BUILDING A DIAGRAMMING TOOL WITH RADDIAGRAM FOR ASP.NET AJAX

Telerik

CONTENTS

The Big Picture ... 3
Architecture of the App ... 4

Shapes Toolbox ... 7
Layout and Appearance ... 7
Interactive shapes ... 7
Server-side architecture ... 8

Properties Panels ... 10
Canvas & Layout ... 10
Shape Properties ... 11
Align and Arrange ... 11
Connection Properties ... 12
General concept of the Toolbar ... 12
Save diagram ... 13
Open diagram from JSON file ... 14

Final thoughts ... 16

Feedback ... 17
About the Author ... 17

THE BIG PICTURE

The making of an application is a challenging task with lots of difficult decisions to take. You should always consider the scalability of the application, its ability to grow and become better and at the same time performance should be top notch. The software architecture should have all these under consideration.

At the same time creating a web application can be quite a rewarding activity. You create a system out of simpler elements, you construct the complexity of the intercommunication and you are the one to bring this all to life. It is more or less a great experience.

Overall this is how we feel when we are presented with the task of creating a sample application using the Telerik UI for ASP.NET AJAX controls. The excitement is amplified when we are using a brand new control and more so when that control presents us the wonderful universe of diagramming. That is why we want to share this journey with you.

Here is a list of some of the cool things we will be discussing in the following pages:

- Extensive client-side development – the Diagram control is mainly managed on the client-side.
- Global JavaScript functions can be good.
- Communication across user controls on the client-side.
- OOP principles
- Separation and modularity of functionality. The proximity principle in action – similar features are close to each other with higher cohesion.
- Fine-tuning the client-server communication for files over HTTP – or can we go without the overhead

Architecture of the App

The Diagram sample app in its majority is a single page application built around the RadDiagram control. It is only the "About" page that is a separate Page instance.

We intended this application as an example how to use the Diagram features, how to extend the control and how to integrate it with other controls from the Telerik UI for ASP.NET AJAX suite.

The "Default.aspx" file contains the backbone of the HTML layout.

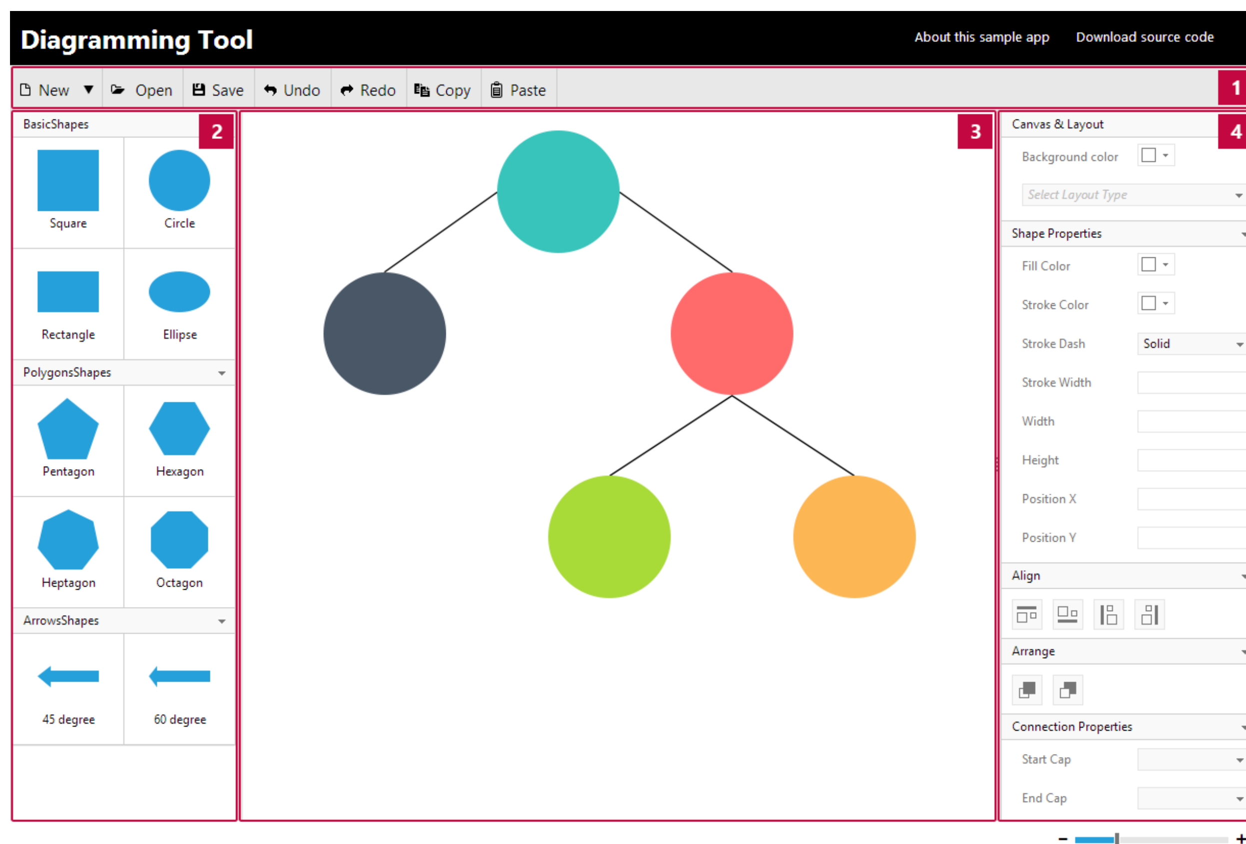

Figure1: Default.aspx with designated functionality areas: 1) Toolbar 2) Shapes toolbox 3) Diagram canvas 4) Properties panels

Every group of UI elements related to a specific functionality is separated in a user control. The intention is to have each group of features in its own space.

The majority of the functionality and logic is implemented on the client-side and most of it is centered on the Diagram control. In that regard understanding JavaScript is of great importance.

The Diagram control on the client-side is actually the **Kendo UI Diagram widget**. This is great news as we can take full advantage of all its functionality and **API**.

Because we have a single diagram in the application we can create a shortcut as a convention to getting a reference to the diagram widget through a global client-side method – getDiagram().

```
function getDiagram() {
    return $find("<%= theDiagram.ClientID %>").kendoWidget;
}
```

Although not the best programming practice, this saves a lot of effort and proves that blindly following the rules is not always the most optimal path.

There are a few very interesting cases, where communication between different user controls on the client-side is needed. Running the sample app you will notice that when opening a diagram from a JSON file or when loading a blank diagram, the properties panels are reset.

For the special case of properties handling we decided to implement this through a centralized broadcaster class – PropertiesManager. Specific implementation objects from the properties user controls register to it. Let's call them Configurators.

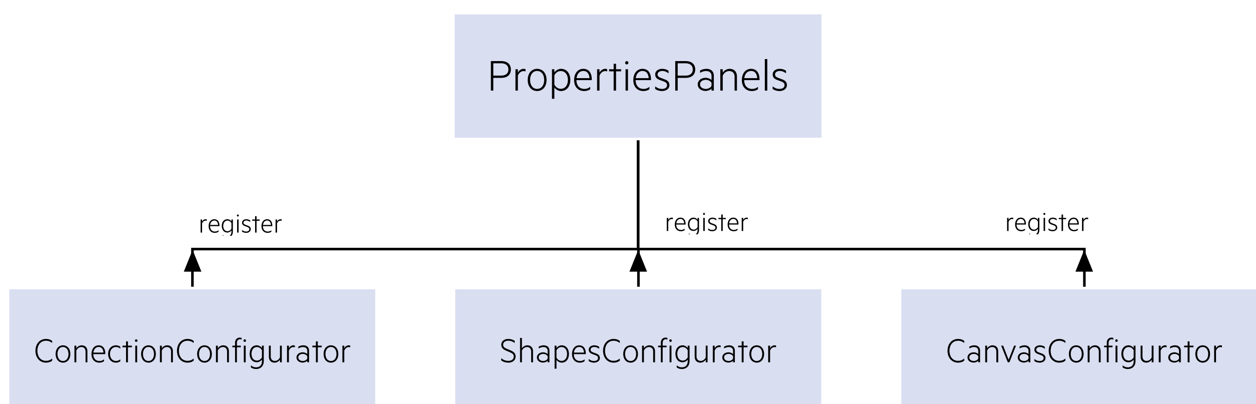

Figure2: Configurators register to the PropertiesPanels Manager

It is important that the Configurator expose a predefined collection of methods. For our case it is pretty simple – they only need the "reset" method. Finally the Toolbar user control uses an instance of the PropertiesManager – PropertiesPanels, to call its "reset" method. It will invoke the reset on every registered implementation, which results in all properties panels being reset.

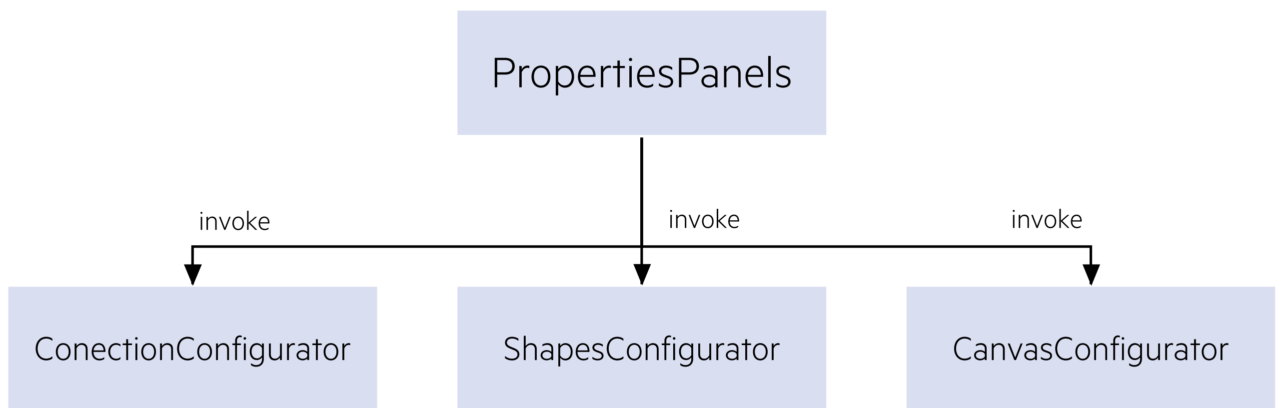

Figure3: Configurators register to the PropertiesPanels Manager

Let's investigate the different user controls in details, thus getting a better picture of the specifics in the implementation.

SHAPES TOOLBOX

Layout and Appearance

The Shapes Toolbox consists of shapes that can be dragged and dropped onto the diagram canvas. It is divided into three groups – Basic Shapes, Polygon Shapes and Arrow Shapes.

Here are the controls from the UI for ASP.NET AJAX suite that we used:

- RadPanelBar
- RadListView

The PanelBar is the main container in the “ShapesToolBox.ascx”. It gives us a flexible UI through the collapsible panels and a clear separation between the different groups of shapes.

Within each PanelItem we have a ListView that iterates the different shapes within the group. All the ListViews have the same ItemTemplate so it will be enough to investigate just one template:

```
<ItemTemplate>
<div class="item-container">
        <div class="item" data-item='<%# DataBinder.Eval(Container.DataItem, "Json") %>'
style="background-position: <%# DataBinder.Eval(Container.DataItem, "SpritePos") %>"></div>
        <span><%# DataBinder.Eval(Container.DataItem, "Name") %></span>
    </div>
</ItemTemplate>
```

From this template 3 properties show up as shape-defining.

The Json property contains the options that specify a shape in the Diagram widget. All the possible properties can be seen in the Kendo UI documentation - **http://docs.telerik.com/kendo-ui/api/dataviz/diagram#configuration-shapes**. This data is used by the built-in drop functionality of the Diagram widget to create and add the new shape.

A shape item is visualized using a well-known technique - **sprites**. Therefore for every specific shape we require the SpritePos property during data-binding to define the sprite position that corresponds to this particular shape.

The Name property is self-explanatory – it specifies the name of the shape – circle, square, rectangle, hexagon, arrows, etc.

Interactive shapes

The appearance of the shapes is irrelevant if they are not easy to use. For that we use the **Kendo UI draggable widget**, because it integrates seamlessly with the Diagram widget and it being a drop target.

```
function enableDraggableShapeBox() {
    var draggables = $telerik.$("#<%= shapesPanels.ClientID %>");
    if (draggables.getKendoDraggable()) return;
    draggables.kendoDraggable({
        filter: "div.item",
        hint: function (draggable) {
            var hint = draggable.clone(true);
            return hint;
        }
    });
}
```

We make sure that only the shape items are by specifying the filter option of the kendoDraggable. Furthermore we provide the hint – a clone of the shape HTML. This creates a nice illusion of taking a shape and really dropping in the diagram surface.

Server-side architecture

The server-side implementation is there to support the complexity of shapes – rectangular, circular and paths. At the same time we want to keep things simple. Evident from the ListView template the information that we really need is a model with 3 properties – Name, Json and SpritePos. Therefore – Interface!

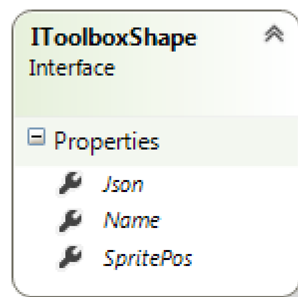

Figure4: ToolboxShape interface exposing the minimal data model for a Toolbox shape

The common code is implemented in the ToolboxShape abstract class.

The specifics – more precisely the Json generation, are hidden in specific classes – Rectangle, Ellipse and PathShape.

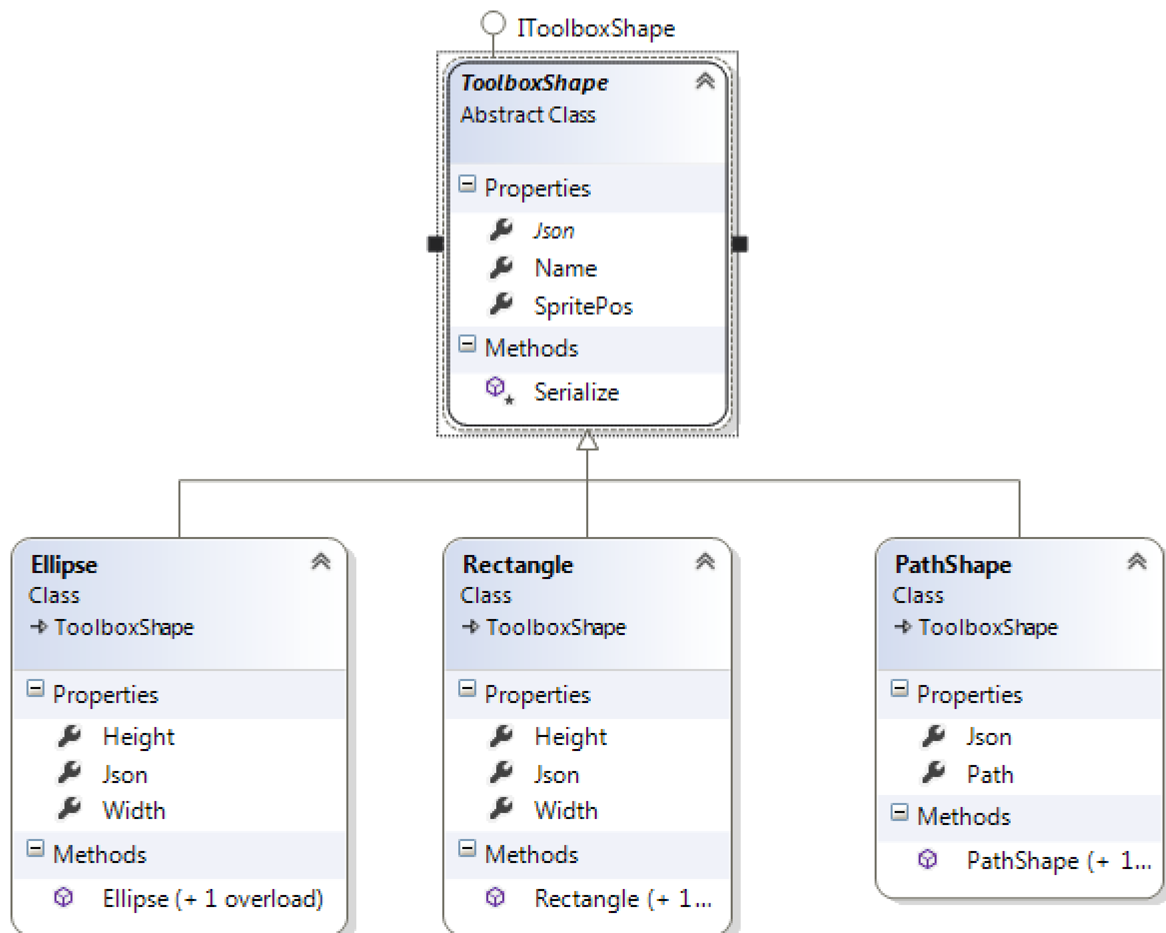

Figure5: Hierarchy of Shape primitives for the Shapes Toolbox

Here are examples how different shapes are created.

```
new Rectangle(120, 120) { Name = "Square", SpritePos = "0 0" };
new Ellipse(120, 120) { Name = "Circle", SpritePos = "-60px 0" };
new PathShape("m30,0 L90,0 L120,52 L90,104 L30,104 L0,52 z") { Name = "Hexagon", SpritePos = "-
300px 0" };
```

The whole architecture becomes obvious in the ToolboxShapesRepository, which serves as the provider class for the different shape groups.

PROPERTIES PANELS

The properties panels are user controls placed in a RadPanelBar. This gives the properties panel a nice look and feel with clear separation of the different features and at the same time promotes a flexible ways to handle the UI. This gives us the option to easily expand the options as new features in the Diagram control appear.

There are no HTML tables used in the layout of the user controls. We tried to keep things simple here as well.

Each user control takes care of its own concrete relationship with the Diagram control. When necessary a Configurator is registered in the PropertiesPanels manager class so that reset can be done on a centralized level through a single method call.

Canvas & Layout

The Diagram control itself does not expose any API for changing the background color of the canvas. Actually it does not need to – we use CSS to change the background-color of the diagram HTML element.

The case with **Diagram layouts** is much more interesting. We use the RadComboBox control as a UI element for choosing between different built-in layouts.

Notice that the combo box items are not standard text, but have icons in them. We use an ItemTemplate to implement this appearance:

```
<ItemTemplate>
    <div class="ci-layout">
        <span class="layoutIcon <%# Eval("CssClass") %>"></span>
        <span class="layoutText"><%# Eval("Text") %></span>
    </div>
</ItemTemplate>
```

The CssClass property is needed because sprites are used for the layout icons.

The Value of the ComboBox items is a combination of layout **types** and **subtypes** separated by a dash: tree-down, tree-tipover, tree-mindmaphorizontal, etc. Both values are necessary to best describe the options when calling the **diagram layout method**.

Following is how we handle the OnClientSelectedIndexChanged event of the ComboBox with all the considerations described above:

```
function changeLayoutHandler(dropDown, args) {
    var diagram = getDiagram(),
        itemValue = args.get_item().get_value(),
        layoutType = parseLayoutType(itemValue);
    diagram.layout({
        type: layoutType.type,
        subtype: layoutType.subtype,
        animate: true
    });
}

function parseLayoutType(value) {
    var valueSplit = value.split('-');
    return {
        type: valueSplit[0],
        subtype: valueSplit[1]
    };
}
```

Shape Properties

The shapes properties panel appears huge in comparison with the rest, but there actually is quite simple.

As it should track changes in the selection of the diagram or changes in the shape bounds, in this user control the **select** and **itemBoundsChange** events of the Diagram widget are handled.

At the same time any changes in the form elements should be reflected in the diagram selection. This functionality is more or less a two-way binding to the Diagram widget.

This user control takes advantage the shape.bounds method and the shape.redraw methods of a Shape object in the Diagram widget. The first method is to change the dimensions and position of the shape as the name of the method suggests, whereas the second method is used to change the appearance – fill and stroke.

Align and Arrange

Align and Arrange user controls are very similar in their implementation. In both RadButton controls are used to provide an easy way to align shapes or arrange them in the canvas.

The Diagram widget methods that are used are **alignShapes** for aligning and **toFront** and **toBack** for arranging.

Connection Properties

The connection properties user control works with the selected connections. It changes the start and end caps of the connections through a predefined, built-in markers.

The implementation uses the redraw method of a Connection by defining the startCap or endCap option in the options parameter.

```
function updateStartCap(sender, args) {
    updateSelectedConnections({ startCap: sender.get_value() });
}
function updateEndCap(sender, args) {
    updateSelectedConnections({ endCap: sender.get_value() });
}
function updateSelectedConnections(options) {
    var diagram = getDiagram(),
        selection = diagram.select();
    for (var i = 0; i < selection.length; i++)
        if (selection[i] instanceof Connection)
            selection[i].redraw(options);
}
```

General concept of the Toolbar

The implementation of the Toolbar at the top of the "Default.aspx" page is intended as a very thin layer between the user and the Diagram control. As a first simple step we want to evade the common switch/case approach in the event handler for the OnClientButtonClicked client-side event.

Here is the idea:

1. Create a JavaScript class that exposes methods with the same names as the CommandName of the corresponding Toolbar buttons.
2. In those methods implement the specific logic for the command.
3. Provide the dependent diagram as a parameter through the constructor of the JavaScript class.

The name of the class is DiagramToolBarActionManager. Here are a couple of methods that are part of it:

```
undo: function() {
    if(!this._diagram) return;
    this._diagram.undo();
},
redo: function() {
    if(!this._diagram) return;
    this._diagram.redo();
}
```

This approach is straight-forward and results in a very clean implementation of the event handler.

```
function toolBarClicked(sender, args) {
    var item = args.get_item(),
        action = item.get_commandName(),
        actionArgs = item.get_commandArgument();

    if (actionManager && action && actionManager[action])
        actionManager[action](actionArgs);
}
```

In the above code sample the responsibility for the handling of the specific command is relayed to the ActionManager.

This described approach is a slightly more complex implementation of a **well-known paradigm of evading switch/case statements in JavaScript through the use of vanilla JS objects**.

So far things look simple. However they get a bit complicated when client-server boundaries are crossed as is the case with

- Saving the diagram to a JSON file
- Loading the diagram from a JSON file
- Loading a predefined diagram from a JSON file

Save diagram

You can use the save method of the Diagram widget to get the content in a JSON format. This JSON is enough to recreate the same diagram through the load method.

As a next step that is not built-in in the Diagram widget, we need to implement saving of the JSON content to a file. The traditional approach to provide a downloadable content in web applications is to send the content to the server, where it is redirected back to the browser through HTTP as a file stream.

On the other hand the HTML5 standard comes with a specification for **File API**. This means that files can be created and its content can be managed on the client-side without even touching the server. This saves a lot of traffic and improves performance by eliminating the overhead of the client-server communication.

Both approaches are implemented in this sample app. For brevity we will investigate only the steps in the mentioned approaches skipping a lot of details. Remember that you can always download the sample code and investigate the specifics.

In the first approach, when sending the content to the server, we should use these steps:

1. Create dynamically a form element
2. Add a hidden input to it

3. Set the JSON of the diagram as a value of the input
4. Submit the form to a specific generic handler – DownloadJSON.ashx

The server code is very simple:

```
var json = context.Request.Form["json"];
context.Response.AddHeader("content-disposition", "attachment; filename=diagram.json");
context.Response.ContentType = "application/json";
context.Response.Write(json);
```

The tricky part is specifying the content-disposition HTTP header as attachment.

For modern browsers that support the File API we don't need to go to the server:

1. Create a Blob with the JSON content
2. Dynamically create an anchor tag
3. Set the download property of the anchor to the name of the file – diagram.json
4. Call the click method of the anchor to explicitly execute the default functionality.

Open diagram from JSON file

For opening a JSON file to load into the diagram we use RadAsyncUpload along with the RadAjaxManager. The idea is to upload the JSON content to the server, store it temporarily in the HTTP server cache and load it through an AJAX call.

The first step – uploading the file to the server, is easy. The functionality is already present in the RadAsyncUpload. By positioning the control under the Open Toolbar button, a click of the mouse will open the OS File chooser dialog and after choosing, will stream the content to the server.

In the OnFileUploaded server-side event handler we should store the content of the uploaded file to the HTTP cache:

```
Context.Cache.Insert(GetCacheKey(), content, null, DateTime.Now.AddMinutes(20), TimeSpan.Zero);
```

At this point we still don't have the content on the client-side, where we actually need it. Therefore in the OnClientFileUploaded client-side event handler, that is raised when the file is already uploaded on the server, we do an AJAX postback through the RadAjaxManager:

```
$find("<%=RadAjaxManager.GetCurrent(this.Page).ClientID%>").ajaxRequest("fileUploaded_trigger");
```

In the server-side event handler for the Ajax_Request, get the cached file content and if it still exists, push it back as a parameter in a JavaScript method call:

```
(sender as RadAjaxManager).ResponseScripts.Add(String.Format("loadDiagram({0})", json));
```

As a result the loadDiagram client-side method will be called with the JSON content as a parameter. In that method just call the Diagram widget load method and, voilà, you have loaded the JSON file into the diagram widget.

```
function loadDiagram(json) {
    if (actionManager) {
        actionManager.open(json);
    }
}
```

FINAL THOUGHTS

In this walkthrough of the diagram sample app, we got to know not only the RadDiagram control but a whole range of other controls from the Telerik UI for ASP.NET AJAX. We learned how to use them in respect to solving problems specific to the domain of web applications. What becomes obvious is that the controls themselves are great, but being able to combine them in a coherent manner they show their real power.

Feel free to download the code, try out the new Diagram control and let us know what you think about it.

FEEDBACK

The Diagramming control that is part of the Telerik UI for ASP.NET AJAX offers even more that what you can see in the sample app. Take some time to check out its cool features and the power it harness. Combined with the perfomance and extensibility of the whole suite you can feel more confident in a successful end result.

In addition, the trial comes with 30 days of dedicated technical support from the people who build the controls.

Download a fully-functional trial of Telerik's **more than 80 Telerik controls** for every project need. You will find controls for data visualization and management, navigation and layout, editing, interactivity and more.

APPLICATION SOURCE CODE

The code for both samples is available for download **here**.

About the Author

Nikodim Lazarov is a Senior Software Developer in the Telerik's ASP.NET AJAX division. He has years of experience building web applications of various scales. Niko is a strong advocate of Web standards, TDD, Agile and basically anything that makes every developer a true craftsman.

You can find him on Twitter at **@nlazarov** and can reach him at **nikodim.lazarov@telerik.com**.

This content is sponsored by
Telerik UI for ASP.NET AJAX
and
Telerik UI for ASP.NET MVC.

With over 150 UI controls and thousands of scenarios covered out-of-the-box, you can leave the complexities of UI to us and focus on the business logic of your apps.

TRY ASP.NET AJAX CONTROLS

TRY ASP.NET MVC EXTENSIONS