Sitecore

SPEAK Developer Guide

UX/UI accelerator for Sitecore's developer

Table of Contents

Chapter 1

Introduction

SPEAK is here to help you as a developer by trying to make things as easy as possible.

If SPEAK does not help you, makes you develop faster and increases the quality of your UI, SPEAK has failed.

Sitecore SPEAK is a technology for building web applications in Sitecore.

While standard Sitecore is very usable to build information web site and deep business integrations, it misses some key components to make the building of applications easy. SPEAK extends Sitecore to provide these components.

SPEAK is a replacement for Sheer UI and is built upon standard technologies like Html5, CSS3, jQuery, Backbone, Knockout, Require, Twitter Bootstrap and more.

Sitecore SPEAK provides a library of components that accelerates development (the Component Library).

The overarching goals of SPEAK are to:

- Provide a strict and streamlined approach to developing application UIs in Sitecore
- Enable reuse of UI elements
- Enforce a consistent look and feel
- Provide automated testing

The SPEAK project is a UI/UX framework which runs on top of an existing Sitecore instance. You will need to have an up and running sitecore instance on your workstation in order to install it.

Chapter 2

The commandments of SPEAK

Print these commandments and hang them on your wall.

2.1 Commandments

Thou shall:

- Using Sitecore layouts, sublayouts and renderings for every UI element
- Use Html5
- Use CSS3
- Use jQuery
- Obey SPEAK design guidelines
- Obey SPEAK interaction guidelines
- Use JSON as format for transporting data between client and server
- Reuse controls in the Component Library
- Use semantic names for Place Holders
- Use Bootstrap grids
- Use the SPEAK-Layout layout for every page
- Provide automated tests
- Provide BDD oriented tests
- Use the CSS classes provided by SPEAK

2.2 Sins

Thou must not:

- Use Xml Controls
- Use Sheer UI
- Use Xaml#
- Require manual/smoke testing
- Specify fonts or colors anywhere
- Use Item Paths – always use Item IDs.

2.3 Virtues

Thou can:

- Create components that can be reused

Chapter 3 Architecture

SPEAK is about embracing the UI platform which for SPEAK is the web browser.

SPEAK is built upon standard and well known technologies.

- Sitecore layouts, sublayouts and renderings
- Sitecore MVC
- Sitecore Item Web API
- jQuery
- Backbone
- Knockout
- Require
- Twitter Bootstrap

SPEAK is the glue that binds these components together.

3.1 Requirements

SPEAK must be supported in:

- Internet Explorer 9+
- Chrome
- Firefox

SPEAK requires:

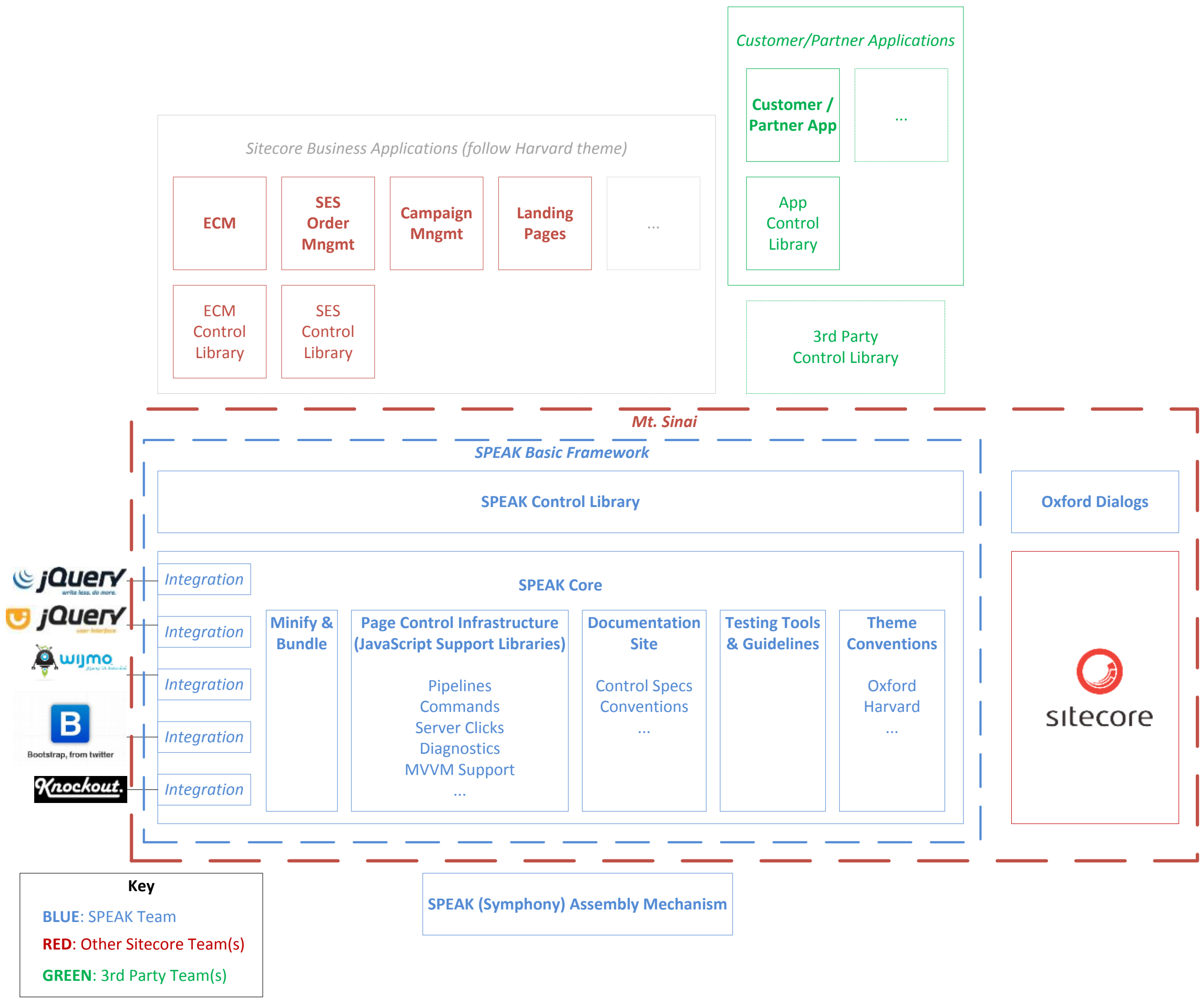
- Html5
- Css3

SPEAK uses a significant amount of JavaScript, so the requirements for the client machine is expected to be higher than the requirement for a machine running classic Sitecore.

SPEAK is not designed to require Sitecore MVC, but the initial version is built on it. There are very few bindings on MVC, and it must remain so, so that SPEAK may easily be ported to WebForms, if required.

3.2 Architecture diagram

SPEAK is the mediator between the client and the server and the foundation upon which the various UIs of Sitecore are built.



3.3 External documentation

Since SPEAK is using standard technologies, a lot of documentation is already available.

Component	Link
jQuery	http://docs.jquery.com/Main_Page
Backbone	http://backbonejs.org
Knockout	http://knockoutjs.com/documentation/introduction.html
Twitter Bootstrap	http://twitter.github.com/bootstrap/
Require	http://requirejs.org/

It is highly recommended to run the Knockout tutorial as it is quite amazing.

3.4 Components

The basic element in SPEAK is a component. In Sitecore terms, a component is implemented as a Sitecore rendering.

A component (aka control) is a UI element and consists of a Backbone model and view. The model is the data structure that exposes the properties of the component and the view handles the interaction between the browser and the model.

3.5 Models

As mentioned SPEAK uses models and views, meaning the each component has a model part, that is a pure data structure, and a view part that handles the interaction between the Html and the model.

In the JavaScript world there are 2 predominate libraries for this separation: Backbone and Knockout. While Backbone is MVC, Knockout is MVVM. Both libraries are equally popular and have their own strengths and weaknesses.

Backbone is very good at data-modeling while Knockout excels at data-binding between the model and the Html.

From a business perspective, it would be difficult to choose one library over the other, as we would lose half of the developers, if we only supported one. As a platform SPEAK cannot impose a single library. Instead SPEAK strives to support both libraries, trying to expose the best of both libraries: Backbone for the model and Knockout for the data-binding.

As such SPEAK supports 3 types of models: Hybrid (default), Backbone and Knockout.

A model in SPEAK is always a Backbone model, and the 3 types determine how Knockout is used in the model.

3.5.1 Hybrid model

The Hybrid is the default model. SPEAK create a special attribute on the model "viewModel" and every Backbone attribute is copied to the "viewModel" as a Knockout observable. This means that you have a pure Backbone model and a pure Knockout model. SPEAK automatically synchronizes values between the Backbone and Knockout models.

When data-binding SPEAK applies the "viewModel" attribute to the Html, which allows you to use markup, like this: data-bind="text: name". The text is bound to the "name" observable on the "viewModel" attribute of the model.

Inter-model bindings or cross-bindings is handled on the Backbone attributes, but as SPEAK synchronizes the values between the Backbone and Knockout model, any changes are automatically reflected in the viewModel.

The Hybrid model is enable by setting "ko: true" in the model. This is done automatically on all base classes like ComponentModel and ControlModel.

3.5.2 Backbone model

When using a Backbone model, the "viewModel" attribute is not created. Backbone models are determined by having "ko: false".

Backbone models do not usually work with Html data-binding as they do not have the “viewModel” attribute. However SPEAK will data-bind to the model, if the “viewModel” attribute is not present, so any values (not Backbone attributes) that you put on the model can be data-bound.

Cross-binding still works.

3.5.3 Knockout model

In a Knockout model, you choose to ignore the fact that every model in SPEAK is a Backbone model, and just create observables directly on the model. Knockout models have “ko:false” as the “viewModel” attribute is not needed.

Cross-binding does not work.

As the “viewModel” attribute is not present on the model, SPEAK will bind to the model instead, and since you have created all your observables are created here, everything works.

3.6 Data binding

Data binding in SPEAK works in the same way as traditional data binding, but comes in two variants; Knockout data binding and cross-binding data binding.

The first variant is to synchronize the model and the UI, and for this Knockout is used. General Knockout bindings are usually sufficient, but SPEAK contains a few custom bindings.

Please refer to the Knockout documentation for further information.

The other variant is to synchronize properties between Backbone models. For instance a data source component has a ‘searchText’ property that is synchronized with the ‘text’ property of a text box – whenever the ‘text’ property is changed, the ‘searchText’ property is updated and the data source is queried again.

Cross-binding is a custom implementation, but the properties are still Backbone attributes.

3.7 Centralized Page Code

Each page has a centralized JavaScript code file that has access to all components in the page. This object is called an application.

The Page Code is responsible for initializing the page. This is analogue to CodeBehind in ASP.NET or the form code in WPF/WinForms.

In the Page Code, each component is represented by en intermediate object which has two properties; 'model' and 'view'. This allows access to both the model and the view.

The Page Code is implemented by the 'PageCode' rendering which must be present in all pages.

When the page is initialized, the Html is scanned for components and each component is initialized. As part of the component initialization, a reference to the component is stored on the current application object.

For instance the page contains a 'TextBox' rendering which defines a text box with an ID of "SearchTextBox". When this component is initialized, the reference Sitecore.app.SearchTextBox is created. This means that the text property of the SearchTextBox can be obtained by writing:

```
var text = this.SearchTextBox.get(“text”);
```

The purpose of the Page Code is to provide a familiar and simple programming model for the page on the client side. As in ASP.NET, WPF and WinForms the Page Code typically handles UI events like clicks and selections and updates the page model to reflect any changes.

3.7.1 Applications

In SPEAK an application is an object that contains references to a number of components on the page. SPEAK supports multiple applications in the same page.

The default application is called 'app' and can in debug mode be access using 'Sitecore.app'.

Multiple applications enabled easy integration between various separate subsystems.

3.8 Sitecore layouts, sublayouts and renderings

All pages in SPEAK are built using standard Sitecore layouts, sublayouts and renderings. SPEAK requires no changes to standard Sitecore layouts, but imposes a strict approach.

A SPEAK page consists of a layout, a grid and a number of renderings (usually quite a lot of renderings).

The layout is always the same – the SPEAK-Layout layout.

DO use the SPEAK-Layout layout.

The basic layout of the page is specified by the grid. The grid is usually a sublayout (or an MVC view renderings) that uses the Bootstrap grids. SPEAK provides a number of default grids.

DO use one of the standard grids.

The layout and the grids provide a standard set of place holders that makes it easy to switch basic grid, if necessary.

The names of place holders should always be semantic and not refer to position, size or color.

DO use names like Body, Bar, Buttons, ToolBar or Pane.

AVOID using names like Left, Right, Top, Bottom, Wide, Small, Blue, Red, Header and Footer.

3.9 Debugging

If you enable debugging (by adding sc_debug=1 in the URL), SPEAK annotates the Html with rendering information, if available.

The following information is added:

Html attribute	Description
data-sc-debug-renderingname	The name of the rendering that outputted this Html
data-sc-debug-renderingid	The id of the rendering
data-sc-debug-datasource	The data source of the rendering
data-sc-debug-contextitemid	The current context item id
data-sc-debug-renderingparameters	The rendering parameters

3.10 Grids

Every page has a grid. This grid defines the basic layout of the page.

Traditionally there have been a lot of approaches to specifying the grid, some more successful than others. It can be quite tricky to ensure that the grid works on all supported browsers.

SPEAK requires all grids to be specified using the Bootstrap grid. This provides a strict approach to the basic layout, while ensuring consistency and platform compatibility.

Furthermore it provides a single point for extended the grids, e.g. to support responsive design.

You should never have to build a new grid. SPEAK should provide you with sufficient pre-defined grids. If the desired grid is not available, the design may have to be reconsidered.

3.11 Items

Sitecore items in SPEAK are represented by the SItecore.Definitions.Data.Item class.

In general items are fetched from the server using the Item Web API. The Item Web API has been wrapped in several APIs: Database API, Backbone Sync and Data Sources.

3.11.1 URIs

Databases, items and fields are uniquely identified by URI classes.

The Sitecore.Definitions.DatabaseUri class points to a database by name. To get a database instance pass a DatabaseURI parameter to the constructor of the database:

```
var databaseUri = new Sitecore.Definitions.Data.DatabaseUri(“master”);
var database = new Sitecore.Definitions.Data.Database(databaseUri);
```

The approach may seem cumbersome, but it provides a lot of flexibility.

A database URI also contains information that is used by the Item Web API. It has a webApi property that specifies the Item Web API URL prefix (“/-/item/v1”) and a virtualFolder property that is used by Sitecore to resolve the site, e.g. setting virtualFolder to “/sitecore/shell” forces the Item Web API to use the “shell” site.

The Sitecore.Definitions.ItemUri class points to an item. The item is identified by a DatabaseUri and an ItemId. The ItemId is a Guid.

```
var databaseUri = new Sitecore.Definitions.Data.DatabaseUri(“master”);
var itemUri = new Sitecore.Definitions.Data.ItemUri(databaseUri, “{110D559F-DEA5-42EA-9C1C-
8A5DF7E70EF9}”);
```

The Sitecore.Definitions.ItemVersionUri class points to an item version. The version is identified by an ItemUri, a language and a version.

```
var databaseUri = new Sitecore.Definitions.Data.DatabaseUri(“master”);
var itemUri = new Sitecore.Definitions.Data.ItemUri(databaseUri, “{110D559F-DEA5-42EA-9C1C-
8A5DF7E70EF9}”);
var itemVersionUri = new Sitecore.Definitions.Data.ItemVersionUri(itemUri, “en”, 0);
```

The Sitecore.Definitions.FieldUri class points to a field in an item version. The item is identified by a ItemVersionUri, and a FieldId. The FieldId is a Guid.

```
var databaseUri = new Sitecore.Definitions.Data.DatabaseUri(“master”);
var itemUri = new Sitecore.Definitions.Data.ItemUri(databaseUri, “{110D559F-DEA5-42EA-9C1C-
8A5DF7E70EF9}”);
var itemVersionUri = new Sitecore.Definitions.Data.ItemVersionUri(itemUri, “en”, 0);
var fieldUri = new Sitecore.Definitions.Data.FieldUri(itemVersionUri, “{37762017-2CD5-4F48-
AF91-407791FE46AC}”);
```

3.11.2 Database

The Sitecore.Definitions.Data.Database class represents a database. It is primarily used to obtain items.

To instantiate a database, pass a DatabaseUri in the constructor.

```
var databaseUri = new Sitecore.Definitions.Data.DatabaseUri(“master”);
var database = new Sitecore.Definitions.Data.Database(databaseUri);
```

To get and item, use the getItem function:

```
database.getItem(“{110D559F-DEA5-42EA-9C1C-8A5DF7E70EF9}”, function(item)
  {
    alert(item.getValue(“Title”));
  });
```

The Database class contains 4 ways of getting items:

1. getItem
2. getChildren
3. query
4. search

All functions accept a callback that is invoked when the request completes.

In getItem, if the item is not found, the callback is invoked with a null item.

The functions getChildren, query and search all return an array of items.

3.11.3 Item

The Sitecore.Definitions.Data.Item class represents an item. The Item is a Backbone model and can be data-bound.

Property	Description
itemUri	The URI of the item
itemName	Name of the item.
displayName	Display name.
templateName	Name of the template
templateId	ID of the template

icon	Icon
path	Item path
fields	Array of Sitecore.Definitions.Data.Field

The Item has a number of support functions.

Function	Description
getField	Gets a Field class by field name
getValue	Gets the value of a field
setValue	Sets the value of a field

As the Item class is a Backbone model, it also supports the Backbone Sync API. However please notice that the 'create' operation is currently not supported as the semantics between the Sync API and the Item Web API differs.

To refresh an item, use the 'read' operation:

```
item.sync('read');
```

To update an item, use the update operation:

```
item.sync('update');
```

To delete an item, use the delete operation:

```
item.sync('delete');
```

3.11.4 Field

The Sitecore.Definitions.Data.Field class represents a field in an item version. The Field is a Backbone model. The Item class contains an array of fields.

Property	Description
fieldUri	The URI of the field
fieldName	Name of the field
type	Field type, e.g. "Single-line Text"
value	The value of the field

Chapter 4

Installing SPEAK

4.1 Installing a Sitecore instance

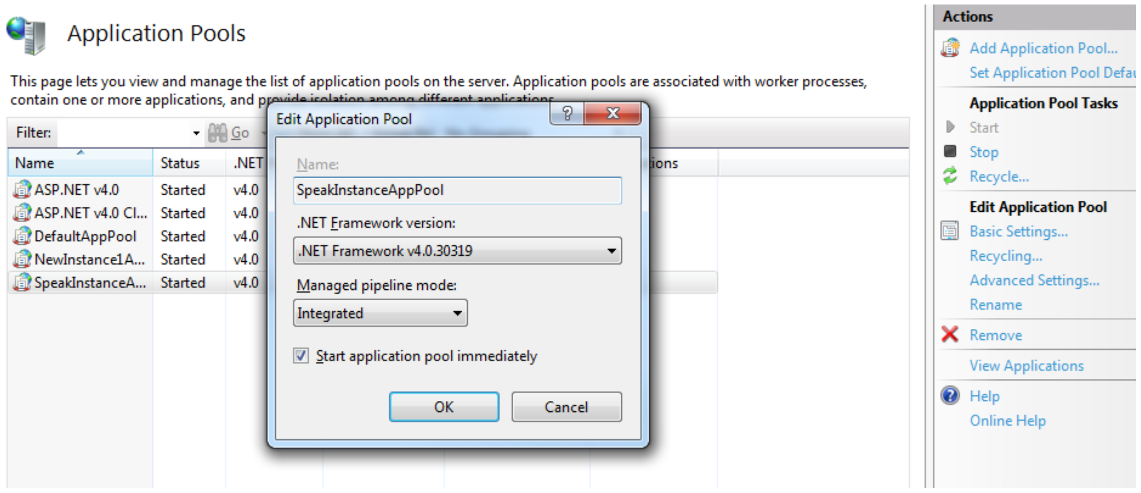
As SPEAK is running on top of a Sitecore instance, in order to use it, you need an up and running sitecore instance. Here are the basics steps:

1. Install a sitecore instance with the help of an msi.

Note: In this documentation the name of the sitecore instance will be “SPEAK.local”.

2. After you have installed sitecore, do not forget to check your application pool.

   Set the .NET framework version to .NET Framework v4.



Application Pools

This page lets you view and manage the list of application pools on the server. Application pools are associated with worker processes, contain one or more applications, and provide isolation among different applications.

Name	Status	.NET
ASP.NET v4.0	Started	v4.0
ASP.NET v4.0 Cl...	Started	v4.0
DefaultAppPool	Started	v4.0
NewInstance1A...	Started	v4.0
SpeakInstanceA...	Started	v4.0

Edit Application Pool

Name: SpeakInstanceAppPool

.NET Framework version: .NET Framework v4.0.30319

Managed pipeline mode: Integrated

☑ Start application pool immediately

3. Make sure the installation of sitecore is working.

   - Open a browser and go to http://SPEAK.local, you should see a welcome screen.

PS: Do not forget to register IIS with the Aspnet_regiis.exe command (http://msdn.microsoft.com/en-us/library/k6h9cz8h%28v=VS.100%29.aspx)

4. Go to the administration panel with this link http://SPEAKinstance/sitecore

   The default username is Admin and the default password is the letter ‘b’.

   If you are able to enter in the administration panel, chances are high that your installation went successfully.

4.2 Installing SPEAK

1. Checkout the sources from this address

   https://svn2dk1.dk.sitecore.net/svn/cms/Sitecore/trunk

2. Before installing the SPEAK files, you need to install 1 packages to your sitecore instance. The package is located under the /code/Packages folder of your source files and is called “Sitecore Item Web API”.

   - Go to the desktop sitecore admin panel > Development Tools > Installation Wizard

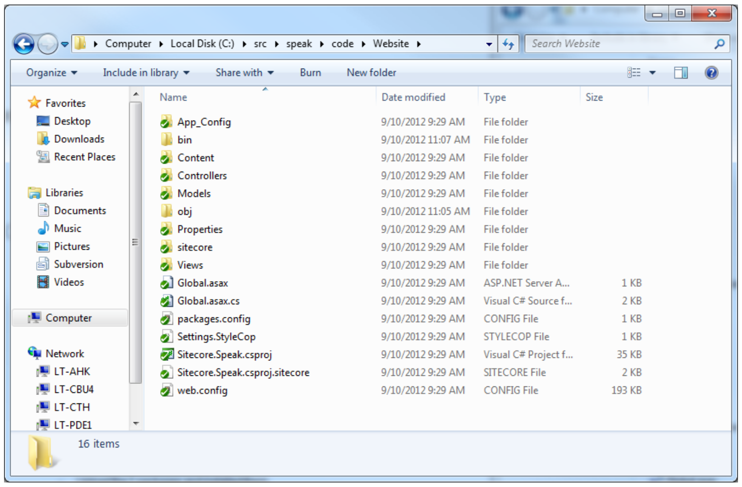
   - Upload the packages and installed it.

3. Rename the Sitecore.Mvc.config.disabled file to Sitecore.Mvc.config (in <website-root>/App_config/Include) and move Sitecore.Mvc.dll from <website-root>\bin_Net4 to <website-root>\bin

4. Now copy all the files located under the /code/website to the root of you sitecore instance (for me C:\inetpub\wwwroot\SPEAKInstance\Website).

   NOTE: do not overwrite any files except the web.config

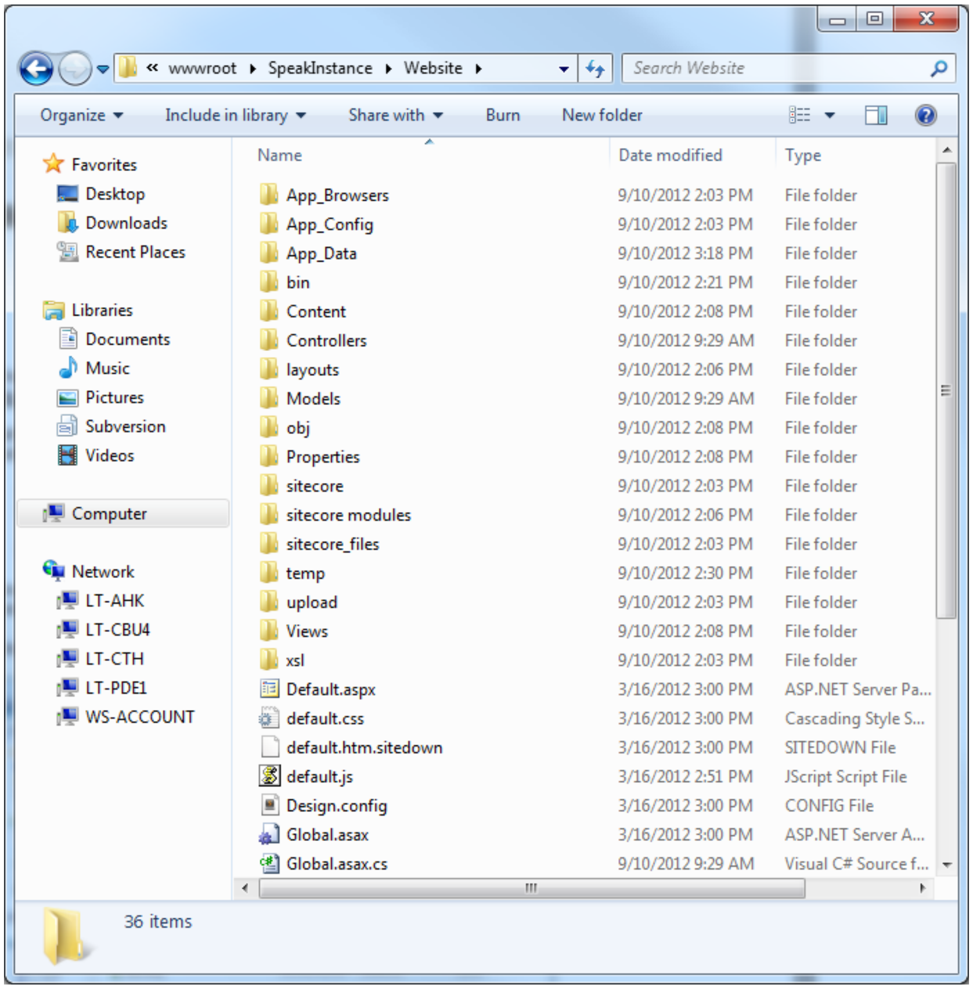
   Source you need to copy:



Name	Date modified	Type	Size
App_Config	9/10/2012 9:29 AM	File folder	
bin	9/10/2012 11:07 AM	File folder	
Content	9/10/2012 9:29 AM	File folder	
Controllers	9/10/2012 9:29 AM	File folder	
Models	9/10/2012 9:29 AM	File folder	
obj	9/10/2012 11:05 AM	File folder	
Properties	9/10/2012 9:29 AM	File folder	
sitecore	9/10/2012 9:29 AM	File folder	
Views	9/10/2012 9:29 AM	File folder	
Global.asax	9/10/2012 9:29 AM	ASP.NET Server A...	1 KB
Global.asax.cs	9/10/2012 9:29 AM	Visual C# Source f...	2 KB
packages.config	9/10/2012 9:29 AM	CONFIG File	1 KB
Settings.StyleCop	9/10/2012 9:29 AM	STYLECOP File	1 KB
Sitecore.Speak.csproj	9/10/2012 9:29 AM	Visual C# Project f...	35 KB
Sitecore.Speak.csproj.sitecore	9/10/2012 9:29 AM	SITECORE File	2 KB
web.config	9/10/2012 9:29 AM	CONFIG File	193 KB

16 items

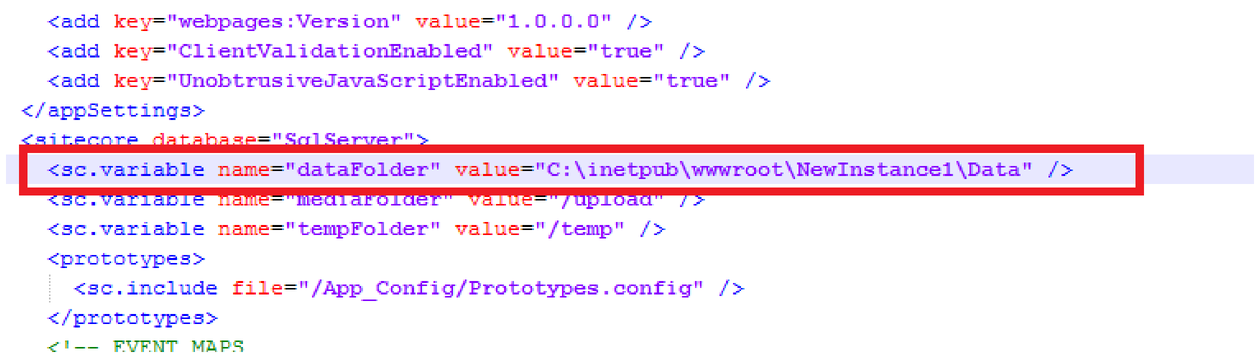
Where you need to copy all these files:



Name	Date modified	Type
App_Browsers	9/10/2012 2:03 PM	File folder
App_Config	9/10/2012 2:03 PM	File folder
App_Data	9/10/2012 3:18 PM	File folder
bin	9/10/2012 2:21 PM	File folder
Content	9/10/2012 2:08 PM	File folder
Controllers	9/10/2012 9:29 AM	File folder
layouts	9/10/2012 2:06 PM	File folder
Models	9/10/2012 9:29 AM	File folder
obj	9/10/2012 2:08 PM	File folder
Properties	9/10/2012 2:08 PM	File folder
sitecore	9/10/2012 2:03 PM	File folder
sitecore modules	9/10/2012 2:06 PM	File folder
sitecore_files	9/10/2012 2:03 PM	File folder
temp	9/10/2012 2:30 PM	File folder
upload	9/10/2012 2:03 PM	File folder
Views	9/10/2012 2:08 PM	File folder
xsl	9/10/2012 2:03 PM	File folder
Default.aspx	3/16/2012 3:00 PM	ASP.NET Server Pa...
default.css	3/16/2012 3:00 PM	Cascading Style S...
default.htm.sitedown	3/16/2012 3:00 PM	SITEDOWN File
default.js	3/16/2012 2:51 PM	JScript Script File
Design.config	3/16/2012 3:00 PM	CONFIG File
Global.asax	3/16/2012 3:00 PM	ASP.NET Server A...
Global.asax.cs	9/10/2012 9:29 AM	Visual C# Source f...

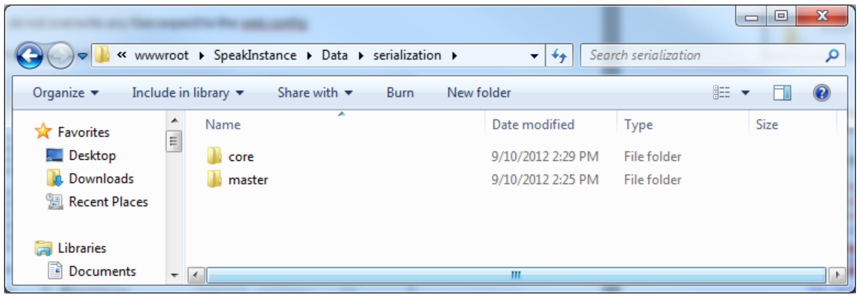
5. Open the new web.config located in the root of your sitecore instance, open it and check if the dataFolder property is set to the right path.

   Here is an example:



```
    <add key="webpages:Version" value="1.0.0.0" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  </appSettings>
  <sitecore database="SqlServer">
    <sc.variable name="dataFolder" value="C:\inetpub\wwwroot\NewInstance1\Data" />
    <sc.variable name="mediaFolder" value="/upload" />
    <sc.variable name="tempFolder" value="/temp" />
    <prototypes>
      <sc.include file="/App_Config/Prototypes.config" />
    </prototypes>
    <!-- EVENT MAPS
```

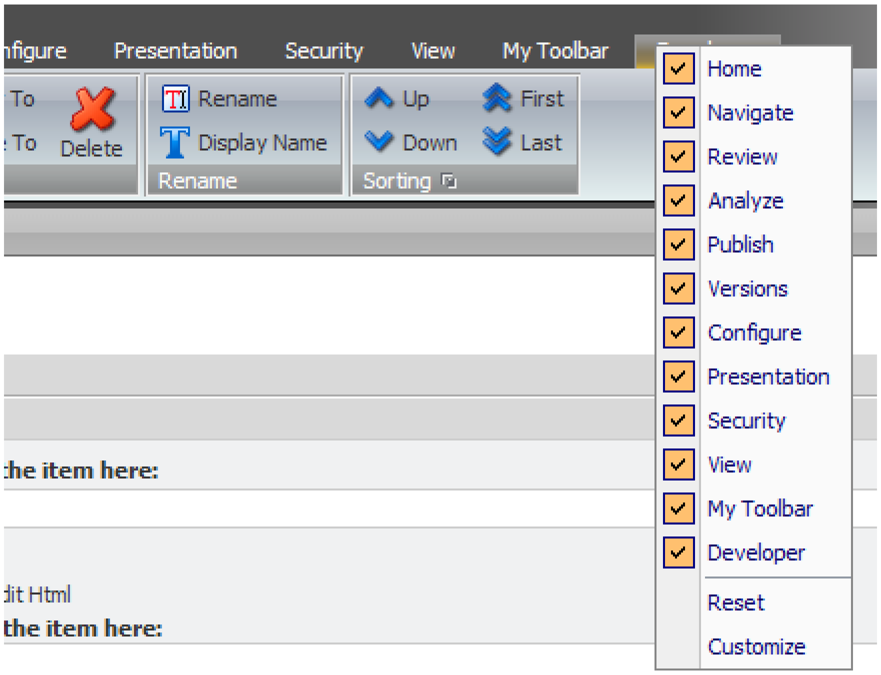
6. Go to the root of your SPEAK sources, copy all the content of the folder ‘serialization’ to the folder /path_to_your_instance/Data/serialization.



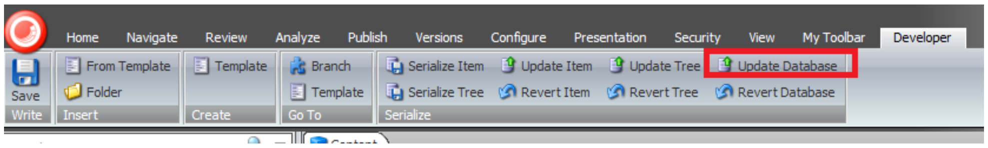
Name	Date modified	Type	Size
core	9/10/2012 2:29 PM	File folder	
master	9/10/2012 2:25 PM	File folder	

7. Go to the admin panel of your sitecore instance and open the editor.

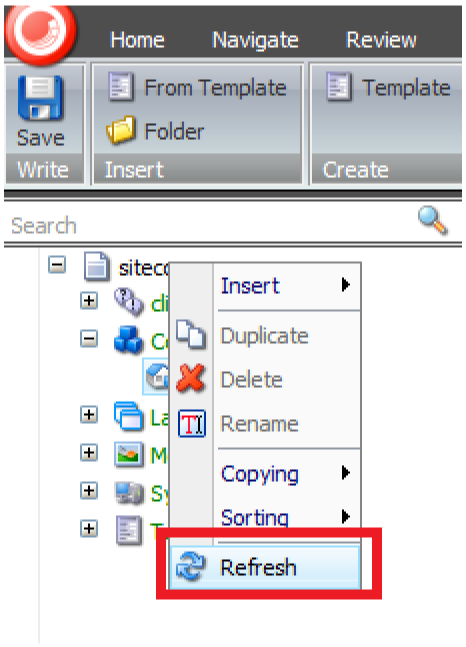
8. Right click on the top menu and check ‘Developer’ in order to have the developer toolbar.



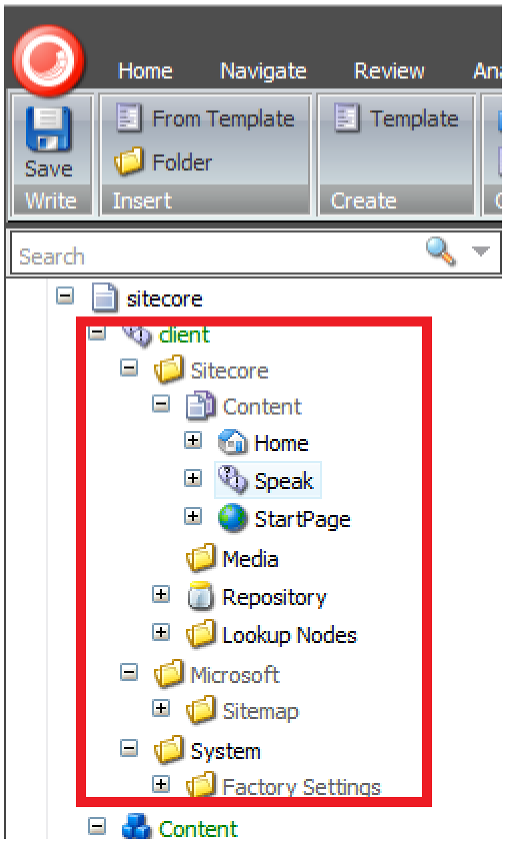
9. Go to the Developer ribbon and click on Update Database



10. Now, go to the root of your tree located in the right panel and right-click to the ‘sitecore’ node, click ‘refresh’



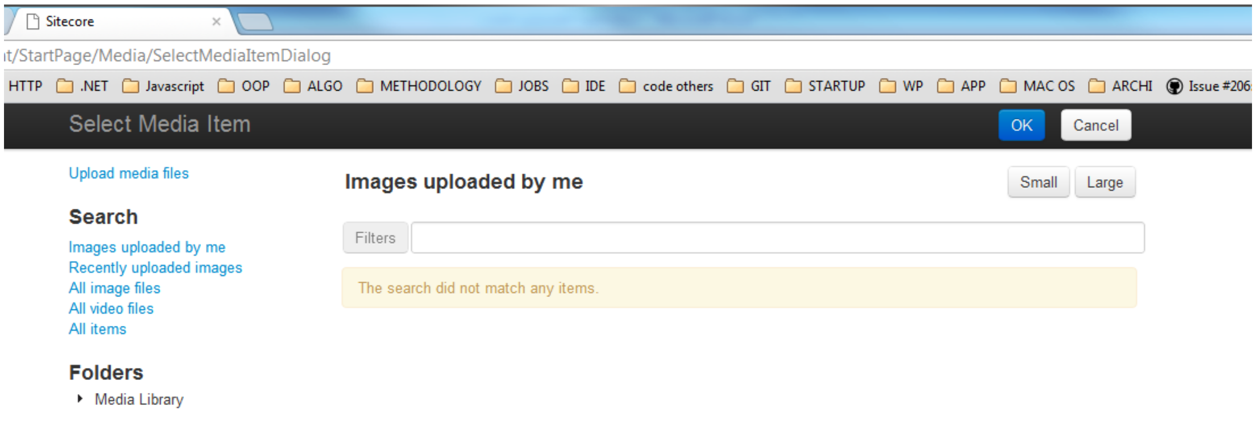
11. You should now see a node called ‘client’ in your application.



12. Go to

    http://[YOUR_INTANCE]/sitecore/client/sitecore/content/StartPage/Media/SelectMediaItemDialog

If you see something like this:



The search did not match any items.

Well done! You have successfully installed SPEAK!

Chapter 5

Building a new page

When building a new page, follow these steps:

Create a new item in Sitecore. The template of the new item should have the one of the SPEAK pages template as a base template.

Create a PageSettings item beneath the new item. The PageSettings item must use the PageSettings template.

Create a new file folder in the appropriate location. The location and name of the folder should follow the naming convention, so that the new item and the folder are connected.

Optional: Create a new Page Code JavaScript file containing the standard initialization code.

Setup the layout on the new item.

The layout must set to the SPEAK-Layout layout.

Add the PageCode rendering to the Page.Code place holder.

Optional: Set the PageCodeScriptFileName property to the Page Code JavaScript file.

Add a grid sublayout to the Page.Body place holder.

The basic page is now ready.

Chapter 6

Building a new component

When building a new component, follow these steps:

Create a file folder for the new component in an appropriate location.

Add the component JavaScript scaffolding file to the folder.

Add a MVC View cshtml file to the folder.

In the component JavaScript file, add the appropriate properties to the model.

In Sitecore, create a new View rendering that point to the cshtml file.

Create a new template beneath the rendering (if the UI does not allow you to create a template underneath the rendering, create the rendering in the Templates section and drag it to the rendering).

Modify the template to expose the properties in the model. This ensures that properties can be set from the Sitecore layout designer. Properties that can be used in data-binding may be placed in a 'Data Bindings' template section.

Set the Parameters Template field of the rendering to point to the template.

Modify the Default Parameters field of the rendering to initialize the properties.

Modify the Place Holders field of the rendering to expose the place holders of the rendering to the Sitecore layout designer. The $Id macro can be used to specify place holder names that depend on the ID of the component.

The basic component is now ready.

6.1 Creating controls

Create new control class as a derivative of the most appropriate base class. The class should have a constructor which obtains a parameter of RenderingParametersResolver type. Initialize control properties in constructor, by invoking the appropriate RenderingParametersResolver methods. Initialize Class property (if control is derived from ControlBase class) explicitly with component specific CSS class name (e.g. “sc-textbox” for textbox component) in the constructor.

```
public TextBox([NotNull] RenderingParametersResolver parametersResolver)
  : base(parametersResolver)
{
  this.Class = " sc-textbox";
  this.Text = parametersResolver.GetString("Text");
  this.Type = parametersResolver.GetString("Type", "text");
  this.IsReadOnly = parametersResolver.GetBool("IsReadOnly");
}
```

Having a parameterless constructor is preferable as well.

```
public TextBox()
{
  this.Class = " sc-textbox";
}
```

Override the Render method to output the desired mark-up.

In order to make the control accessible for MVC views, one should:

1. Add corresponding method to `Sitecore.Mvc.Controls` class:

```
public virtual HtmlString TextBox([NotNull] string controlId, [CanBeNull] object
parameters)
{
Assert.ArgumentNotNull(controlId, "controlId");

var resolver = this.GetParametersResolver(parameters);

var textBox = new TextBox(resolver)
{
    a.  ControlId = controlId
};

return new HtmlString(textBox.Render());
}
```

Please pay attention to how control is initialized: an instance RenderingParametersResolver class is created using the ‘GetParametersResolver’ factory method and then it is passed to corresponding class constructor. The control initialization logic (except ControlId in this particular case) should be placed inside the constructor. When the method is implemented, one can render the TextBox control in Razor view using the following syntax:

```
@Html.Sitecore().Controls().TextBox("TextBox1", new { Text = "Sample text." })
```

2. Add a method overload which accepts parameter of a Rendering type:

   ```
   public virtual HtmlString TextBox([NotNull] Rendering rendering)
   {
   Assert.ArgumentNotNull(rendering, "rendering");

   var resolver = this.GetParametersResolver(rendering);

   var textBox = new TextBox(resolver);

   return new HtmlString(textBox.Render());
   }
   ```

   This overload is necessary in order to enable the TextBox control as a Sitecore rendering. In the corresponding Partial Razor View , one can render the TextBox control using the following syntax:

   ```
   @model Sitecore.Mvc.Presentation.RenderingModel

   @Html.Sitecore().Controls().TextBox(Model.Rendering)
   ```

In order to pass parameters to a control using Rendering Model, one can specify Sitecore rendering parameters in layout definition or specify corresponding fields in a datasource item. Datasource item must be based on corresponding Rendering Parameters Template. Parameters set on the Layout Definition takes precedence over data source item parameters. This behavior is specified by default `DatasourceBasedParametersResolver` class. If necessary the behavior can be changed:

- Implement parameters resolving logic in a new class (CustomParametersResolver)
- Override the `Sitecore.Mvc.Controls.GetParametersResolver` method to return an instance of CustomParametersResolver class in a derived class (CustomControls).
- Register CustomControls class, using:
  `Configuration.MvcSettings.RegisterObject<Controls>(() => new CustomControls());`

6.2 Interaction

A component may provide a number of interactions, for instance a button exposes a click interaction.

Usually such a click interaction comes in 4 different variants or protocols.

Protocol	Description
click	The field contains JavaScript function that is executed when the interaction is invoked.
command	The field contains the JavaScript command that is executed when the interaction is invoked.
serverclick	The field contains an MVC controller action that is called when the interaction is invoked. The format is “controller name/action”. The controller name does not need the Controller postfix.
trigger	The field contains a trigger action.

An interaction is normally exposed as 4 fields in the rendering parameters template, corresponding to each of these protocols.

To execute an action, call the invoke method:

```
Sitecore.Helpers.invocation.execute(invocation, { app: this.app });
```

The ‘data-sc-click’ attribute contains the click/command/serverclick information to execute.

6.2.1 Triggers

The base controls (ComponentView, ControlView, ButtonBaseView etc.) listen for events on the current application. For instance you can hide the control “Button1” by triggering this event:

```
Sitecore.app.trigger(“hide:Button1”);
```

This is very powerful as it decouples the controls.

Standard events:

Event	Control	Description
set	ComponentView	Sets attributes in the model
hide	ControlView	Sets isVisible to false
show	ControlView	Sets isVisible to true
toggle	ControlView	Flips the value of isVisible
enable	InputView	Sets isEnabled to true
disable	InputView	Sets isEnabled to false
enable	ButtonBaseView	Sets isEnabled to true
disable	ButtonBaseView	Sets isEnabled to false

It is possible to pass parameters. For instance

```
Sitecore.app.trigger(“set:Button1”, { text: “Foo” });
```

The above sets the text attribute of the button to “Foo”.

It is also possible to pass parameters when designing - just set the Trigger parameter of the rendering. Parameters are specified inside parenthesis, for instance: set:Button1({“text”: “Foo”}).

When implementing a events in a control, you use the ‘listen’ attribute in the view. If an eventName contains the magic string “$this”, it gets replaced by the id of the control.

```
listen: {
  "set:$this": "set"
}
```

When creating a control that has the Html attribute data-sc-id=”Button1”, the above code will listen for events on “set:Button1”.

6.2.2 Using events

SPEAK uses Backbone events to provide a familiar programming model and facilitate decoupling between controls (for additional information please refer to the Backbone documentation: http://backbonejs.org/#Events).

AVOID: Having one control directly reference another control.

Events are usually raised on the controls application, but may also be raised on the controls model, if appropriate. Events are usually handled in a controls view, not the model.

To raise an event, do this:

```
raisingEvent: function(eventName) {
  this.app.trigger(eventName);
}
```

To listen for an event, set it up in the views initialize function:

```
initialize: function() {
  this._super();
  this.app.on(eventName, function() {}, this);
}
```

Alternatively you can use the ‘listen’ array in the view.

The name of the event should follow this convention: “action:controlid”. For instance to signal that ListBox1 has been updated, raise this event: “updated:ListBox1”.

DO: Use a standard verb as the action name.

The action or verb should be a common verb. Please refer to the PowerShell list of approved verbs: http://msdn.microsoft.com/en-us/library/windows/desktop/ms714428(v=vs.85).aspx.

It is very important to decouple controls. Instead use events or data-binding to synchronize controls (data-binding is implemented using events). A classic example is label and a textbox, where it is very tempting to make the label have a direct reference to the textbox for setting focus – it should not.

The Label control should have a TargetId property that contains the Id of the text box to focus. When the label is clicked, it should raise the event “focus”, e.g. “focus:” + TargetId. In turn the TextBox should listen for the focus event and set focus to it, when the event is raised.

Events are also used to communicate across application boundaries. In that case the event should be raised either on the application object or on the central Sitecore object.

6.3 JavaScript Pipelines

SPEAK supports JavaScript pipelines. The concept is the same as standard Sitecore pipelines - that it is a sequence of processing steps.

The pipelines do not use a configuration file, but is registered through the Sitecore.Pipelines.add method.

A processor is defined by creating a new property on a namespace object. When a pipeline is instantiated to be run, all the properties on the namespace object is collected and sorted by their priority property. The processors are then run one by one.

An empty pipeline processor looks like this:

```
var myProcessor =
  {
    priority: 1000,
    execute: function(context)
    {
      // execute
    }
  };

var myPipeline = new Sitecore.Pipelines.Pipeline("My");
myPipeline.add(processor);
Sitecore.Pipelines.add(myPipeline);
```

To start a pipeline, do this:

```
var context = {
  prop1: ‘a’,
  prop2: ‘b’
};

Sitecore.Pipelines.My.execute(context);
```

As in Sitecore, pipelines provide nice extensibility points and really help customer service making workarounds. It is highly recommended to use pipelines.

6.4 JavaScript Commands

Commands are an abstraction over user actions. By using commands, a function can be moved out of the Page Code file and reused.

Commands in SPEAK are modeled after WPF commands, having canExecute and execute methods. The canExecute method indicates if the command is available in the provided context, and surprise, the Execute method performs the action.

Commands are often page specific and are placed in a Commands file folder under the page folder. By placing commands in this folder, they are automatically included on the page by the PageCode rendering.

An empty command looks like this:

```
Sitecore.Commands.MyCommand =
{
  canExecute: function(context)
  {
    return true;
  },
  execute: function(context)
  {
    alert("Hello World");
  }
};
```

Commands are usually executed by setting the command property in a widget interaction to the fully qualified name of the command.

To explicitly execute a command, do this:

```
$.sc.executeCommand(“Sitecore.Commands.MyCommand”, context);
```

Commands has a tendency to simplify the code by moved it out of the Page Code file. It also supports the single responsibility pattern, in that a command performs a single action.

6.5 Resources

Examples of resources are translatable strings and settings from the web.config.

6.5.1 Translation

Translation of text is normally done through the standard Sitecore API when rendering a control on the server. However in some cases translatable text appear in JavaScript code.

To make translatable text available to JavaScript code, use the StringDictionary rendering. The StringDictionary rendering uses a data source that contains a Dictionary field.

The Dictionary field must be of field type TreeListEx. The items in this field are assumed to be items in the Code database underneath /sitecore/system/Dictionary.

For each item, the phrase is applied to the Sitecore.Resources.Dictionary object.

To use a text from the dictionary, do this:

javascript:alert(Sitecore.Resources.Dictionary[“TextKey”]);

6.5.2 Web.config settings

Occasionally it can be useful to make web.config settings available to JavaScript code.

The SettingsDictionary rendering can apply settings from web.config to the Sitecore.Resources.Settings object.

The SettingsDictionary rendering uses a data source that contains a Settings Multi-line Text field named Settings.

The Settings field contains a list of comma-separated names of web.config settings.

When the SettingsDictionary rendering is rendered, each setting is outputted to the Sitecore.Resources.Settings object.

To retrieve a setting, do this:

javascript: var setting = Sitecore.Resources.Settings[“settingName”];

Chapter 7

Server

In SPEAK the purpose of the server is to render the initial page and subsequently serve data to the client.

Avoid serving Html from the server in an Ajax request.

The server can be used for either querying data or performing remote procedure calls (RPC).

The Sitecore WebAPI is used to perform CRUD operations on items, and Sitecore MVC controllers and actions are used to support RPC.

7.1 Querying for data

Sitecore Item Web API provides a convenient way of performing Create, Read, Update and Delete operations on the server.

Please refer to the Sitecore Item Web API documentation for further information.

SPEAK contains a nice JavaScript API for working with Item Web API.

7.2 Remote Procedure Calls

RPCs are used when an action does not directly involve item operations, for instance publishing items.

SPEAK includes support for calling an MVC controller and action. When an interaction specifies the serverclick protocol, a request is made to the server.

The default MVC route for server clicks is:

```
/api/Sitecore/{controller}/{action}
```

The action must return a Json formatted result.

SPEAK provides a customized View result named SitecoreViewModelResult that adds additional support for SPEAK.

SitecoreViewModelResult has a dynamic property named ViewModel that allows automatic updates to the Sitecore.Page JavaScript object.

Using the ViewModel property, it is very easy to update the client Page object. For instance to set the text property of the SearchTextBox widget, simply write:

```
public JsonResult Clicked()
{
  var result = new SitecoreViewModelResult();

  result.ViewModel.SearchBox.text = "Server Says Hello";

  return result;
}
```

This of course comes at the cost of IntelliSense and type safety, but it is very nice for quickly prototyping a response.

It is important to stress that the ViewModel prototype is for prototyping only. It is recommended to use standard MVC techniques to output the result.

The MVC controller is a standard MVC controller. There are no additional requirements.

Chapter 8

Scripts and stylesheets

SPEAK contains a lot of JavaScript – so much in fact that it would be impractical load all of it upfront.

SPEAK instead uses Require.js to load scripts, but with a Sitecore Twist, of course.

The purpose of using Require.js is to reduce load time and provide code modularity. Require.js is also used to load Css files.

By default Require.js works by calling the require() function which takes a module to load and a callback that is called when the module has been loaded. Most modules are identified by their Url. For instance to load the SPEAK Core which is located in the Sitecore.js file, do this:

```
require([“sitecore”], function(Sitecore) {
})
```

Notice that the modules parameter is an array and that the modules are passed as parameters to the callback.

8.1 Page initialization

During page render the PageCode rendering inserts JavaScript and Css references into the page. These references are located under the SPEAK-Layout item (/Sitecore/client/SPEAK/Layouts/Layouts/SPEAK-Layout). This is done using the PageScripts and PageStylesheets pipelines.

If the page code finds a page code JavaScript file, either by convention or explicitly stated, it insert a script reference to it, setting the mime type to “text/x-sitecore-pagecode”. This will get loaded by the bootstrapper.

After these references, the Page Code inserts a reference to Require.js with an attribute ‘data-main’ that points to the main.js file – this approach is a Require.js convention. The main.js is located in ‘/sitecore/shell/client/SPEAK/assets/lib/dist/main.js’ and contains basic Require.js configuration.

Main.js also bootstraps the page using the following procedure:

1. Find all Html elements that have ‘data-sc-require’ attribute and store the references. for instance <button data-sc-require=”/-/SPEAK/v1/button.js”></button>.
2. Use Require.js to load all found references.
3. If any of the loaded references contains references, repeat 2.
4. If the Html contains a script that has the mimetype set to “text/x-sitecore-pagecode”.
    a. Load that reference. This effectively loads the page code script file.
    b. If the page code contains any references, load them.
    c. Instantiate the application in the page code and run it.
5. Otherwise
    a. Instantiate a generic application and run it.

When working with Require.js, it is important to know, that the script is not ready until the module has been loaded, which is indicated by the callback. As such most code is wrapped in calls to the require() function.

8.2 Resolving scripts

Modules in Require.js are normally identified by their Url. Require.js will use the module name to generate a Url (using a base Url) and load the script from it. This approach still works.

However this leads to hardcoded paths which may be difficult to maintain.

SPEAK implements a Sitecore Custom Handler for resolving script and stylesheet locations. The Custom Handler prefix is stored in web.config, and is currently set to “/-/SPEAK/v1/” – it can be accessed using the SPEAKSettings.Html.RequireJsCustomHandler property. Any Url that contains this string is routed to the SPEAK custom handler.

The custom handler uses the ‘SPEAK.client.resolveScript’ pipeline to find the appropriate script and stream it back. The script and stylesheet are cached pretty heavily using standard Http caching.

The important part of the pipeline is the Controls processor. The purpose of the Controls processor is to abstract the location of the scripts by flattening the folders. This allows developers to use any folder structure they wish, while using easy Urls in the script code.

In the web.config it is possible to configure a number of sources where the Controls processor will look for the script. By default the ‘asset’ source point to the “/sitecore/shell/client/SPEAK/assets” folder and the ‘controls’ source point to “/Sitecore/shell/client/SPEAK/Layouts/Renderings”.

Source may also contain a ‘category’ which functions as a namespace. This allows the developer to have multiple controls with the same name as long as they are in different sources with different categories. The ‘asset’ source use ‘asset’ as category and ‘controls’ use ‘controls’. Vendors should use their Company name as category.

Looking at the Url “/-~/SPEAK/v1/controls/button.js”, it contains the Custom Handler prefix “/-/SPEAK/v1/”, the category “controls” and the control name “button.js”. This resolves to /sitecore/shell/client/SPEAK/Layouts/Renderings/Common/Buttons/Button.js.

8.3 web.config changes

The Custom Handler requires two changes to the web.config which unfortunately cannot be put into the include file.

In <system.webServer>/<handlers> add

```
<add verb="*" path="sitecore_SPEAK.ashx" type="Sitecore.Resources.Scripts.ScriptHandler,
Sitecore.SPEAK.Client" name="Sitecore.SPEAK" />
```

In <system.web>/<httphandlers> add the same line:

```
<add verb="*" path="sitecore_SPEAK.ashx" type="Sitecore.Resources.Scripts.ScriptHandler,
Sitecore.SPEAK.Client" name="Sitecore.SPEAK" />
```

8.4 Adding script references in C# or MVC views

The ComponentBase class, which all controls inherit from, contains the property Requires which encapsulates functionality to output script references.

For instance the Button class requires the button.js script. The Button implementation contains the following call:

this.Requires.Script(“button.js”);

The same is possible for stylesheets.

8.5 Upgrading and caching

Traditionally upgrading has posed a problem because of script caching – often requiring end-users to press Ctrl+F5 to refresh the script cache.

In SPEAK it should be possible to overcome this issue by changing the Custom Handler prefix. It is therefore paramount that all code use the setting in the web.config.

8.6 Content Delivery Network (CDN)

All script reference are generated using the pipeline ‘’. As such it should be possible to store scripts on a CDN as any Url can be modified in the pipeline.

Chapter 9 Minifying and bundling

SPEAK implements a feature for auto-discovering new JavaScript and stylesheet files, if certain conventions are kept.

At appropriate points, SPEAK scans a number of folders for either JavaScript files or stylesheets. The found files are minified and bundled into a single file that is cached using a hash code. The page subsequently contains a single reference to this file.

The approach means that it is not necessary to add explicit references to new JavaScript files or stylesheets. Adding the files to a folder will automatically include the file on the page.

While developing it is desirable to scan the folders every time a page is requested, but in production it is probably sufficient to scans on system start up. Possibly the files can be placed on a Content Delivery Network.

The minifying and bundling processes are implemented in the “SPEAK.client.getScripts” and “SPEAK.client.getStylesheets” pipelines and as such can be overwritten.

The process can also be controlled by using the querystring parameters “minifyjs” and “minifycss”.

The Minify and Bundling process helps you in many ways. You save time by not having to maintain references to JavaScript and stylesheet files. Don’t be afraid of adding new files, since the overhead is very small. In turn this should enable you to practice the single-responsibility pattern – single file, single class, single responsibility.

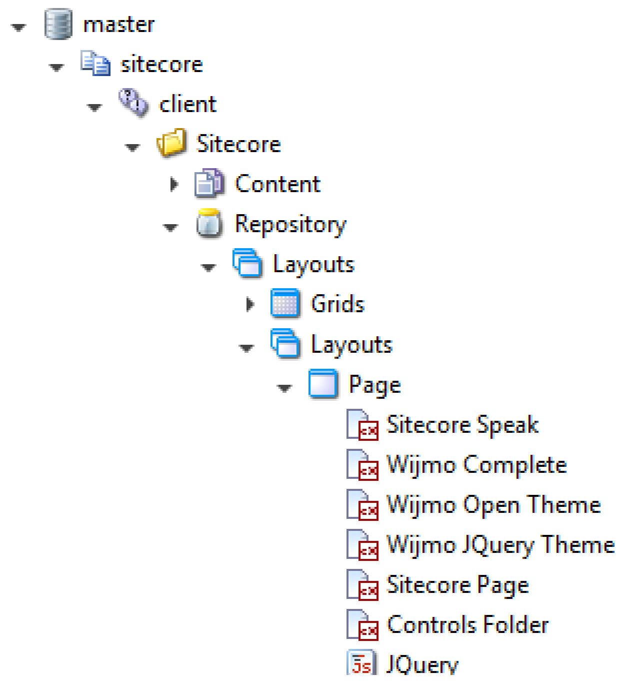
Chapter 10

CSS & LESS

10.1.1 Page-level scripts and stylesheets

Page-level scripts and stylesheets are system files that are present on all pages. An example of a system script is jQuery and Sitecore.Page.js.

Page-level scripts and stylesheets are represented by items in Sitecore. The items are located under /sitecore/client/Sitecore/Repository/Layouts/Layouts/Page.



To add a single page-level stylesheet or script, create a new item there and fill the 'File' field with the required file path. In case of a stylesheet, use 'Page Stylesheet file' template to create a new item. In case of a JavaScript file, use 'Page Script file' template to create a new item.

In order to add a bunch of scripts/stylesheets from the specified folder, create a new item from 'PageScriptFolder'/'Page Stylesheet Folder' template under

‘/sitecore/client/Sitecore/Repository/Layouts/Layouts/Page’ template and fill the ‘Folder’ field with the required folder path.

The folder is processed recursively. Thus files from nested folders are also included.

Scripts/Stylesheets files are included into a page according to the item order under ‘/sitecore/client/Sitecore/Repository/Layouts/Layouts/Page’.

By default, there are several predefined Page Script Folder/Page Stylesheet Folder items, pointing to SPEAK system folders. So:

There’s no need to create new Page Script item, if the script file is located under ‘/sitecore/shell/client/Sitecore/Repository/Layouts/Renderings’ or ‘/sitecore/shell/client/assets/lib’.

There’s no need to create new Page Stylesheet item, if the stylesheet file is located under ‘/sitecore/shell/client/Sitecore/Repository/Layouts/Renderings’.

This approach allows us more control over the page-level scripts which are now processed through the “SPEAK.client.getPageScripts” and “SPEAK.client.getPageStylesheets” pipelines.

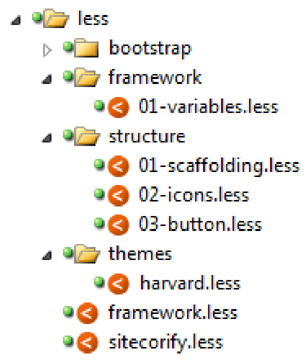
10.2 Less SDK

10.2.1 The structure

SPEAK does not provide currently any handlers to interpret less files directly, we have chosen the approach to provide you a less SDK in order to ease the creation of your style sheets.

This SDK is based on Twitter Bootstrap and the Sitecore Less Framework. It allows you to create custom themes to styles the components and generate the css files you need.

Here is the tree structure of the Framework:



```
less
    bootstrap
    framework
        01-variables.less
    structure
        01-scaffolding.less
        02-icons.less
        03-button.less
    themes
        harvard.less
    framework.less
    sitecorify.less
```

The Bootstrap folder contains the bootstraps less files.

The framework folder contains the files which will be used in all the other less files.

The structure folder contains the style that needs to be integrated in a Page in order to work properly.

The Themes is the place where you could create your own. To do so, create a new file and copy the content of the 'harvard.less' file in you newly created file and changes all the values you need. Now you just need the framework.less to import your new theme.

Example of a framework.less file which uses a custom theme:

```
@import "bootstrap/mixins.less";

@import "themes/harvard.less";
@import "themes/mynewtheme.less";
@import "framework/01-variables.less";
```

10.2.2 How to use

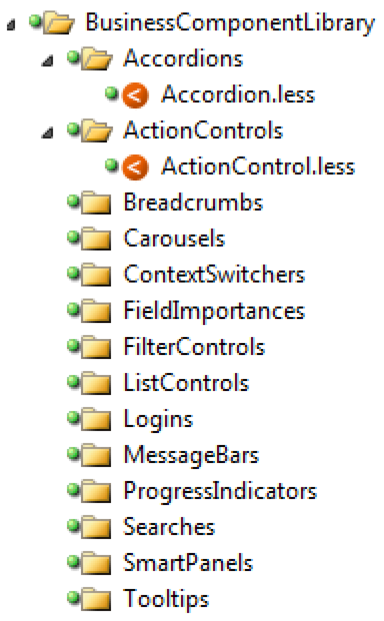
10.2.2..1 Create you less files

Create a folder and add some LESS files in it.

If you use the LESS SDK to generate the css, all the mixins, the utilities and variables from Bootstrap and Sitecore will be available in these files.

It means that you can use the theme’s variables in order to keep your component consistent and to change the way your components are styled only by changing the variable inside harvard.less (or your own theme).

Here is a folder which will use the LESS SDK to generate the style sheets:



```
BusinessComponentLibrary
    Accordions
        Accordion.less
    ActionControls
        ActionControl.less
    Breadcrumbs
    Carousels
    ContextSwitchers
    FieldImportances
    FilterControls
    ListControls
    Logins
    MessageBars
    ProgressIndicators
    Searches
    SmartPanels
    Tooltips
```

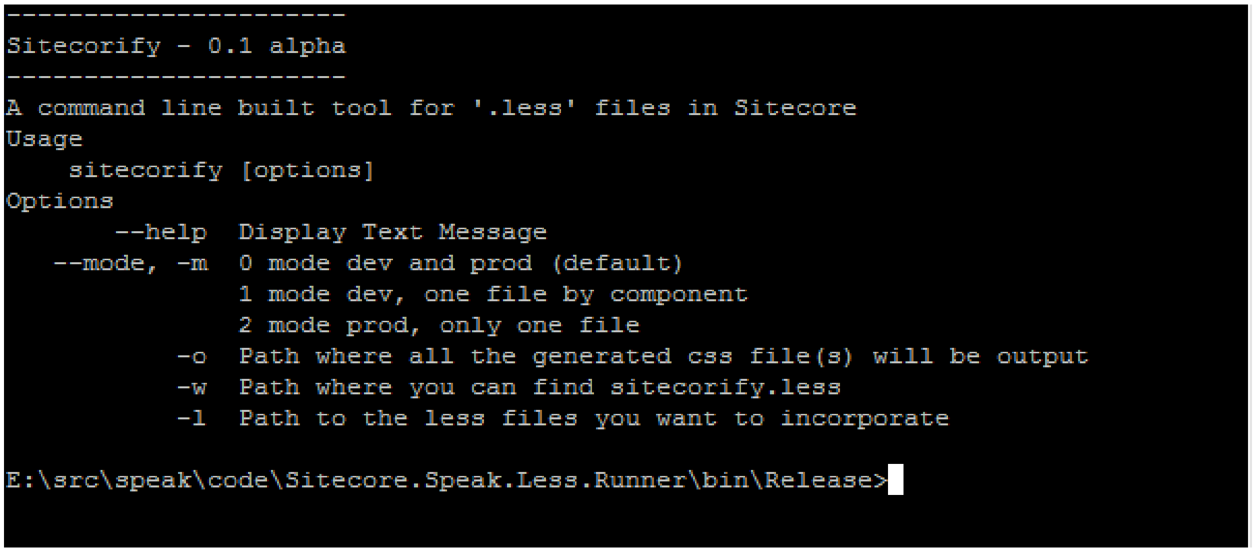
Example of a less file which uses a mixin from bootstrap and a color from the Sitecore Less Framework:

```
.myclass {
    color: @SC-THEME-neutral-dark; /* from Sitecore Framework */
    .border-radius(); /* from bootstrap */
}
```

As you can see, you can use this without adding anything in your environment.

10.2.2..2 Run sitecorify.exe

Now that you have less files created, you can use the application sitecorify.exe in order to compile them in css. Use “sitecore.exe -- help” to see how you can use it.



```
----------------------
Sitecorify - 0.1 alpha
----------------------
A command line built tool for '.less' files in Sitecore
Usage
    sitecorify [options]
Options
       --help  Display Text Message
   --mode, -m  0 mode dev and prod (default)
               1 mode dev, one file by component
               2 mode prod, only one file
           -o  Path where all the generated css file(s) will be output
           -w  Path where you can find sitecorify.less
           -l  Path to the less files you want to incorporate

E:\src\speak\code\Sitecore.Speak.Less.Runner\bin\Release>
```

Sitecorify.exe supports 2 modes, the production mode which will compile all your less files and integrate them with Boostrap and the Sitecore framework into one final minified css file and the development mode which will generate a single css file for each file you have.

By default, it will run both modes.

3 Parameters are required.

- The path where to find the Sitecore Framework (-w)
- The path where to find your less files you want to integrate (in production mode) or to compile to css (in dev mode).
- The path where to generate the css (-o)

Chapter 11

Local Storage

In order to make persisting data of controls safe SPEAK provides local storage API. It uses its own key prefix for saving data items. By default it's equal "#sc#" – it means, that all Sitecore local storage items will be saved with this prefix and will not intersect with other client-data stored in local storage.

It is strongly recommended to use Sitecore Local Storage API instead of using it directly through window.localStorage.

Each control which has the persisting logic should have methods implementing this logic calling Sitecore Local Storage API. But realization of this logic should also take into consideration, that another instance of this control can save its own state, so it's strongly recommended to use complex unique keys. For instance Sitecore pageID can be used to store page-specific data.

Real-life scenario – it needs to save and restore some grid data with some ID:

```
Sitecore.LocalStorage.setItem(Sitecore.Models.PageInfo + "#" + grid.ID, { column1: true,
column2: false });
var gridData = Sitecore.LocalStorage.getItem(Sitecore.Models.PageInfo + "#" + grid.ID);
```

Sitecore Local Storage API is accessible through the Sitecore.LocalStorage object, the list of available methods:

- `getItem(key [, useSessionStorage])`

  returns deserialized object, array or string; useSessionStorage – specifies that sessionStorage will be used instead of localStorage. If this parameter isn't specified localStorage will be used by default.

- `setItem(key, value[, useSessionStorage])`

  saves the specified object, array or string in storage sprecified by value. The object or array can be passed by reference, it will be serialized and saved to localStorage or sessionStorage if `useSessionStorage` parameter is `true`

- `removeItem(key [, useSessionStorage])` removes item specified by key from localStorage. Please take into account, this method removes only item with Sitecore-prefixed key.

- `getItemPerPage/setItemPerPage/removeItemPerPage –`

  These methods do the same operations as getItem/setItem/removeItem but with the only difference of adding Sitecore.Models.PageInfo["pageID"] to the key-prefix. These methods should be used when you want to save some data per page, for example some visual options like grid sorting type.

- `clear([useSessionStorage])`

  clears the localStorage or sessionStorage items.

- `onStorageChanged(handler)`

  set the event handler to the storage event. It doesn't work properly in all browsers.

- `getAllKeys()`

  returns Array of all keys from localStorage

- `getSCKeys()`
  returns Array of only Sitecore-prefixed keys from localStorage

Please take into consideration that the size of localStorage is limited(up to 5 mb). If you commit the saving data and the size exceeds new jQuery event will be triggered. You can bind to this event via:

```
$(window).bind("Sitecore.LocaStorage.quotaExceededError",function(e){/*your code*/});
```

Ideally it can be some kind of notification dialog with link to empty the localStorage.

Chapter 12

Content

In SPEAK the content item are organized in a different way than the traditional Sitecore way.

SPEAK tries to place logically connected items close together. For instance the template for rendering parameters is located underneath the rendering item.

Ideally to delete a single feature (like a component), you should only have to delete single item and a single folder.

12.1 Root item

The entry point to SPEAK is the /sitecore/client item. Everything related to SPEAK must be located underneath this node.

The client item may be located in an arbitrary database, e.g. the Master or the Core database. The important aspect is that the SPEAK items are isolated from the rest of the system.

12.2 Vendor items

Underneath the root item are a number of vendor items.

You can think of vendor items as namespaces. In .NET the company name is usually used as a root namespace.

Each vendor that integrates with SPEAK must include a vendor item, and Sitecore also has a vendor item.

The Sitecore root for SPEAK is therefore:

/sitecore/client/Sitecore

12.3 Content item

Underneath the vendor item is a content item which holds the applications and pages. This is analogue to the traditional content node in Sitecore.

/sitecore/client/Sitecore/content

12.4 Application items

The content item usually holds a number of applications, e.g. ECM or Order Management.

/sitecore/client/Sitecore/content/ECM
/sitecore/client/Sitecore/content/OrderManagement

Under the application item is a structure that is similar to standard Sitecore.

/sitecore/client/Sitecore/content/ECM/content
/sitecore/client/Sitecore/content/ECM/media library
/sitecore/client/Sitecore/content/ECM/layouts
/sitecore/client/Sitecore/content/ECM/system
/sitecore/client/Sitecore/content/ECM/templates

12.5 Page items

Under that application items are the page items.

/sitecore/client/Sitecore/content/ECM/content/LandingPageDesigner

Each page has a PageSettings item that holds configuration items specific to that page. The PageSettings items must have the PageSettings template to distinguish it from subpages.

/sitecore/client/Sitecore/content/ECM/content/LandingPageDesigner/PageSettings

12.6 Renderings

Renderings are usually placed in the layouts as in standard Sitecore. The rendering folder should have the Speak-RenderingFolder template.

/sitecore/client/Sitecore/Layouts/Renderings/TextBox

The template for the renderings parameters are placed underneath the rendering item.

/sitecore/client/Sitecore/Layouts/Renderings/TextBox/TextBoxParameters

Chapter 13

Files

As with content items, the SPEAK files is also isolated from the rest of the system.

13.1 Root folder

SPEAK uses the root folder

\sitecore\shell\client

SPEAK contains a preprocess Http request handler that maps this folder to the shell site, so SPEAK can use the shell site.

13.2 Vendor folders

Underneath the root folder is a number of vendor folders.

The Sitecore root folder for SPEAK is therefore:

\sitecore\shell\client\Sitecore

13.3 Content folder

The vendor folder contains a Content folder that holds the files for each page.

\sitecore\shell\client\Sitecore\Content

Chapter 14 SDK

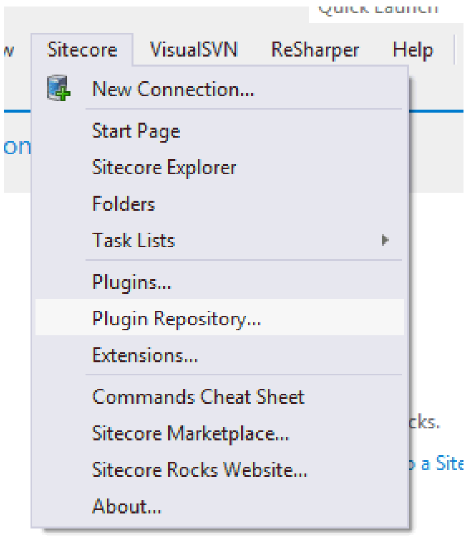
The SDK contains samples, tools, and documentation.

14.1 Sitecore Rocks Integration

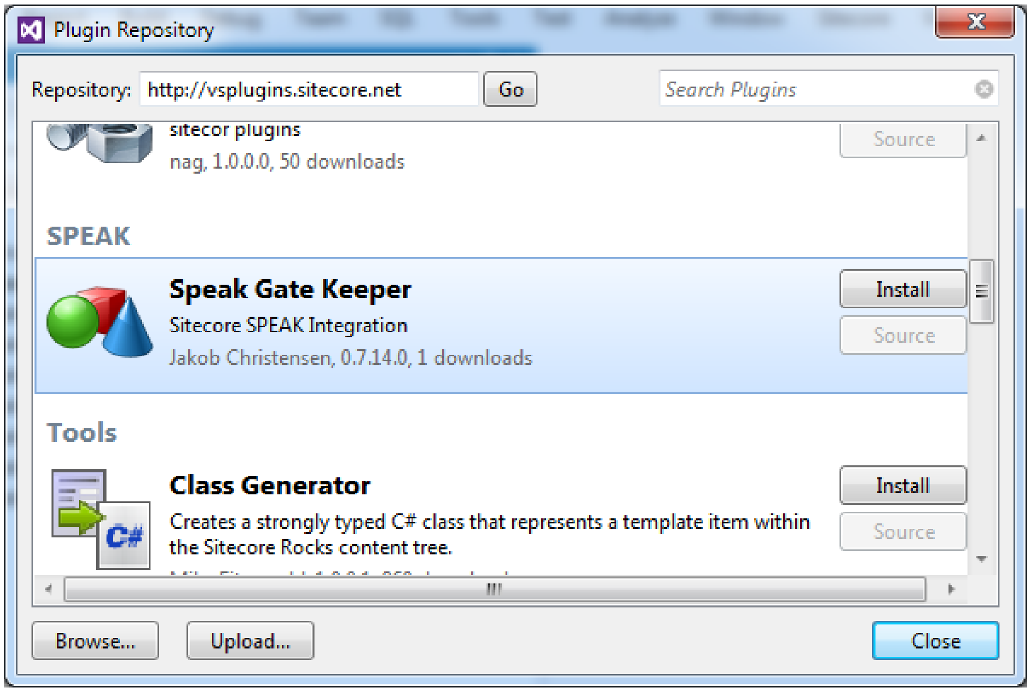
The SPEAK SDK contains a Sitecore Rocks plugin that contains various integrations and tools.

14.1.1 Installation

Open the Plugin Repository in the Visual Studio Sitecore main menu.



In the Plugin Repository dialog, find the SPEAK Gate Keeper, and click the Install button.


Plugin Repository
Repository: http://vsplugins.sitecore.net
Go
Search Plugins
sitecor plugins
nag, 1.0.0.0, 50 downloads
Source
SPEAK
Speak Gate Keeper
Sitecore SPEAK Integration
Jakob Christensen, 0.7.14.0, 1 downloads
Install
Source
Tools
Class Generator
Creates a strongly typed C# class that represents a template item within the Sitecore Rocks content tree.
Install
Source
Browse...
Upload...
Close

The plugin contains a server component, so remember to update server components.

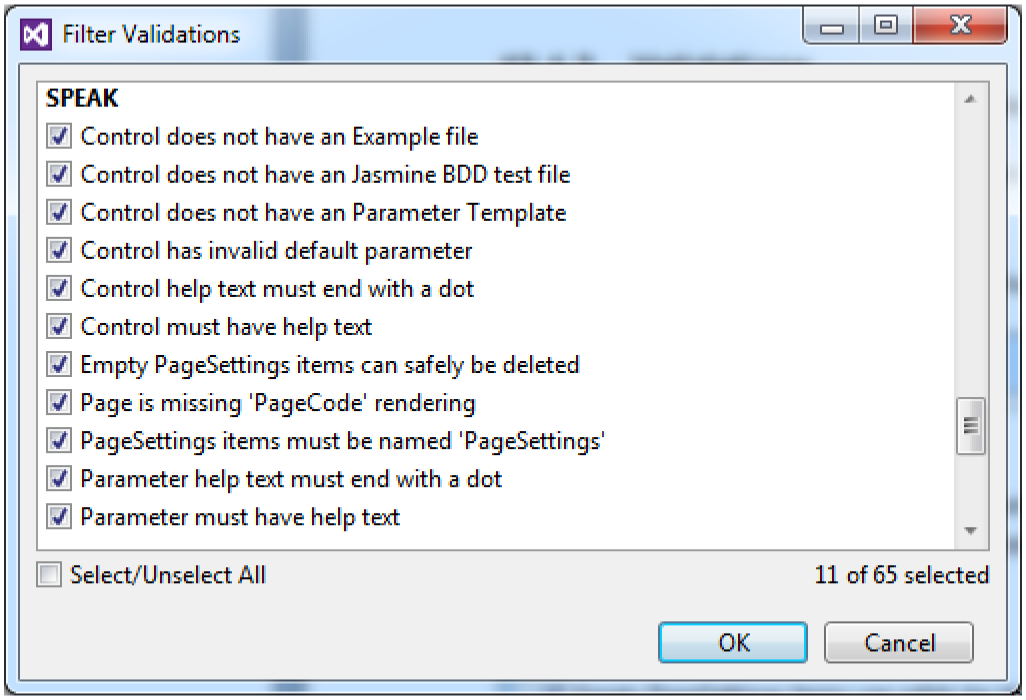
14.1.2 Using the Sitecore Rocks plugin

14.1.3 Validations

The plugin integrates into the Sitecore Rocks validation framework to provide SPEAK specific validations.

To run the validations, open the Validations screen by right-clicking a site in the Sitecore Explorer and selecting Tools | Validations.

Next click the Filter Validations button in the toolbar to setup the SPEAK validations.



Filter Validations

SPEAK

- [✓] Control does not have an Example file
- [✓] Control does not have an Jasmine BDD test file
- [✓] Control does not have an Parameter Template
- [✓] Control has invalid default parameter
- [✓] Control help text must end with a dot
- [✓] Control must have help text
- [✓] Empty PageSettings items can safely be deleted
- [✓] Page is missing 'PageCode' rendering
- [✓] PageSettings items must be named 'PageSettings'
- [✓] Parameter help text must end with a dot
- [✓] Parameter must have help text

[ ] Select/Unselect All — 11 of 65 selected

When the dialog closes that validations are run and an assessment report is shown.

Most of the validations have Fix buttons, so fixing the issues is often easy.

Chapter 15

Component Library

15.1 Buttons

Control	Description
Button	Standard button
SplitButton	Standard split button
DropDownButton	Standard drop down button

15.2 Data display

Control	Description
ListView	Standard List View
DataGrid	Advanced data grid
TreeView	Standard tree view
ChildRenderer	Renders the children of an item
ItemRenderer	Renders an item

15.3 Date display and selection

Control	Description
Calendar	Calendar
DatePicker	Date/time picker

15.4 Layout

Control	Description
Border	Simple div tag
Expander	Standard expand/collapse
Splitter	Grid splitter
ScrollViewer	Simple div tag with scrollbars

15.5 Menus

Control	Description
NavigationBar	Top menu
ContextMenu	Standard context menu

15.6 Selection

Control	Description
CheckBox	Standard check box
RadioButton	Standard radio button
ComboBox	Standard combo box
ListBox	Standard list box

15.7 Navigation

Control	Description
Hyperlink	Standard hyperlink
TabControl	Standard tab strip
Frame	IFrame

15.8 Text

Control	Description
Text	A simple, translatable text
Label	Standard label
ProgressBar	Standard progress bar
BusyIndicator	Spinner

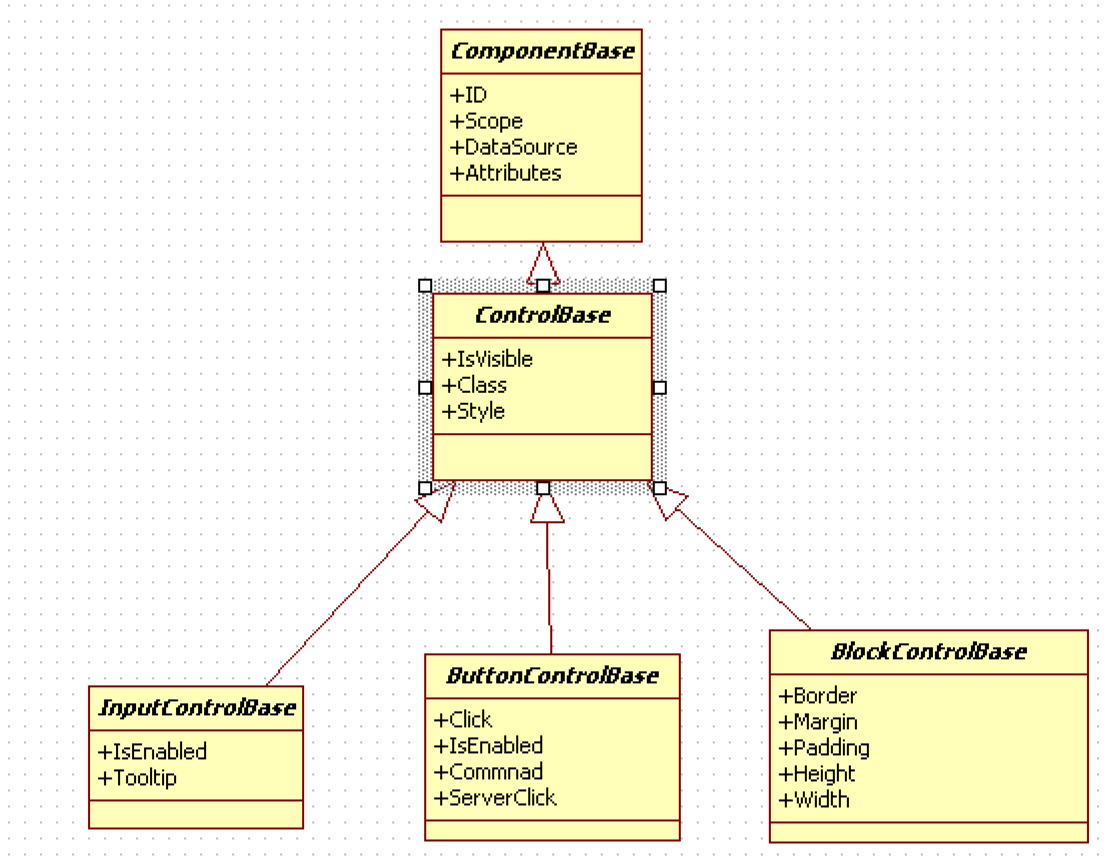
15.9 Media

Control	Description
Image	Standard image
Icon	Standard icon

15.10 SPEAK

Control	Description
Accordion	
ActionControl	
Breadcrumb	
Carousel	
ContextSwitcher	
Dialogs	
FieldImportance	
FilterControl	
ListControl	
Login	
ProgressIndicator	
Report	
Search	
SmartPanel	
TabControl	
ToolTips	
MessageBar	

15.11 Abstract base controls



15.12 Controls

15.12.1 Component

Component is the base element for all visual and non-visual elements.

Inheritance hierarchy

None

Members

Id	string	Id of the control	

15.12.2 Control

Control is the base element for all visual elements.

Inheritance hierarchy

Component

Model Members

isVisible	bool	Visibility state	In
toggle()		Toggles the visibility state	

15.12.3 Input

Input is the base element for all elements that handle user input like textboxes.

Inheritance hierarchy

VisualControl

Members

isEnabled	bool	Enabled state	In

15.12.4 Panel

Panel is the base element for all elements that provides block layouts.

Inheritance hierarchy

VisualControl

Members

border	string	CSS border property	In
padding	string	CSS padding property	In
margin	string	CSS margin property	In

15.12.5 ButtonControl

Button is the base element for all elements that handles clicks.

Inheritance hierarchy

VisualControl

Members

text	string	Text	In
click	string	Local click handler	
command	string	Local command	
serverclick	string	Server click handler	

15.12.6 Button

Standard button

Inheritance hierarchy

ButtonControl

Members

15.12.7 SplitButton

Standard split button.

Inheritance hierarchy

ButtonControl

Members

menu	string	Menu Id	

15.12.8 DropDownButton

Standard drop down button

Inheritance hierarchy

ButtonControl

Members

Menu	String	Menu Id	

15.12.9 ListView
Standard list view

Inheritance hierarchy

Input, Panel

Members

items	array	Items in the list view	In
selectedItems	array	Selected items	Out
selectedItemId	string	Currently selected item	Out
IsMultiSelect	bool	Is multi selected allowed	Design

15.12.10 DataGrid
Standard data grid

Inheritance hierarchy

Input, Panel

Members

items	array	Items in the list view	In
selectedItems	array	Selected items	In/Out
selectedItemId	string	Currently selected item	In/Out
IsMultiSelect	bool	Is multi selected allowed	Design

15.12.11 TreeView
Standard tree view

Inheritance hierarchy

Input, Panel

Members

selectedItems	array	Selected items	In/Out
selectedItemId	string	Currently selected item	In/Out
IsMultiSelect	bool	Is multi selected allowed	Design

15.12.12 Calendar
Input, Panel

Inheritance hierarchy

Panel

Members

selectedDatetime	dateTime	Selected date time	In/Out

15.12.13 DatePicker
Button with a calendar drop down

Inheritance hierarchy
Input

Members

selectedDateTime	dateTime	Selected date time	In/Out

15.12.14 Border
Simple div tag

Inheritance hierarchy
Panel

Members

15.12.15 Expander
Standard expand/collapse

Inheritance hierarchy
VisualControl

Members

isExpanded	bool	Expanded state	In/Out
text	String	Header	In
toggleExpanded		Toggles the expanded state	

15.12.16 ScrollViewer
Standard scrollbox

Inheritance hierarchy
Panel

Members

15.12.17 ScrollViewer
Standard scrollbox

Inheritance hierarchy

Panel

Members

15.12.18 NavigatiomBar

Top navigation bar.

TBD

15.12.19 Context Menu

Standard context menu that appears on right-click

TBD

15.12.20 CheckBox

Standard checkbox

Inheritance hierarchy

Input

Members

isChecked	bool	Checked state	In/Out
text	String	Header	In
toggleChecked		Toggles the checked state	

15.12.21 CheckBox

Standard checkbox

Inheritance hierarchy

Input

Members

isChecked	bool	Checked state	In/Out
text	String	Header	In
toggleChecked		Toggles the checked state	

15.12.22 RadioButton

Standard radio button

Inheritance hierarchy

Input

Members

isChecked	bool	Checked state	In/Out
text	String	Header	In
toggleChecked		Toggles the checked state	

15.12.23 ComboBox

Standard combo box

Inheritance hierarchy

Input

Members

items	Array	List of items in the drop down	In/Out
selectedItem	object	Selected item	In/Out
selectedItemId	string	The Item ID of the selected item	In/Out

Data source

The child items of the data source are the items in the drop down.

15.12.24 ListBox

Standard list box

Inheritance hierarchy

Input, Panel

Members

items	Array	List of items in the drop down	In/Out
selectedItem	object	Selected item	In/Out
selectedItemId	string	The Item ID of the selected item	In/Out

Data source

The child items of the data source are the items in the drop down.

15.12.25 HyperLink

Standard hyper link

Inheritance hierarchy

ButtonControl

Members

15.12.26 TabControl

Standard tab strip

Inheritance hierarchy

Panel

Members

tabs	Array	List of tabs	In/Out
selectedTab	object	Selected tab	In/Out

Data source

The children of the data source are the tabs. Each tab item is rendered using an Item Renderer.

15.12.27 Frame

Standard IFrame

Inheritance hierarchy

Panel

Members

Source	string	HRef	In

15.12.28 Text

Text control

Inheritance hierarchy

VisualControl

Members

text	strting	The text	In/Out

15.12.29 Label

Label control

Inheritance hierarchy

VisualControl

Members

text	strting	The text	In/Out
target	string	The id of the control to focus	In

15.12.30 ProgressBar

Standard progress bar

Inheritance hierarchy

VisualControl

Members

Value	double	The value of the progress bar.	In/Out
MinValue	double	The min value of the progress bar.	In/Out
MaxValue	double	The max value of the progress bar.	In/Out
ShowLabel	bool	Indicates whether progress label is visible or not.	In/Out
LabelFormatString	string	Sets the format of the label text.	In

15.12.31 BusyIndicator

Control for showing that an operator is running

Inheritance hierarchy

VisualControl

Members

isBusy	bool	Indicates if the busy indicator is visible and running	In/Out
TargetControl	string	The ID of control, which initiated the long running operation. Keep blank to show busy indicator over the entire page.	In/Out
AutoShow	bool	Indicates if busy indicator should be shown automatically, when AJAX request takes more than a certain amount of time.	In
AutoShowTimeout	int	Timespan in milliseconds. If AJAX request exceeds the timespan, BusyIndicator will be shown. Makes sense when 'AutoShow' is set to true.	In

15.12.32 Image

Standard image

Inheritance hierarchy

VisualControl

Members

Source	string	Src attribute	In/Out
Width	string	Width	
Height	string	Height	
Alt	string	Alt text	

15.12.33 IconButton

Standard icon that can be clicked.

Inheritance hierarchy

ClickableControl

Members

Source	string	Sitecore Icon path	In/Out
Dimension	string	“16x16”, “24x24”, “32x32” or “48x48”	

15.13 Data Controls

15.13.1 ItemDataSourceBase

Base class for datasource controls, that expose a set of items as a datasource.

Inheritance hierarchy

ComponentBase

Members

language	string	The language name. Keep blank to use context language.	In
database	string	The database name. Keep blank to use context database.	In
fields	array	The array of field names or IDs which should be retrieved for each item. If set to null all fields will be retrieved. Specify empty array, in order not to retrieve item fields.	In
Items	array	Resulting items array	out

15.13.2 QueryDataSource

QueryDataSource control enables retrieving Sitecore items specified by Sitecore Query or Sitecore Fast query.

QueryDataSource supports server-side pagination.

Inheritance hierarchy

ItemDataSourceBase

Members

Query	string	Sitecore query or fast query	In
PageSize	int	The size of the page. Specify page size equals 0, to disable paging	In
PageIndex	int	0-based page index. Doesn’t make sense if pageSize equals 0.	In
PageCount	int	The number of pages, which is calculated according to PageSize	out
TotalItemsCount	int	The total number of items. Pagination doesn’t take	out

		influence on it.	

15.13.3 SearchDataSource

SearchDataSource control enables retrieving Sitecore items specified by search query.

QueryDataSource supports server-side pagination.

Inheritance hierarchy

ItemDataSourceBase

Members

searchText	string	The search query	In
pageSize	int	The size of the page. Specify page size equals 0, to disable paging	In
pageIndex	int	0-based page index. Doesn't make sense if pageSize equals 0.	In
pageCount	int	The number of pages, which is calculated according to PageSize	out
totalItemsCount	int	The total number of items. Pagination doesn't take influence on it.	out
searchType	string	The searchtype, e.g. 'AllItems', 'MediaItems' etc.	in
facets	array	The list of available facet search filters.	out
selectedFacets	array	The list of selected facet search filters.	in
facetsRootItemId	string	The id of the item, containing available search facets as children.	Design
rootItemId	string	The id of the item, where searching should start from.	In/out
searchTypeDisplayText	string	The description of selected search type.	out