broker statistics available:	True
Servers available:	2
Should register with NameServer?	True

Command and control

Broker Control — *Start or stop broker*

Monitoring Plans — *AppServer conditions on which to alert*

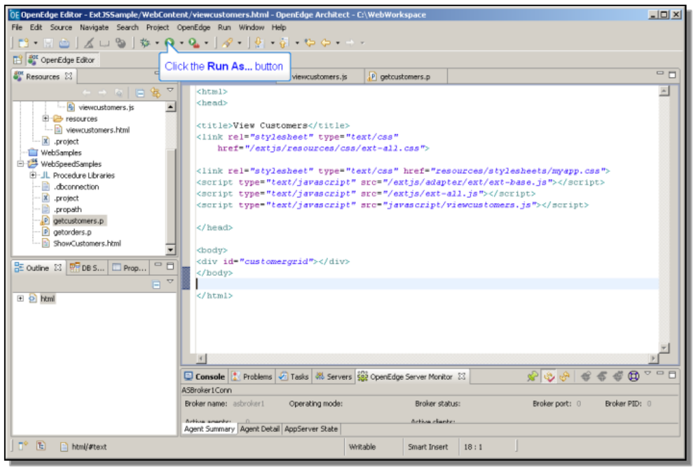
Agent Pool Control — *Add or trim agent pool*

Configuration Advisor — *Suggests rule thresholds based on data in the FathomTrendDatabase*

Log File Monitor of Broker — *Log file errors on which to alert*

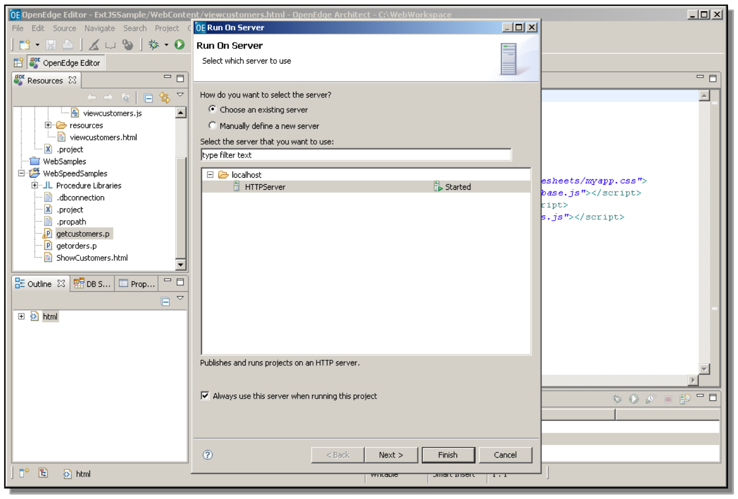
Log File Monitor of Agents — *Log file errors on which to alert*

Once I confirm that the broker is now active, I can go back to my static web project, select the **`viewcustomers.html`** file that represents the top of my client application, and run it:



```html
<html>
<head>

<title>View Customers</title>
<link rel="stylesheet" type="text/css"
    href="/extjs/resources/css/ext-all.css">

<link rel="stylesheet" type="text/css" href="resources/stylesheets/myapp.css">
<script type="text/javascript" src="/extjs/adapter/ext/ext-base.js"></script>
<script type="text/javascript" src="/extjs/ext-all.js"></script>
<script type="text/javascript" src="javascript/viewcustomers.js"></script>

</head>

<body>
<div id="customergrid"></div>
</body>

</html>
```

I get prompted whether I want to run it as an **OpenEdge Application** or on a **Server**. Of course I want to run it on my **HTTP Server**. My one HTTP Server is displayed, and that's the one I always want to use for files in this project. That's as much as I need to specify:


OpenEdge Editor - ExtJSSample/WebContent/viewcustomers.html - OpenEdge Architect - C:\WebWorkspace
Run On Server
< Back
Next >
Finish
Cancel

Run On Server

Select which server to use

How do you want to select the server?

(●) Choose an existing server

( ) Manually define a new server

Select the server that you want to use:

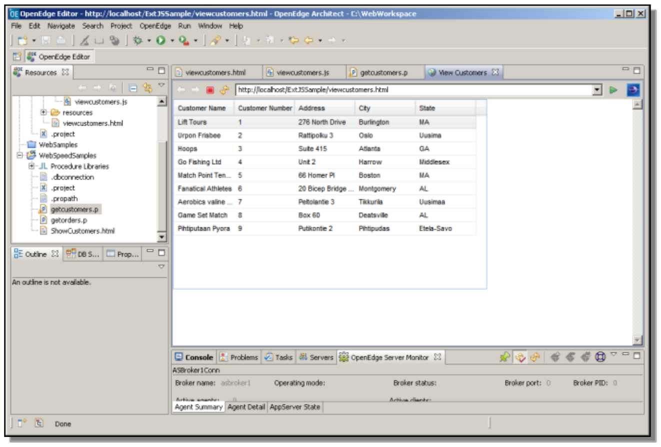
type filter text

localhost
HTTPServer Started

Publishes and runs projects on an HTTP server.

[✓] Always use this server when running this project

Now **viewcustomers.html** is loaded, invokes **viewcustomers.js**, which loads its **customerStore** by running **getcustomers.p** in the webspeed broker, and voila. Below you can see the HTML page displayed right here in Architect, with the Extended JavaScript **gridPanel** control displayed in the **customergrid** slot in the HTML:


OpenEdge Editor - http://localhost/ExtJSSample/viewcustomers.html - OpenEdge Architect - C:\WebWorkspace
viewcustomers.html
viewcustomers.js
getcustomers.p
View Customers

http://localhost/ExtJSSample/viewcustomers.html

Customer Name	Customer Number	Address	City	State
Lift Tours	1	276 North Drive	Burlington	MA
Urpon Frisbee	2	Rattipolku 3	Oslo	Uusima
Hoops	3	Suite 415	Atlanta	GA
Go Fishing Ltd	4	Unit 2	Harrow	Middlesex
Match Point Ten...	5	66 Homer Pl	Boston	MA
Fanatical Athletes	6	20 Bicep Bridge ...	Montgomery	AL
Aerobics valine ...	7	Peltolantie 3	Tikkurila	Uusimaa
Game Set Match	8	Box 60	Deatsville	AL
Pihtiputaan Pyora	9	Putikontie 2	Pihtipudas	Etela-Savo

And that's all there is to it.

This two-part session covers a lot of territory, and certainly there's a lot to learn with any of the RIA libraries and frameworks that are available. But I hope the key message of this presentation is clear: that OpenEdge 10 provides the language elements that you need share to data with many client-side environments, as JSON or in other standard formats, and OpenEdge Architect lets you define the kinds of projects and servers you need to use to do development of all the different pieces of a distributed application with a modern user interface.