Advanced Sitefinity Development

About This Course

Course goals

- By the end of this course, you should be able to:
  - Develop the presentation layer of Sitefinity using Sitefinity Feather
  - Connect the presentation layer with the data layer using different Sitefinity APIs
  - Model the data layer in Sitefinity
  - Integrate external content and integrate with external systems
  - Perform advanced tasks such as optimizing your application performance, testing your code, etc.
  - Build Applications using Sitefinity

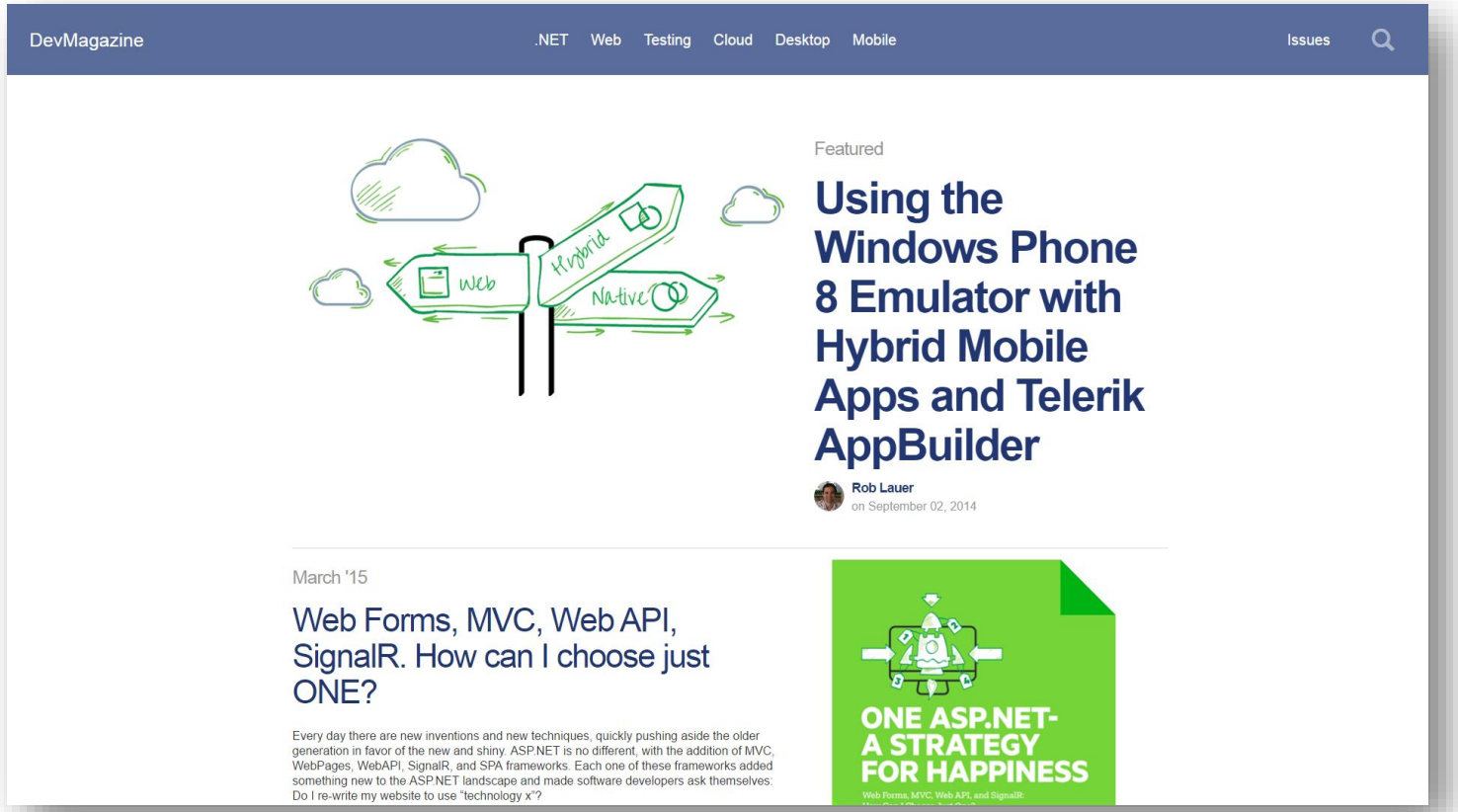
Audience and prerequisites

- The intended audience for this course are back-end developers who need to develop a web application using Sitefinity
- Before taking this course, students should have:
  - Passed the *Basic Sitefinity Developer Certification Exam*
  - Strong experience developing ASP.NET applications
  - A working knowledge of ASP.NET MVC
  - Strong experience with HTML and JavaScript
  - A working knowledge of AngularJS
  - Experience with relational databases

Course overview

- Day 1 – Laying the Foundation
  - About This Course
  - Lesson 1: Brief Review of the Sitefinity Features
  - Lesson 2: Developing the Presentation Layer
  - Lesson 3: The Widget Designer Framework
- Day 2 – Taking it to the Next Level
  - Lesson 4: Bringing Content to the Presentation Layer Using APIs
  - Lesson 5: Using Providers to Connect to Different Data Sources
  - Lesson 6: Localization of Content
  - Lesson 7: Working with Events
- Day 3 – Advanced Topics
  - Lesson 8: Optimizing the Performance of Your Sitefinity Application
  - Lesson 9: Managing Sitefinity Configurations
  - Lesson 10: Testing Your Code

Business case: The DevMagazine site



Let's see a demo of the site!
http://devmagazine.cloudapp.net/

Exercise Setup Requirements

Before starting class, you are expected to have:

1. Checked if you have the required software installed on your computer
   - Windows 7
   - IIS 7 or higher
   - Microsoft Visual Studio 2012 or higher
   - .NET Framework 4.5 or higher
   - Microsoft SQL Server or Microsoft SQL Server Express
2. Installed the latest version of Sitefinity.
3. Configured IIS to host Sitefinity projects.
4. Created a Sitefinity project named **DevMagazine.**
5. Installed Sitefinity's NuGet packages.
6. Optimized the project for faster startup.
7. Deployed the Sitefinity project on IIS.

Training Conventions

Tip

Exercise

Demo

Question

Best Practice

Code example

Important

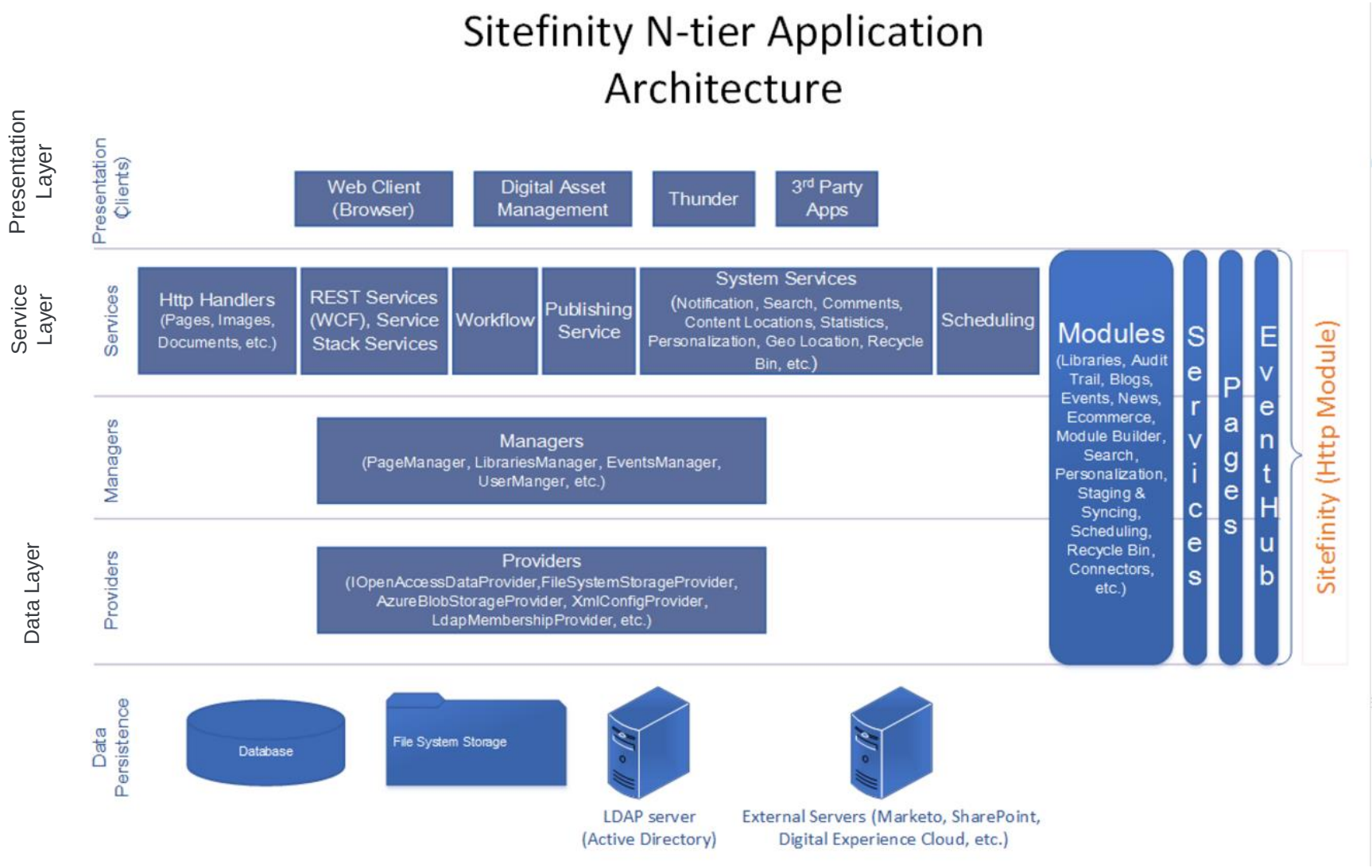
Gist on GitHub

Lesson 1: Brief Review of the Sitefinity Features

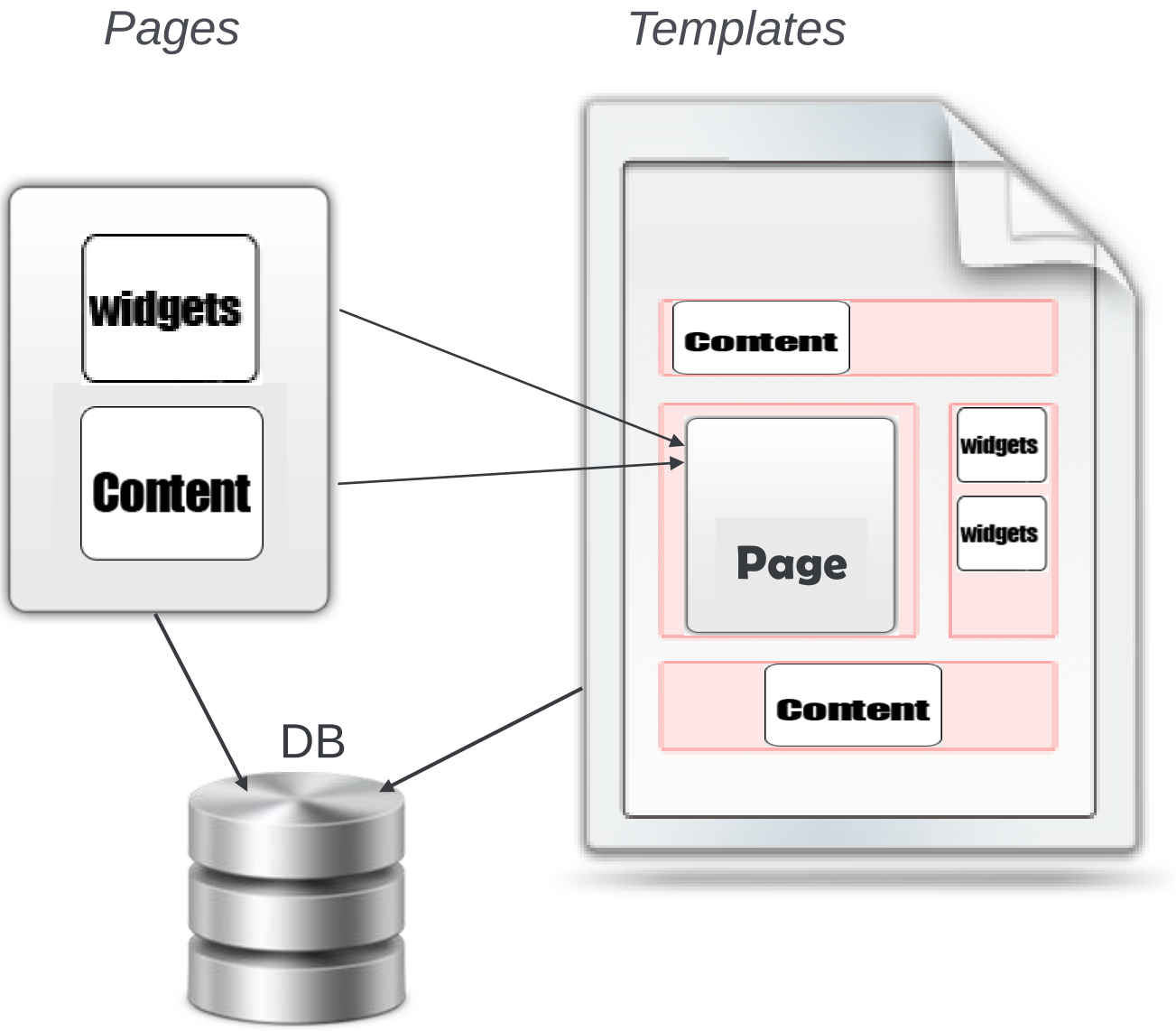
Lesson objectives

- By the end of this lesson, you should be able to:
  - Recall the main architectural layers of Sitefinity.
  - Recall the main components of the presentation layer.
  - Recall key Sitefinity features.

Sitefinity application architecture



Presentation layer architecture



Introducing Sitefinity Feather | The framework for the presentation layer

Sitefinity Feather introduces a modern, convention-based, mobile-first UI framework for the Sitefinity CMS.

For a full overview, check out: http://projectfeather.sitefinity.com/

- *Core infrastructure*
- *MVC Stock Widgets*
- *Front-End of Your Choice*
- *Mobile First*
- *Convention-Based Framework*
- *New Designers Framework*

Feather is Open Source!

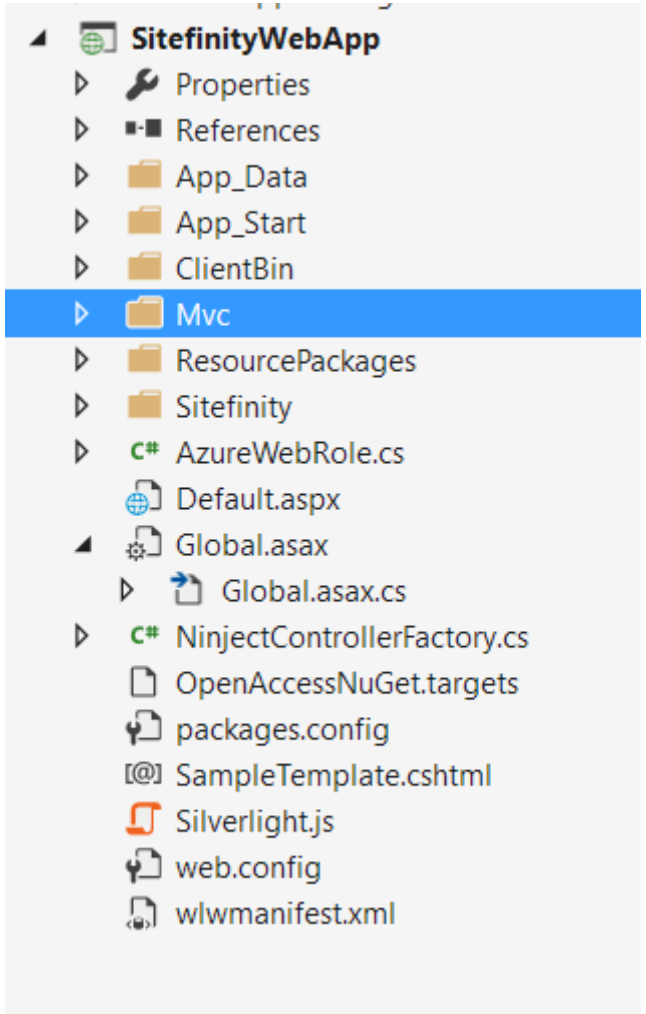
Demonstration: Walkthrough of Sitefinity features

- Content management
- Pages management
- Page templates management
- Page templates, pages and content – how they fit in the *big picture*
- Sitefinity front-end and back-end
- Module Builder
- Related Data

Overview of tasks to develop a Sitefinity website

- Develop the presentation layer
  - Surface data from data layer.
- Build the data layer
  - Create or customize models to hold data.
- Write the business logic
  - Mostly extensions to existing subsystems or separate components.

The Project Structure



Lesson 2: Developing the Presentation Layer

Lesson objectives

- By the end of this lesson, you should be able to:
  - Describe the components of the presentation layer.
  - Create a razor layout view – the Page Template.
  - Add grids to your page template.
  - Develop custom widgets.
  - Extend existing widgets.

High-level development tasks

Presentation Layer

- Surface data from data layer in some way.

Data Layer

- Create or customize models to hold data.

Business Logic

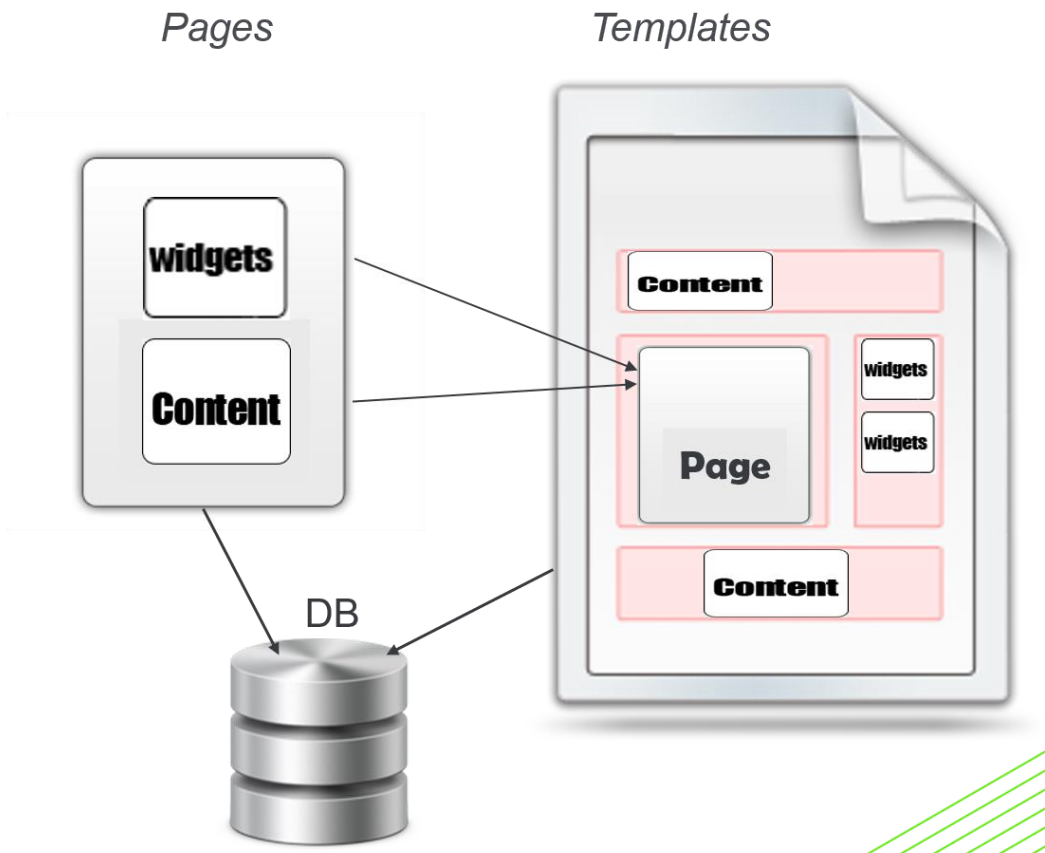
- Mostly extensions to existing subsystems or separate components.

Components of the presentation layer

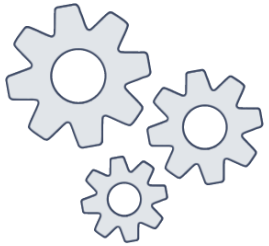
- Pages (in DB)
  - Based on Page Templates (in DB)
    - Based on physical layout files / master pages

Contain

- Widgets (MVC Controllers or ASP.NET Controls)
  - Interface with all other application layers



Exercise: Create a simple Sitefinity page



1. Open Sitefinity's back-end administration portal.
2. Create a page and add a Content Block.
3. Enter any static HTML (or some text) inside the widget editor.
4. Publish the page.
5. View the published page.

Sitefinity pages are modularized—Sitefinity uses many components to assemble a page

Steps to develop the presentation layer

1. Obtain HTML from the front-end developer.
2. Create a razor layout view
   - *Sitefinity will automatically create a page template based on it*
3. Optionally, refine your page template by adding grid widgets.
4. Create custom grid widgets, if needed.
5. Determine widget requirements.
6. Develop custom widgets, if needed.
7. Extend existing widgets, as required.

>> Identify Sitefinity widgets that can be reused. The more the better

Always look for existing functionality and don't reinvent the wheel. DO NOT REPEAT YOURSELF

Step 1: Obtaining HTML from front-end developer

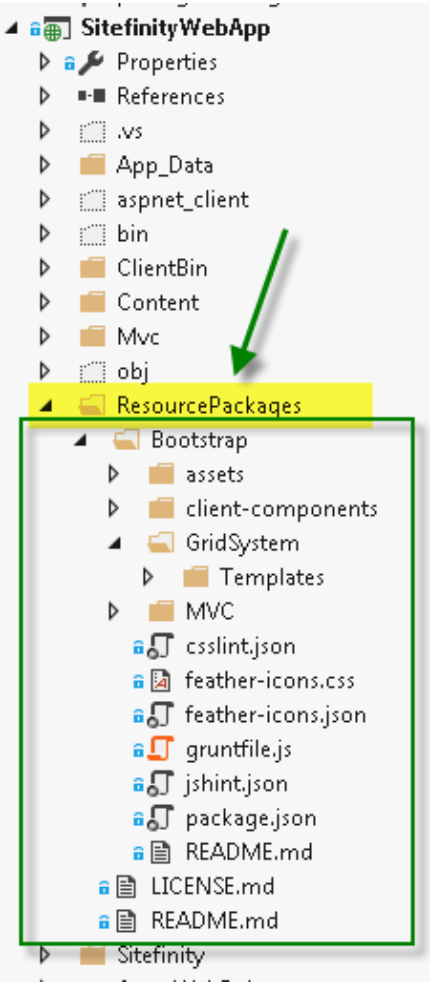
- Front-end developers produce a resource package consisting of:
  - Sample HTML code based on mockups of the website's pages
  - CSS
  - Images
  - Icons, etc

Front-end resource packages: The essence

A Resource Package is a complete encapsulation of the look and feel in Sitefinity

Within each resource package you can manage:

- *The CSS files and it source files - Sass, LESS and etc.*
- *JavaScript files*
- *Page Template files – MVC layout files*
- *Stock and Custom widget templates – Views*
- *Sitefinity Layout widgets templates*
- *Static resources used by the CSS – Images, Icons, Fonts, etc.*



Front-end resource packages: A look at the structure

Location - All resource packages should be placed in ~/ResourcePackages folder. Each folder is regarded as a separate package.

Context - If a request is received in the context of a package, Sitefinity checks for a corresponding file in the resource package folder.

Let’s see a demo of the resource packages folder structure!

Step 2: Creating a razor layout view

1. Create a razor layout view file following this convention:
   `/ResourcePackages/[ResourcePackageName]/Mvc/Views/Layouts/[ViewName].cshtml`

2. Copy the sample HTML code (received from the front-end Developer) into your layout file.
   - Sitefinity will automatically create a **page template** based on this layout file.

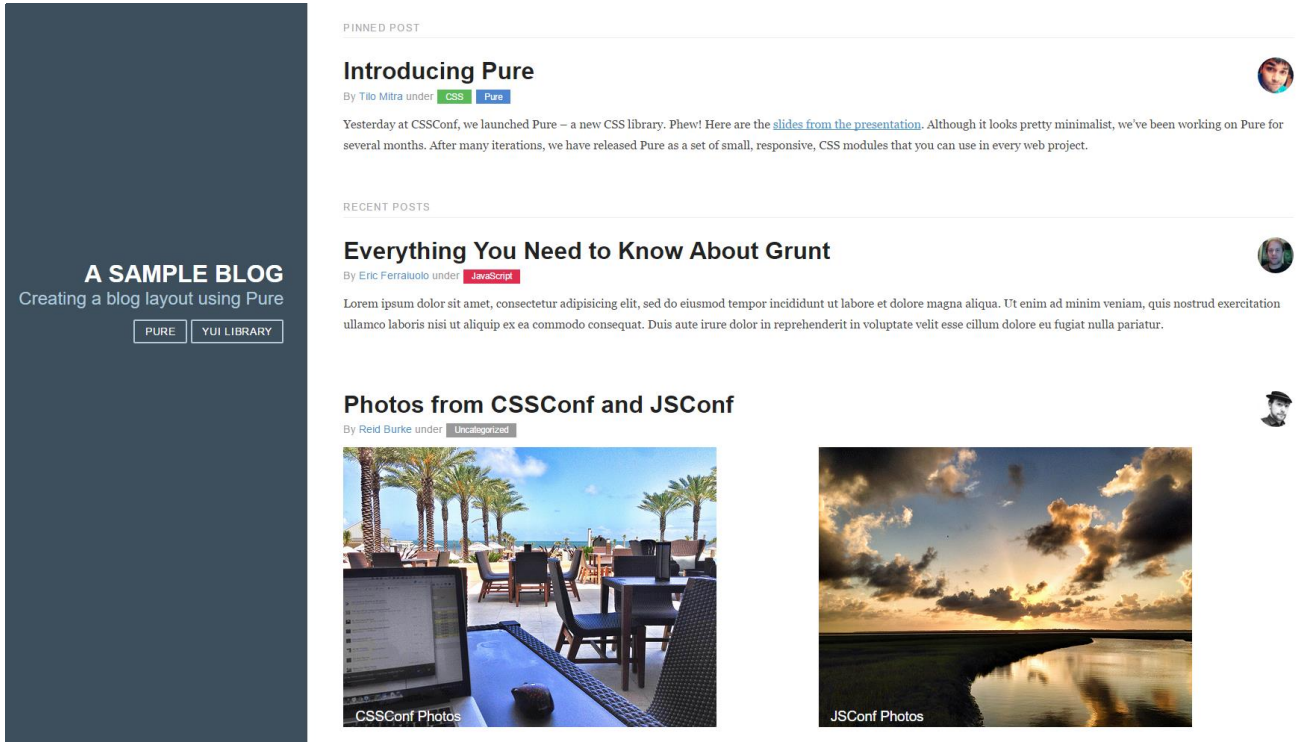
3. Open the page template in Sitefinity and identify what HTML elements need to be replaced by widgets.

4. Create placeholders for those elements. Note that:
   - Placeholder IDs are used as widget container IDs in the database, so DO NOT change the IDs. If you do change these, widgets will need to be migrated from one placeholder to the other.

Avoid having hardcoded resources strings/images in the master page that need to be managed/localized

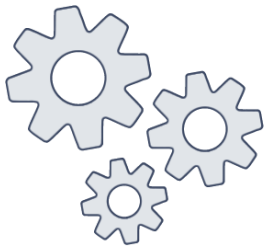
Demonstration: Creating a razor page layout template

Now watch us in action! We will take the pure CSS Blog template and import it into Sitefinity - http://purecss.io/



github:gist https://gist.github.com/ivaneftimov/85dd2a7f6eb702204bc364f337ca972e

Exercise: Creating a razor layout view for the DevMagazine website



1. Download the Resource Package for the DevMagazine site from:
   - https://github.com/Sitefinity/AdvancedDevCert/tree/master/DevMagazineHtml
2. Create the required folder structure for the page layout template.
3. Create the page layout template file – <yourname>.cshtml.
4. Copy the static HTML from the Home.html file.
5. Identify what elements needs to be replaced, and insert placeholders.

How many placeholders will you create?

Where will you put them?

Exercise: End result

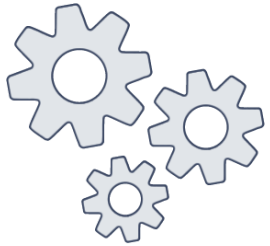
Something like:

```
@using System.Web.Mvc;
@using Telerik.Sitefinity.Frontend.Mvc.Helpers;
@using Telerik.Sitefinity.Modules.Pages;
@using Telerik.Sitefinity.UI.MVC;
@using Telerik.Sitefinity.Services;

<!DOCTYPE html>
<html @Html.RenderLangAttribute()>
<head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1" />
    <meta charset="utf-8" />
    <title></title>

    @Html.Section("head")
</head>

<body>
    @Html.Section("top")

    <div class="innerContent" id="innerContent">
        @Html.SfPlaceHolder("innerContent")
    </div>

    @Html.Section("bottom")
</body>
</html>
```

OR

github:gist https://gist.github.com/ivaneftimov/e26c1385902a297b823e49c94080be52

Exercise: Use the created page template



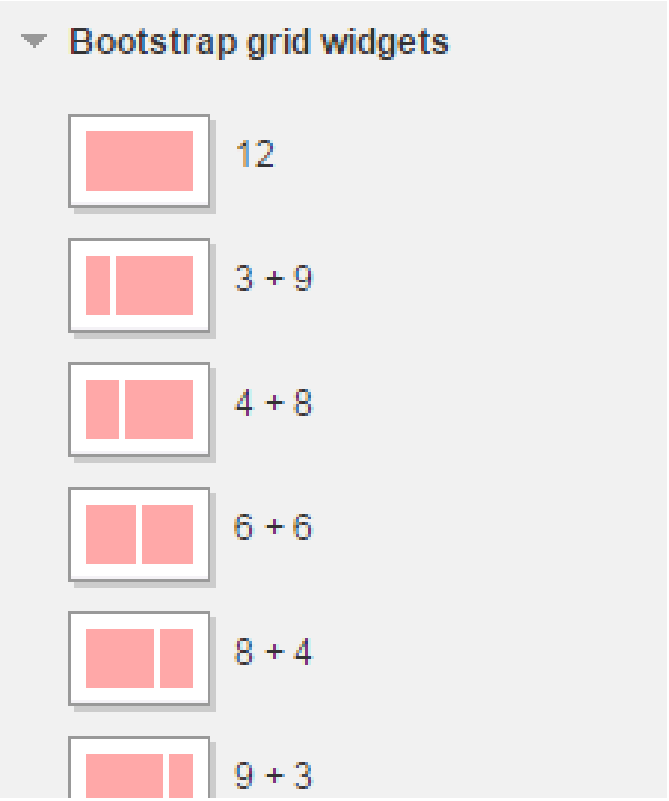
1. Open the backend and validate that a template based on your file was created in Sitefinity.
2. Create a page in Sitefinity using your template.
3. Populate with widgets.
4. Publish and test.

Step 3: Refining your page template by adding grid widgets

- You can define the layout for your page template by adding grid widgets. A grid widget is a container for other widgets.

- To add grid widgets:
  1. Open the page template.
  2. Add grid widgets to the page template.

- You can use any of the built-in grid widgets or create a custom grid widget.
  - Their markup should contain CSS classes that are consistent with the CSS framework of choice.



Step 4: Creating a custom grid widget

- Create a plain HTML file and place it in:
  `/ResourcePackages/[PackageName]/GridSystem/Templates/[grid-template-name].html`

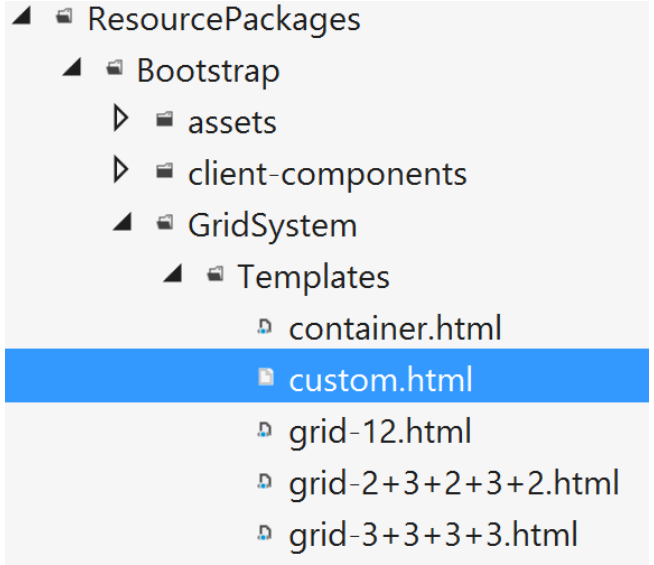
- Use the following Sitefinity-specific elements to define your custom grid widget:
  - `sf_colsIn` – a CSS class that Sitefinity uses to detect container elements
  - `data-sf-element="Row"`– AngularJS directive used to find the wrapper/parent containers
  - `data-placeholder-label="Text"` – used to provide user-friendly labels for editors

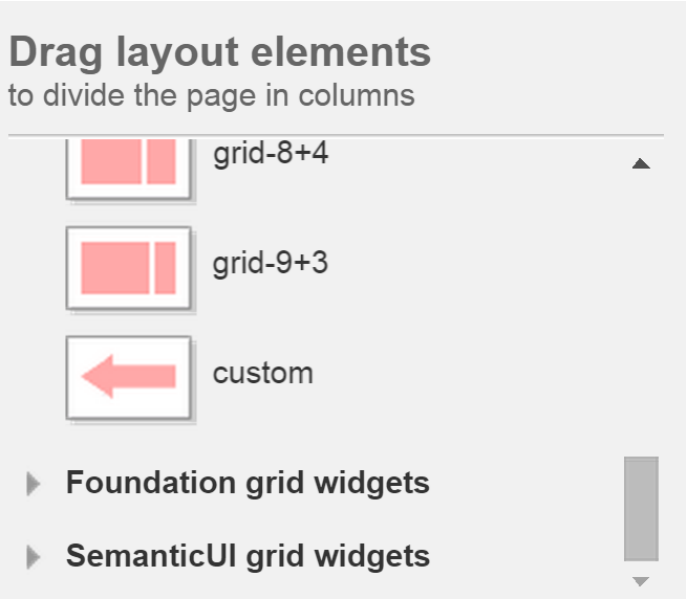
Conventions are documented here:

http://docs.sitefinity.com/feather-add-customizable-grid-controls

Example: Creating a custom grid widget

```
<div class="row" data-sf-element="Row">
    <div class="sf_colsIn col-md-3" data-sf-element="Column 1"
        data-placeholder-label="Column 1">
    </div>
    <div class="sf_colsIn col-md-9" data-sf-element="Column 2"
        data-placeholder-label="Column 2">
     </div>
</div>
```





Demonstration: Creating a custom grid widget

Now watch us create a grid widget
http://purecss.io/grids

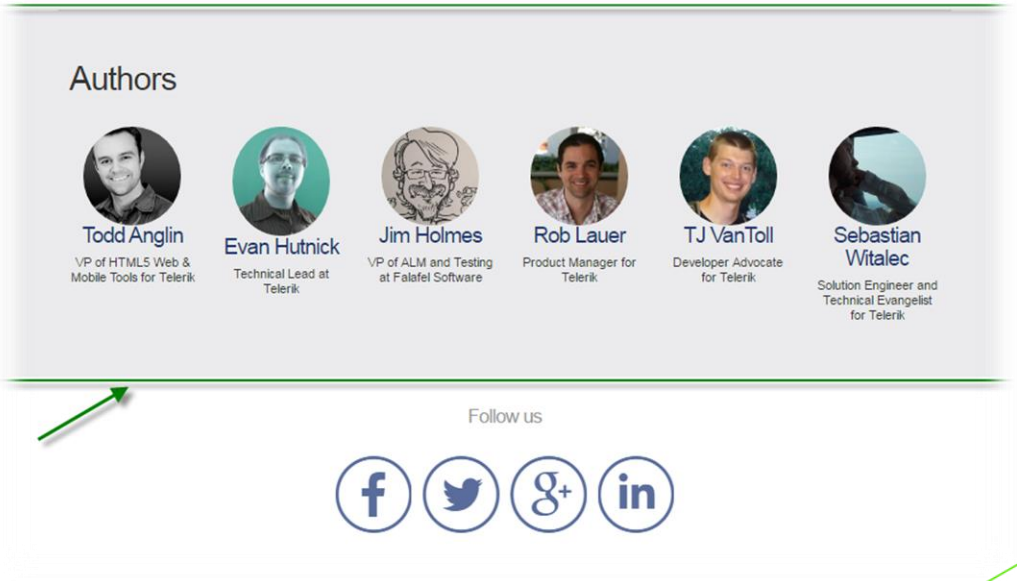
```
<div class="pure-g">
    <div class="pure-u-1-3"><p>Thirds</p></div>
    <div class="pure-u-1-3"><p>Thirds</p></div>
    <div class="pure-u-1-3"><p>Thirds</p></div>
</div>
```

```
<div class="pure-g" data-sf-element="Row">
    <div class="sf_colsIn pure-u-1-3" data-placeholder-label="Column Left"></div>
    <div class="sf_colsIn pure-u-1-3" data-placeholder-label="Column Middle"></div>
    <div class="sf_colsIn pure-u-1-3" data-placeholder-label="Column Right"></div>
</div>
```

Exercise: Creating a custom grid widget

Create a grid widget for the grey section used in the DevMagazine site

1. Open the wireframes and get the CSS/HTML that defines the authors section.
2. Use the convention and create a grid widget named grey-section.html.
3. Make sure that the HTML will generate the wrapper needed.
4. Drag and drop the grid widget onto the main template.



github:gist https://gist.github.com/ivaneftimov/e2a00e10ce0525b26ed3b470c1948e8b

Step 5: Determining widget requirements

Think of widgets

- Can you extend existing widgets?
- Do you need to develop a custom widget?

Always look for existing functionality and don’t reinvent the wheel. DO NOT REPEAT YOURSELF

Intermediate Recap: Presentation Layer

What have we learned?

1. Obtain HTML from front-end Developer.
2. Create a razor layout view
3. Optionally, refine your page template by adding grid widgets.
4. Create custom grid widgets, if needed.
5. Determine widget requirements.
6. Develop custom widgets, as required.
7. Extend existing widgets, as required.

Reminder: What are the main components of an MVC widget?

- Sitefinity MVC widgets, as the name suggest, follow the MVC architectural style
  - They have a model
  - They have a view
  - They have a controller

❖ If you have ever created an MVC page in ASP.NET, you would know how to work with an MVC widget in Sitefinity.

Step 6: Developing a custom MVC widget

- You can either create a completely new custom MVC widget or inherit from a built-in widget as follows:
  1. Create the controller (.cs).
  2. Create the model (data format, .cs).
  3. Create the view (.cshtml).

- If you are inheriting from an existing built-in widget, note that:
  - Majority of *actions* are not virtual.
  - You can add additional actions to existing controllers.
  - You can override models.

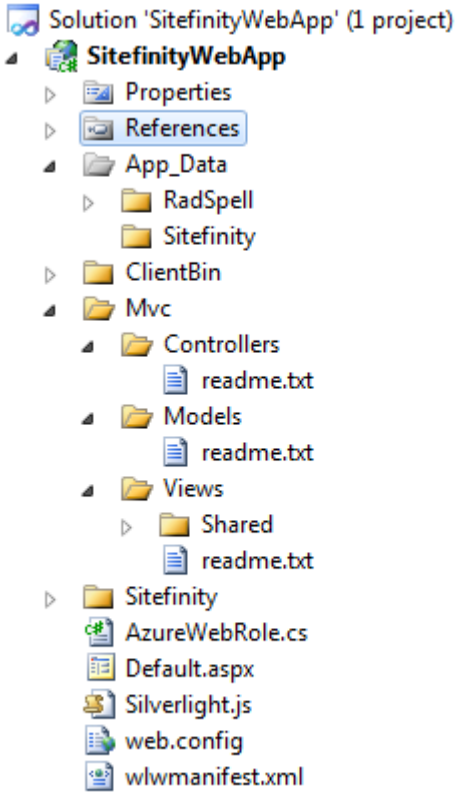
Demonstration: Creating a custom MVC widget

Now watch us create a new custom widget

BreakingNews

github:gist https://gist.github.com/ivaneftimov/c2a64a235f09d3eb34b544114d023b23

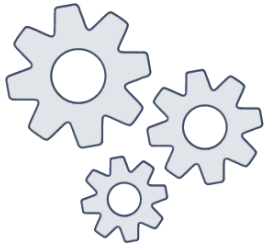
Exercise: Create a “Hello World” Sitefinity MVC widget



1. Create an ASP.NET MVC controller named HelloWorldController in the Mvc/Controllers folder.
2. Use the MVC pattern to create the View and the Model class. The controller should pass a “Hello World” string message to the view. Note: You could follow the approach described in the documentation: http://docs.sitefinity.com/for-developers-create-custom-models-controllers-and-views
3. Use the `ControllerToolboxItem` attribute on the controller class:
   [ControllerToolboxItem(Name = “HelloWorld", Title = “Hello World", SectionName = "CustomMvcWidgets")]
4. Build and test.

Why do we need the controller toolbox attribute?

Exercise: Add an action that prints a random number

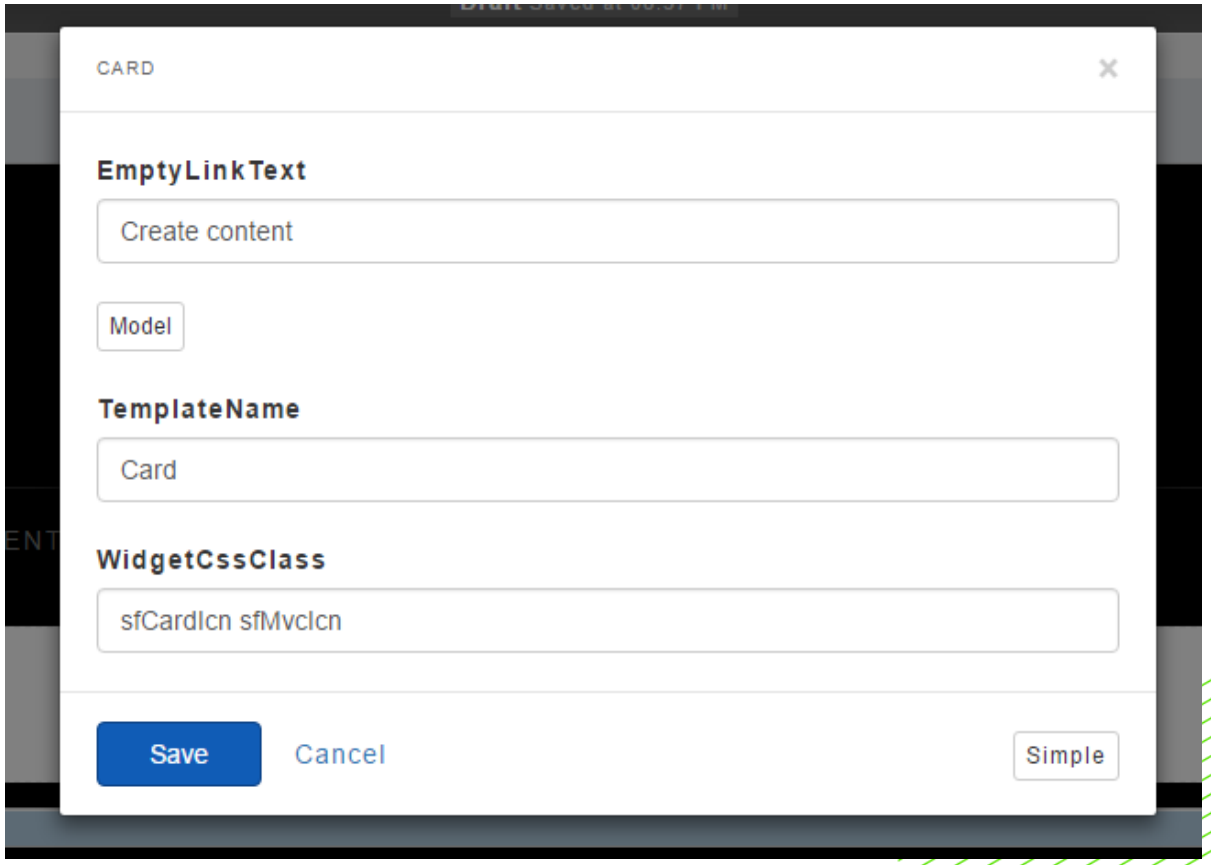


1. Add a new action named RandomNumber to the HelloWorldController.
2. Add an integer property to the HelloWorldModel class.
3. Wrtie code in the controller’s RandomNumber action to generate a random number and use it to set the integer property's value.
4. Create a view for the RandomNumber action to renders the random number.
5. In the controller, write code to pass the model to the view.

How would you invoke the RandomNumber action?

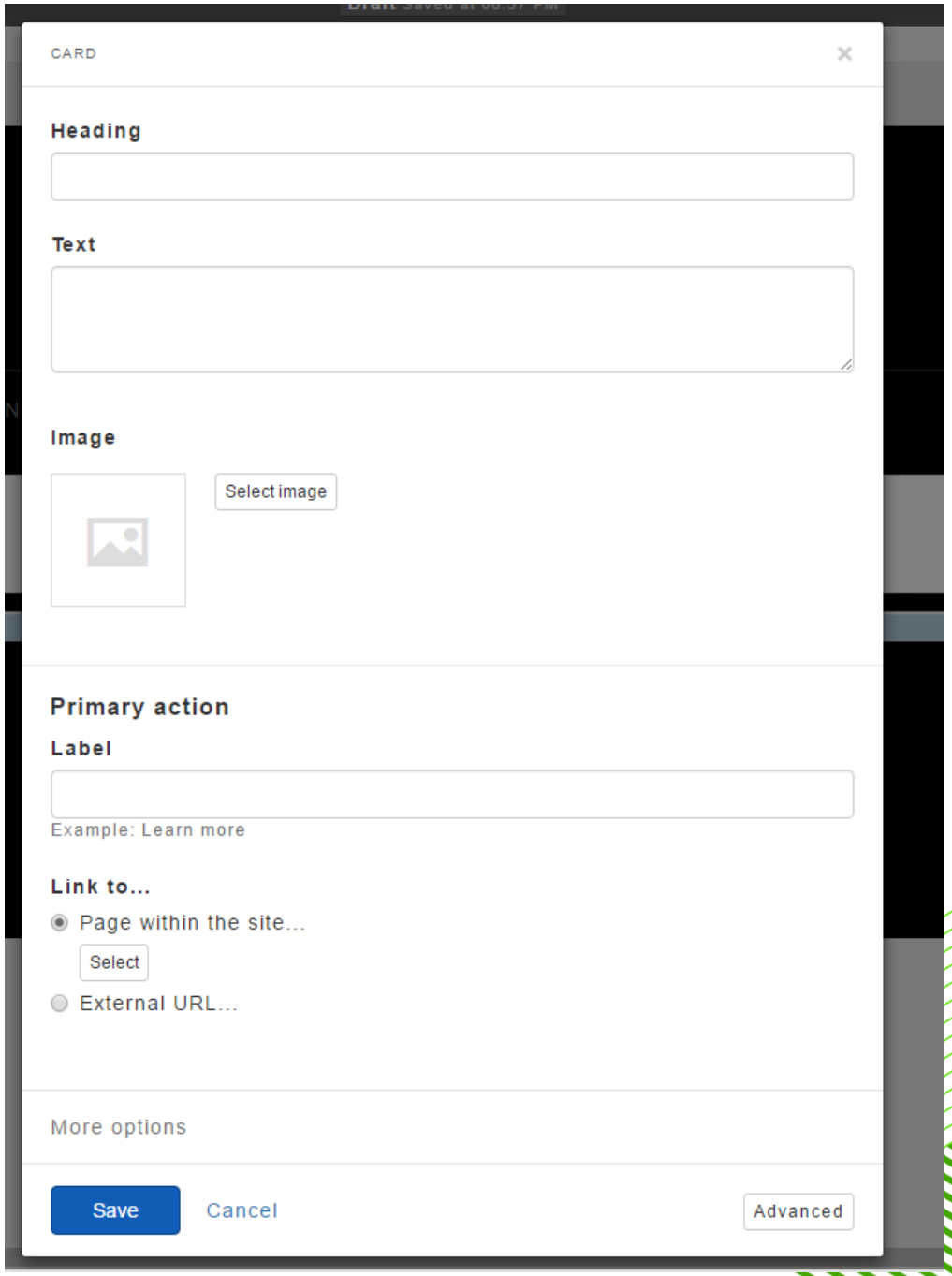
Property persistence

- Sitefinity persists a controller's public properties (for example: `SomeProperty {get; set;}`) in the database.
- Sitefinity automatically generates an editing UI called the **widget designer** for these properties.

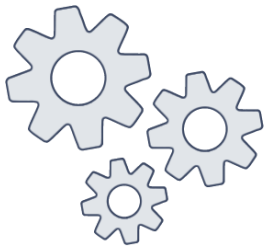


Property persistence, continued

- Properties can be also exposed with a **manually developed widget designer** to offer a tailored and fully customizable user experience for editing the properties.



Exercise: Create a static widget that adds two numbers



1. Create new controller called Adder.
2. Add two public properties: `int A` and `int B`.
3. Write code in the Index action to compute and return the sum of both properties (A + B).
4. Create a model class and a view, and pass the model from the controller's Index action to the view.

What was the problem with this approach?
Redundancy – it goes against DRY principles

Exercise: Create a static widget that adds two numbers, continued

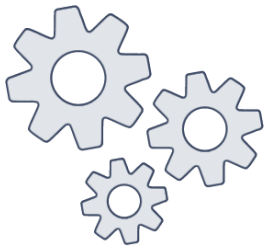
- Redundancy: Model has fields, Controller has fields
- Technically: public fields of the controller become the model

⇒ Solution: persist the model as the field using the `TypeConverter` attribute

```
[TypeConverter(typeof(ExpandableObjectConverter))]
public ExampleModel Model
{
    get
    {
        if (this.model == null)
            this.model = this.InitializeModel();

        return this.model;
    }
}
```

Exercise: Refactor model and test

Exercise: Invoking controller actions

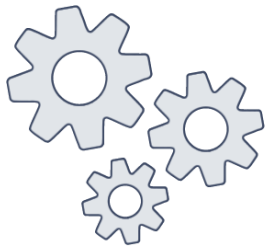


1. Add both (HelloWorld and Adder) custom widgets on the same page and publish the page.
2. Invoke the RandomNumber action.
3. Observe.
4. What happened?

Action Invoking Flow

The default action registered in the route is the “Index” action and it will be invoked when the route does not specify any other action.

Exercise: Invoking controller actions, continued



1. Now, add the built-in ContentBlock widget onto the same page (that has the Adder and the HelloWorld widgets).
2. Enter some text in the content block widget.
3. Publish the page and invoke the RandomNumber action.
4. What happened this time?

The ContentBlock Widget stayed, right?

Let’s see how this happened!

Action routing

- Action couldn't be found on the controller.
- This is not desired in most cases.
- It is important to have template names hardcoded as strings, otherwise the template name will be resolved to the current action.

Solution? Use `HandleUnknownAction`

```
protected override void HandleUnknownAction(string actionName)
{
    this.Index().ExecuteResult(this.ControllerContext);
}
```

Action routing, continued

- It is possible to define routes using attributes and MVC5 RelativeRoute

```
public class SampleController : Controller
  {
      [RelativeRoute("my-sample-path")]
      public ActionResult Action1()
      {
          return Content("This is Action1");
      }

      [Route("my-sample-path")]
      public ActionResult Action2()
      {
          return Content("This is Action2");
      }
  }
```

Action routing, continued

- Relative path to the page node where the widget is placed – within Sitefinity route:

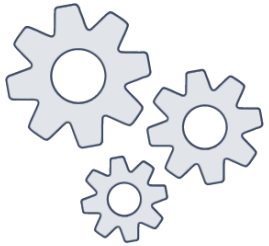
  [RelativeRoute("my-sample-path")]
  Result: ~/my-page/my-sample-path

- Route relative to the application path – ignores Sitefinity route:

  [Route("~/my-sample-path")]
  Result: ~/my-sample-path

Documentation link - http://docs.sitefinity.com/feather-use-the-relative-routes-api

Exercise: Creating a custom MVC widget



- Create a widget named *Webinar* with the following properties:
  - Title
  - Description
  - StartTime
  - EndTime

❖ You will use this widget later in the course, connecting it to the data layer.

github:gist https://gist.github.com/ivaneftimov/105e9fb7248efd6414f12839f8913515

Step 7: Extending existing widgets

- There are three ways to extend an existing widget:
  - By extending the widget template
  - By creating a new template
  - By extending the widget view model

Extending existing widgets | Template Conventions

- Examples of widget templates that are extended are located in the Bootstrap resource package.
- Each widget should have a folder with its name for resolving views:
  - ~/ResourcePackages/[PackageName]/MVC/Views/[WidgetName]/[ViewName].cshtml
- There should be a dedicated folder per package, for example, the ‘Bootstrap’ folder.
- You only need to edit and save views; there is no need to build the solution.
- Naming conventions:
  - List.[ViewName].cshtml
  - Detail.[ViewName].cshtml

Widget templates are entirely powered by conventions. Conventions are king!

Extending existing widgets | Extending the widget template

1. Create a razor view template file (`.cshtml`) following the correct convention depending on whether you extend the `List` or the `Detail` view of the widget
   - Normal
     `/ResourcePackages/[PackageName]/Mvc/Views/[BuiltInWidgetName]/[TemplateName].cshtml`
   - List
     `/ResourcePackages/[PackageName]/Mvc/Views/[BuiltInWidgetName]/List.[TemplateName].cshtml`
   - Detail
     `/ResourcePackages/[PackageName]/Mvc/Views/[BuiltInWidgetName]/Detail.[TemplateName].cshtml`

2. Build the markup of this view as required.

Sitefinity controllers declare template paths as properties, therefore you can change existing templates with no code

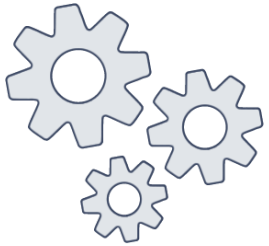
Demonstration: Extending a widget template

Now watch us extend a few widgets

Convention is king!

There is a very descriptive documentation on the topic -
http://docs.sitefinity.com/feather-modify-widget-templates

Exercise: Extending a widget template

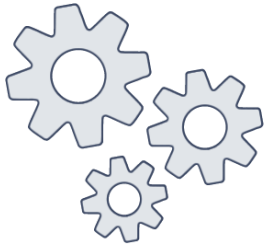


Extend the NewsList widget template:

1. Open /ResourcePackages/Bootstrap/MVC/Views/News.
2. Copy the existing List.NewsList.cshtml template.
3. Create the same folder structure in your resource package and paste the template.
4. Modify it according to the DevMagazine requirements, applying the CSS classes and structure.
5. Objective: Render the `UrlName` property as a hashtag after the title of the news item.

Include the view in the project. This will activate IntelliSense and help with properties as you build the view. Doing this reduces errors and speeds up development.

Exercise: Creating a widget template

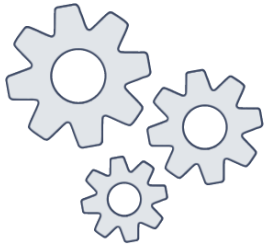


Create a new List widget template for the News widget.

1. Create an entirely new view for the News widget.
2. Start by cloning an existing one.
3. Use the correct convention:
   `/ResourcePackages/[PackageName]/Mvc/Views/[BuiltInWidgetName]/List.[TemplateName].cshtml`
4. Apply the view on a widget.

- How will you get the widget name?
- What is the best way to do so?

Extending existing widgets | Extending the widget view model



- In some cases, we need to include additional data fields in the views without changing the control's logic.
- Prime example: `NavigationController`

Exercise: Extend the existing `NavigationController` template and `ViewModel` to include a ***Rating*** field for pages.

- Ratings can be random values
- Sample output:

```
<li>
    <a href="PageUrl">
        <span>{{PageTitle}} <i>Rating: {{RatingValue}} </i></span>
    </a>
</li>
```

?

- Can you also add business logic?
- If yes, how would you approach it?

Extending existing widgets | Extending the widget view model, continued

- Steps
  1. `public class CustomNavigationModel : NavigationModel`
     - `override NodeViewModel InstantiateNodeViewModel(SiteMapNode node)`
     - `override NodeViewModel InstantiateNodeViewModel(string url, string target)`
  2. `public class CustomNodeViewModel : NodeViewModel`
  3. Modify the view to reflect the new field
  4. Inject your custom model on Initialize

github:gist https://gist.github.com/ivaneftimov/b4f6746c19f7fe9d6a26319806a20918

Widget caching and cache dependencies

- Feather adds cache dependencies on the view templates of its widgets so the output cache is invalidated when any of the templates is updated.
- There is an extension method for the controller of the widget called `AddCacheDependencies` that is used to add dependencies to the output cache much like WebForms widgets that implement `IHasCacheDependency`.
- The virtual file resolver chain accumulates cache dependencies as it goes down the chain.
  - For example: If a resource is not present on the file system and is found on the database, a cache dependency is still added on the file system AND the DB persistent object.

Lesson 3: The Widget Designer Framework

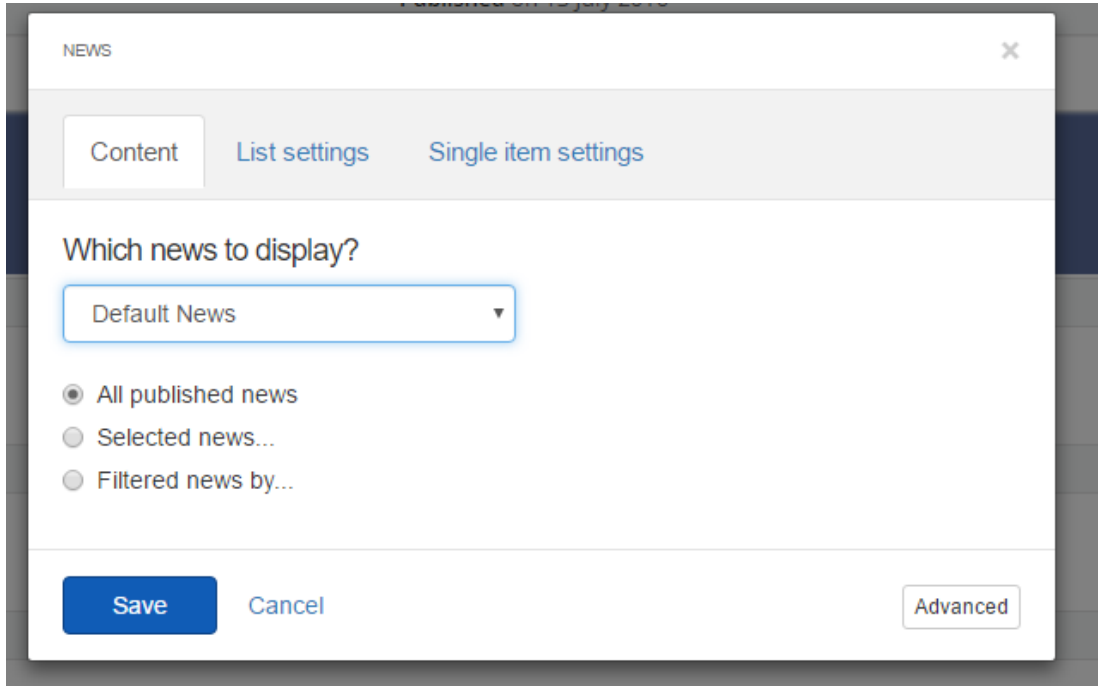
Lesson objectives

- By the end of this lesson, you should be able to:
  - Describe why you may need to develop a widget designer.
  - Create a custom widget designer.
  - Configure settings and add script references for the custom widget designer.
  - Use client components in your widget designer.

What is a widget designer? Why do you need to create one?

- Each widget has a widget designer which is used to configure the widget.

- Here is an example of a News widget designer:



You typically need to create widget designers for custom widgets, although you may also create a widget designer for a built-in widget.

Steps to create a widget designer

1. Create the designer view template (.cshtml) using the following convention:
   `/Mvc/Views/[WidgetName]/DesignerView.[DesignerName].cshtml`
2. Add the HTML/Razor code you need.
3. Optionally, use some of our built-in AngularJS components.
4. Optionally, create an AngularJS controller using the following convention:
   `/Mvc/Scripts/[WidgetName]/DesignerView-[WidgetDesignerName].js`
   - Write JS code in the AngularJS controller to manipulate data (including any to manipulate the built-in components)

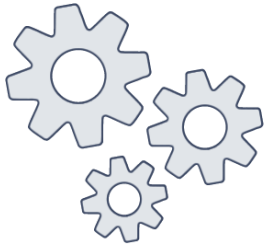
Note: If your widget designer requires additional script references, the best practice is to add them to a special JSON file that contains configuration information about your designer. You'll learn more about this soon.

Demonstration: Creating a widget designer

Watch us create a widget designer for the *Breaking News* widget

https://gist.github.com/ivaneftimov/5483158dfd48cde9ef29f802e150435b

Exercise: Creating a widget designer



- Create a widget designer for the Webinar widget.
- Add HTML5 **input**s of type **date** for Start and End properties.

github:gist https://gist.github.com/ivaneftimov/bb13fd06495f771bd0186e9ec61cce84

Configuring settings and adding script references to the widget designer

- After you create a widget designer, you may need to:
  - Set it as the default designer for the widget (if you have multiple widget designers for that widget).
  - Add additional script references that are required by the widget designer.

- In order to do that you need to:
  - Create a JSON file with the following convention:

```
/Mvc/Views/[WidgetName]/DesignerView.[DesignerName].json
```

  - Add configurations to the JSON file as shown here:

```
{
  "priority": 1,
  "scripts" : [
    "/<myscript>.js"
  ]
}
```

Use client components in your designer

- Sitefinity Feather comes with many out-of-the box client components
  - Items selectors
  - Pages selector
  - Taxonomies selector
  - Fields
    - Image field
    - File URL filed
- The list goes on and on …

The full list is here: http://docs.sitefinity.com/feather-client-components

Demonstration: Using a news item selector

- Let's put a news item selector to our BreakingNews widget designer
  - Let's refer to the documentation: http://docs.sitefinity.com/feather-client-components
  - First we add the script references, **if needed.**
  - Next we add the module dependency in our .js file:
    ```
    angular.module('designer').requires.push('sfSelectors');
    ```
  - Next we add the HTML element of the selector:
    ```
    <sf-list-selector sf-news-selector />
    ```
- And that's it – we should now have a news item selector in our designer!

Demonstration: Using a news item selector | Part 2

- In order to be able to save the selection, the news selector has two additional attributes: `sf-selected-item` and `sf-selected-item-id`
- These two attributes expose the selected item/id and allow us to store their value as widget properties
  - We need a widget property for this:
    ```
    public string SelectedItem { get; set; }
    ```
  - Then we need angular watches that will listen for the changes in the selection and update the values appropriately:

```
$scope.$watch('properties.SelectedItem.PropertyValue', function (newValue, oldValue) {
     if (newValue) {
         $scope.selectedItem = JSON.parse(newValue);
     }
});

$scope.$watch('selectedItem', function (newValue, oldValue) {
    if (newValue) {
        $scope.properties.SelectedItem.PropertyValue = JSON.stringify(newValue);
    }
});
```

Demonstration: Customizing the news item selector

- Change the template of the closed dialog
  - By default, when you close the dialog, you see the title of the selected item and a button to open the dialog. To change this behavior, you must add HTML code between the opening and closing tags of the selector's directive. For example:

```
<sf-list-selector sf-news-selector sf-selected-item="selectedItem">
    <span ng-bind="sfSelectedItem.Title"></span>
    <button class="openSelectorBtn" ng-click="open()">Select a breaking news item</button>
</sf-list-selector>
```

github:gist https://gist.github.com/ivaneftimov/6b8bbcc66b3fcba9b8ee8556853cebbe

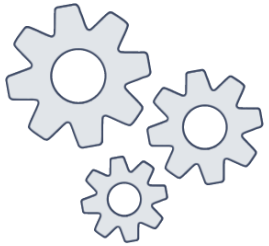
- Change the template of selector

```
<sf-list-selector sf-news-selector sf-template-assembly="SitefinityWebApp"
        sf-template-url="templates/custom-selector.html" />
```

- Change the text of buttons

```
<sf-list-selector sf-news-selector sf-select-button-text="Select news..."
        sf-change-button-text="Change news..." />
```

Exercise: Dynamic content selector in the Webinar widget designer



- Add an item selector for the Webinar items:
  - Create a new Webinars dynamic module
  - Refer to the documentation for configuring the selector: http://docs.sitefinity.com/feather-client-components
- Customize the template of the selector.

github:gist https://gist.github.com/ivaneftimov/e01f51efddeb3cf7e617e6873b8525c9

Review of Presentation Layer concepts

Quick review of presentation layer

- Sitefinity Feather provides the framework for building the presentation layer of your application in an MVC way.
- The presentation layer consists mainly of page templates, pages, and widgets.
- You create page templates and then create pages based on the page templates.
- Pages do not have a physical location in the file system. They are stored in the database.
- You can use built-in widgets as is, extend them as needed, or develop your own custom widgets.
- You can also build you own widget designers.

Now that you have your presentation layer ready, the next step is to work with the data layer.

Lesson 4: Bringing content to the presentation layer using APIs

Lesson objectives

- By the end of this lesson, you should be able to:
  - Use the Native API to bring content to a custom MVC widget.
  - Use the Fluent API to bring content to a custom MVC widget.
  - Use the REST API to expose Sitefinity content to client applications.

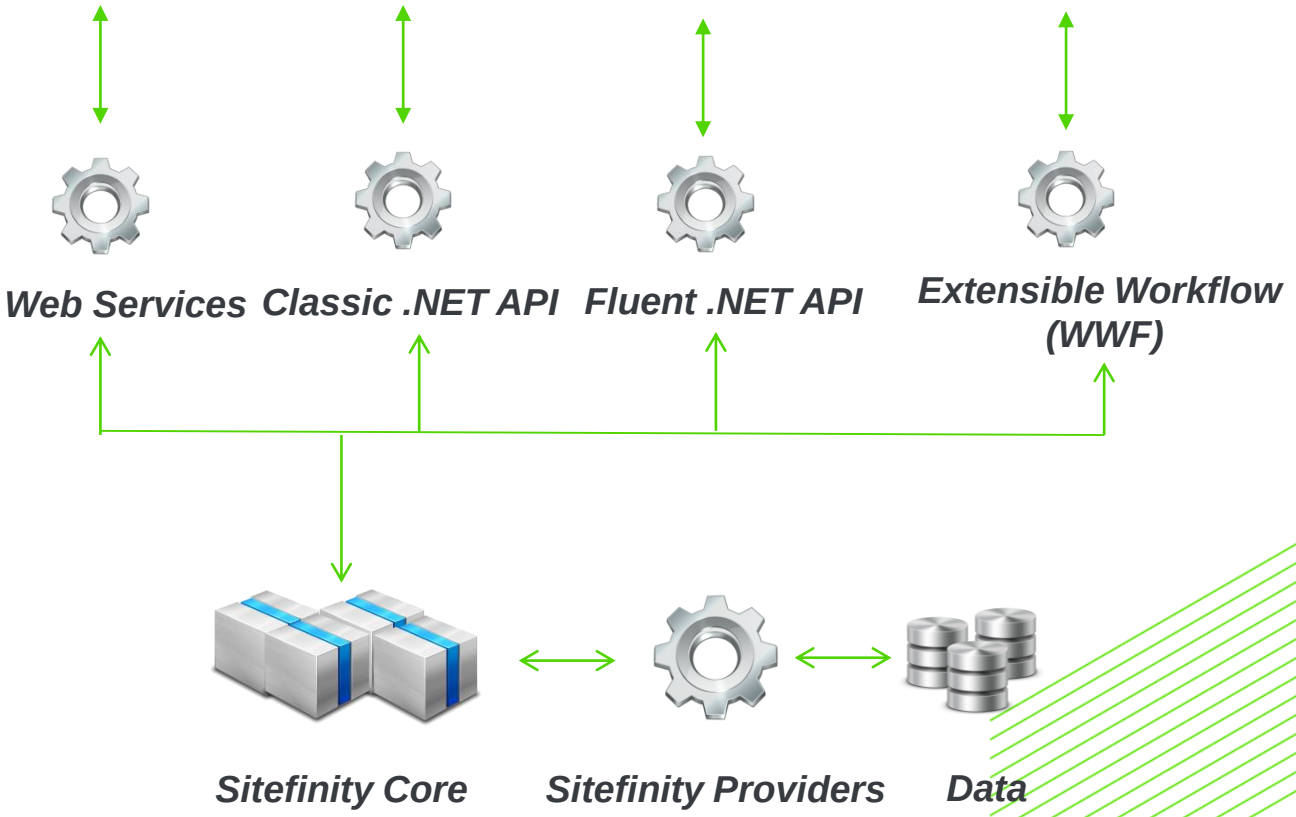
Everything is an API

Requirements are always different.

Available APIs:

- ***API and REST Services for Content***
- ***API for Taxonomies***
- ***API for Workflows***
- ***API for UI (widgets)***
- ***System API – Synchronization, Tasks scheduling, Integration, publication sstem…***
- ***.. more***

Everything in Sitefinity is an API:



Three ways to bring content to the presentation layer

- Native API - The most direct way to access content. Based on the Provider pattern, it exposes Managers for each content type.
- Fluent API - Built on top of the Native API. It abstracts the Provider-Manager architecture and provides an easy way to work with content.
- REST Services - Provides a standard way for exposing content to client applications.

- How to use them:
  - Native API provides the most flexibility and enables testing.
  - Fluent API provides simplicity but does not enable testing and cannot work with dynamic content.
  - REST Services are mainly used by client applications to work with Sitefinity content. You can also use them while building widget designers.

Using the Native API

- All content types have a dedicated Manager which exposes full range of methods for manipulating the correspondent content:
  - NewsManager
  - EventsManager
  - BlogsManager
  - …
  - DynamicContentManager – used to manage all your dynamic content types

- Additional managers which are not directly related to specific content type but still follow the Manager->Provider architecture:
  - PageManager
  - ConfigManager
  - TaxonomyManager

Code example: Using the Native API

- Using a Manager in Sitefinity
  - Get an instance of the `Manager`:

```
NewsManager newsManager = NewsManager.GetManager();
```

or

```
NewsManager newsManager = NewsManager.GetManager("providerName");
```

  - Perform the required CRUD operation to the items:
    - Example: Get all news items published in the last day

```
var newsItems = newsManager.GetNewsItems()
      .Where(n => n.PublicationDate > DateTime.UtcNow.AddDays(-1));
```

    - Example: Change the title of a news item

```
var newsItem = newsManager.GetNewsItem(newsItemId);
newsItem.Title = "New Title";
newsManager.SaveChanges();
```

Using the Native API | Bringing the content to the presentation layer

- Retrieving the content in the controller
- Building a model of the content
- Passing the model to the view

Demonstration: Using the Native API to bring content to the presentation layer

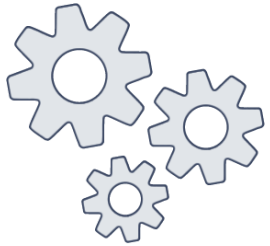
- Import/create NewsItems to the DevMagazine site.
- Add the tag BreakingNews to some of the NewsItems.
- Bring the tagged NewsItems to the BreakingNews widget.

Look at the code below for an example of getting news items tagged with the `BreakingNews` tag.

```
TaxonomyManager taxonomyManager = TaxonomyManager.GetManager();

var breakingNewsTaxon = taxonomyManager.GetTaxa<FlatTaxon>()
        .Where(t => t.Name == "BreakingNews").FirstOrDefault();


NewsManager newsManager = NewsManager.GetManager();

var newsItems = newsManager.GetNewsItems()
        .Where(n => n.GetValue<IList<Guid>>(breakingNewsTaxon.Taxonomy.Name)
                        .Contains(breakingNewsTaxon.Id));
```

github:gist https://gist.github.com/ivaneftimov/30a54375a84672961f43f26f8ff2e07d

Exercise: Using the Native API to bring content to the presentation layer



- Remember that you created a custom MVC widget named Webinar earlier.
- You also created a Webinars content module.
- Use the Native API to bring the first *published* Webinar item to the Webinar widget

Do not use dynamic content selector!

The point of the exercise is to practice using the Native API.

Using the Fluent API

- Sitefinity developers need to know four important things in order to use the Fluent API:
  1. The entry point and entry methods
  2. The façades
  3. The end methods
  4. The limitations of the API

! *Dynamic content cannot be managed by the Fluent API.*

Using the Fluent API | Entry point and entry methods

- You must start each Fluent API call with the `App` static class.
- You must follow the `App` class with:
  - The `WorkWith()` method if you want to use the default provider for your content. For example:

```
var newsItems = App.WorkWith().NewsItems().Get();
```

or

  - The `Prepare()` method if you want to use advanced configurations in your façade, for example, when you want to use a non-default provider:

```
var newsItems = App.Prepare().SetContentProvider("MyProvider")
                           .WorkWith()
                           .NewsItems()
                           .Get();
```

Using the Fluent API | Façades

- All of the *static* content types can be managed by the Fluent API. They all have a façade over their correspondent Manager in order to simplify the most common actions of managing the content. Some of them are:
  - Page - used to work with a page - PageFacade
  - Pages - used to work with a set of pages - PagesFacade
  - NewsItem - used to work with a news item - NewsItemFacade
  - NewsItems - used to work with a set of news items - NewsItemsFacades
  - …

For the full list of façades go here: http://docs.sitefinity.com/for-developers-facades

Using the Fluent API | End methods

- Each Fluent API call ends up with one of the following methods
  - `SaveChanges()`
    - Saves all the changes in the database - it commits the transaction
  - `CancelChanges()`
    - Discards any actions initiated within the current call
  - `Done()`
    - It saves the changes in the current scope, but the transaction is not executed until you call the `SaveChanges()` method

More details and examples here:
http://docs.sitefinity.com/for-developers-save-and-discard-changes

Code example: Using the Fluent API

- Let's rewrite some of the Native API examples with the Fluent API

Native API

- Get an instance of the Manager

```
NewsManager newsManager = NewsManager.GetManager();
```

- Example: Get all news items published in the last day

```
var newsItems = newsManager.GetNewsItems()
    .Where(n => n.PublicationDate > DateTime.UtcNow.AddDays(-1));
```

- Example: Change the title of a news item

```
var newsItem = newsManager.GetNewsItem(newsItemId);
newsItem.Title = "New Title";
newsManager.SaveChanges();
```

Code example: Using the Fluent API

- Let's rewrite some of the Native API examples with the Fluent API

Fluent API

- Example: Get all news items published in the last day

```
var newsItems2 = App.WorkWith().NewsItems().Get()
    .Where(n => n.PublicationDate > DateTime.UtcNow.AddDays(-1));
```

- Example: Change the title of a news item

```
App.WorkWith().NewsItem(newsItemId)
     .Do(n =>
      {
          n.Title = "New title";
      })
     .SaveChanges();
```

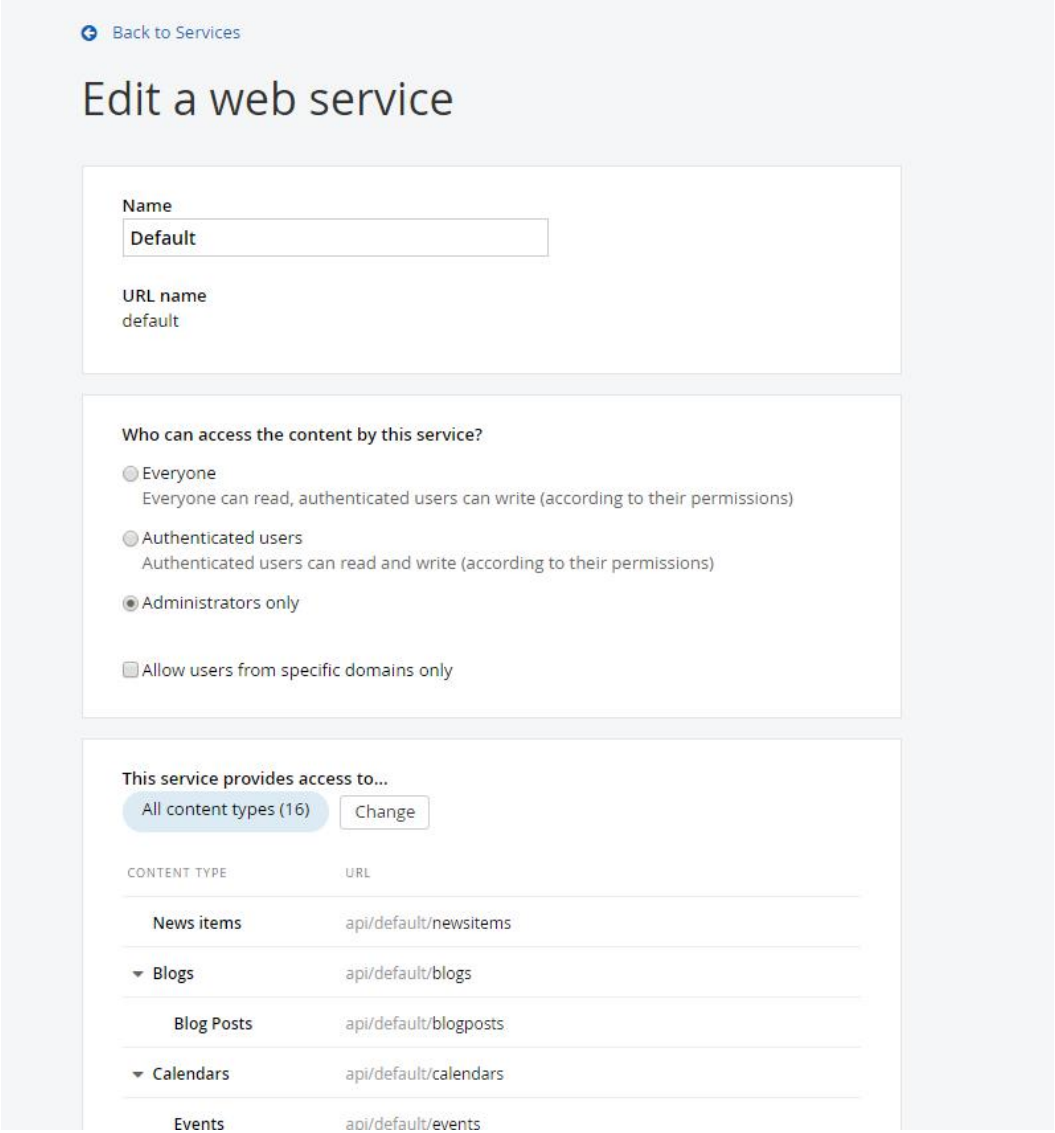
Using REST Services

- Sitefinity exposes its content and functionality through a variety of web services, so that third-party applications can use it (such as mobile applications). The different types of web services are:
  - RESTful WCF services – the old web services for managing content in Sitefinity. They are specifically tailored for Sitefinity's backend functionalities.
  - ServiceStack services – used in the backend and frontend of Sitefinity. ServiceStack license is included with Sitefinity, so you can develop your own services with it.
  - Configurable REST services – the new web services in Sitefinity, based on ASP.NET Web API and supporting the oDATA protocol. We are going to focus on these.

Using REST Services | The *oData* Services

- The main benefits of using these services are the following:
  - Support for multiple RESTful API services
  - User-friendly UI for configuration of the services
  - Control over the set of content types that are exposed
  - Option to allow anonymous access per profile or type
  - Easy access to related data
  - Auto-generated API reference
  - Support for saved queries and calculated fields

Demonstration: Using REST Services



Let's show a breaking news dropdown list in the BreakingNews widget designer:

1. Call the news REST service with filter for the BreakingNews tag (The GUID is the ID of this tag):

```
$http.get('/api/default/newsitems?$filter=Tags/any(x:x eq
          8DE887D5-6FCD-664F-8951-FF000085FCE0)')
   .then(function (response) {
        $scope.newsItems = response.data.value;
    });
```

2. Add the required markup in the designer view:

```
<select ng-options="item.Title for item in newsItems"
ng-model="selectedItem"></select>
```

Lesson 5: Using Providers to Connect to Different Data Sources

Lesson objectives

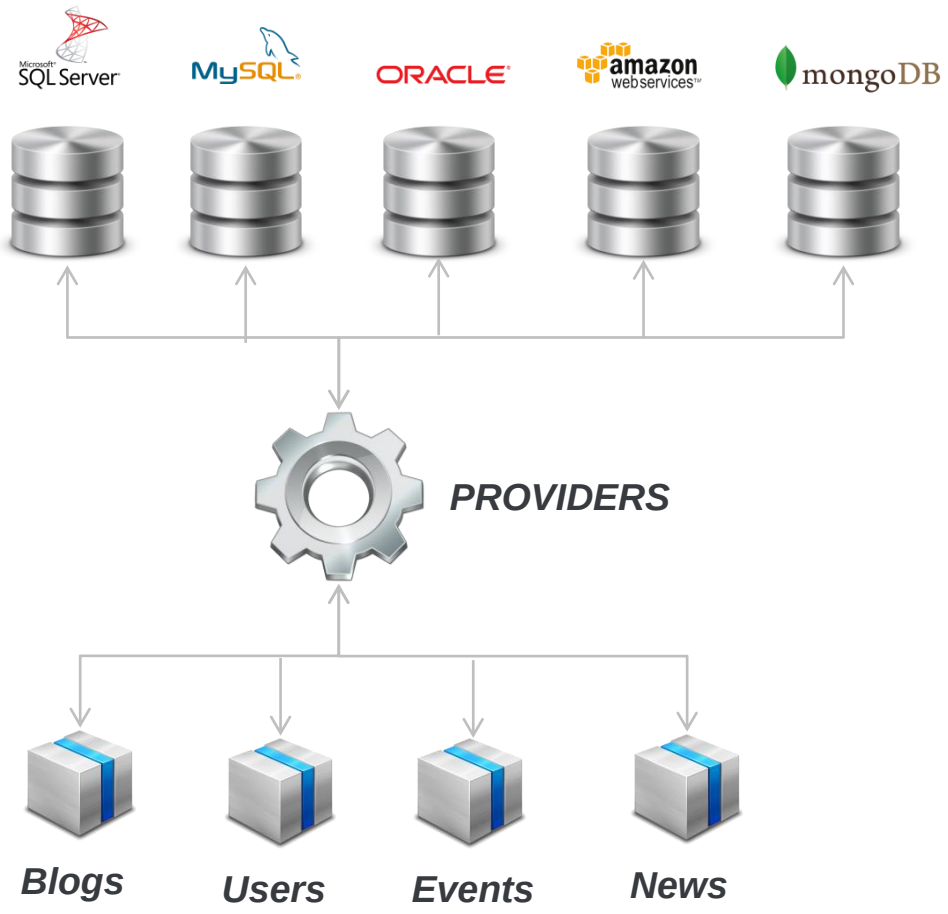
- By the end of this lesson, you should be able to:
  - Have clear understanding of what provider is.
  - Connect to different data sources using providers.

Quick Review: What is a Provider?

- A Provider is an architectural pattern that enables you to connect your Sitefinity application to different data sources.
- Regardless of the data source, as long as you implement Sitefinity's base structure for a Provider, Sitefinity will be able to use and visualize the data in the same way as native content.

- https://en.wikipedia.org/wiki/Provider_model

Quick Review: Data layer architecture



Single Sitefinity Instance

Steps to connect to different data sources using Providers

1. Create a new class that inherits from the Provider Base Class.
2. Implement methods you intend to use: create, get, delete, etc.
3. Register the provider in the configuration of the appropriate module.

Demonstration: Connecting to static membership (users) content

- The three crucial steps:
  1. Inherit `MembershipDataProvider`.
  2. Implement `IQueryable<User> GetUsers()` and `User GetUser(Guid id)` methods.
  3. Register the provider in the SecurityConfig.

❖ Now you should be able to see this new provider in Administration > Users screen

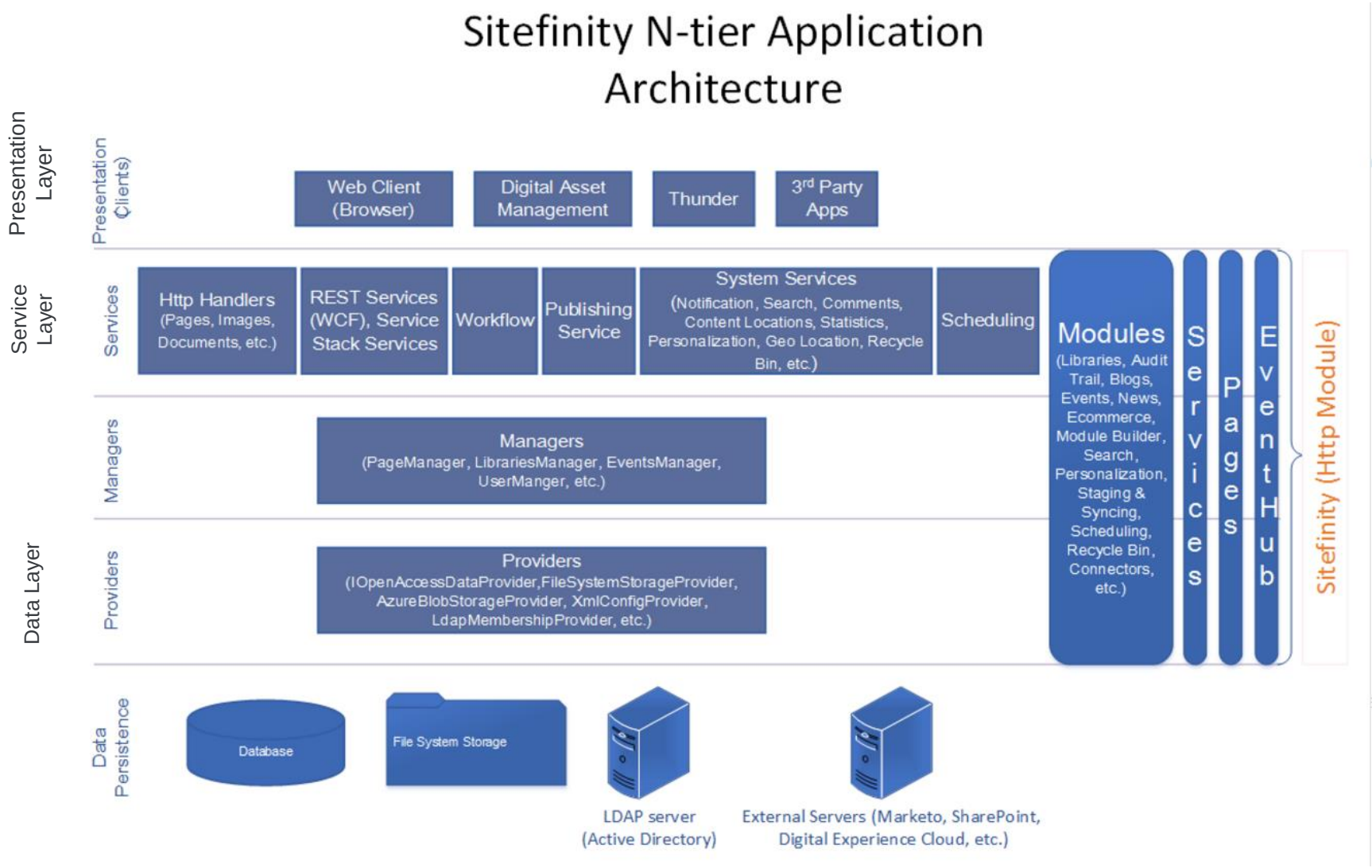
github:gist https://gist.github.com/ivaneftimov/9accb72f9fe36599002202aa5e1c5b6d

Exercise: Connecting to an external news source

- Follow the same steps as in the demo you saw before.

https://gist.github.com/ivaneftimov/3120ef4677264b2f835f0bd970ac6d59

The full picture: Sitefinity application architecture



Lesson 6: Localization

Lesson objectives

- By the end of this lesson you should be able to:
  - Localize your widget template.
  - Localize your widget designer components.

- We will not focus on the multilingual features of Sitefinity, which allow you to store different language versions of a specific page or content item. The focus will be on localizing static text which appears in the frontend. This is the type of localization which the Sitefinity developer will usually deal with.

Code example: Localize your widget templates

- In order to localize your widget template, you need to follow there simple steps
  1. Create a *resource* class

     Inherit from `Telerik.Sitefinity.Localization.Resource`

  2. Add the resource entries for the strings you wish to localize

```
[ResourceEntry("Message", Value="Message", Description="The breaking news message.",
LastModified="2016/06/13")]
public string Message { get { return this["Message"]; } }
```

  3. Tie your resource class to your widget controller:

```
[Localization(typeof(BreakingNewsResources))]
```

  4. Use the *`Resource`* HTML helper to get the localized value of your string in your Razor templates

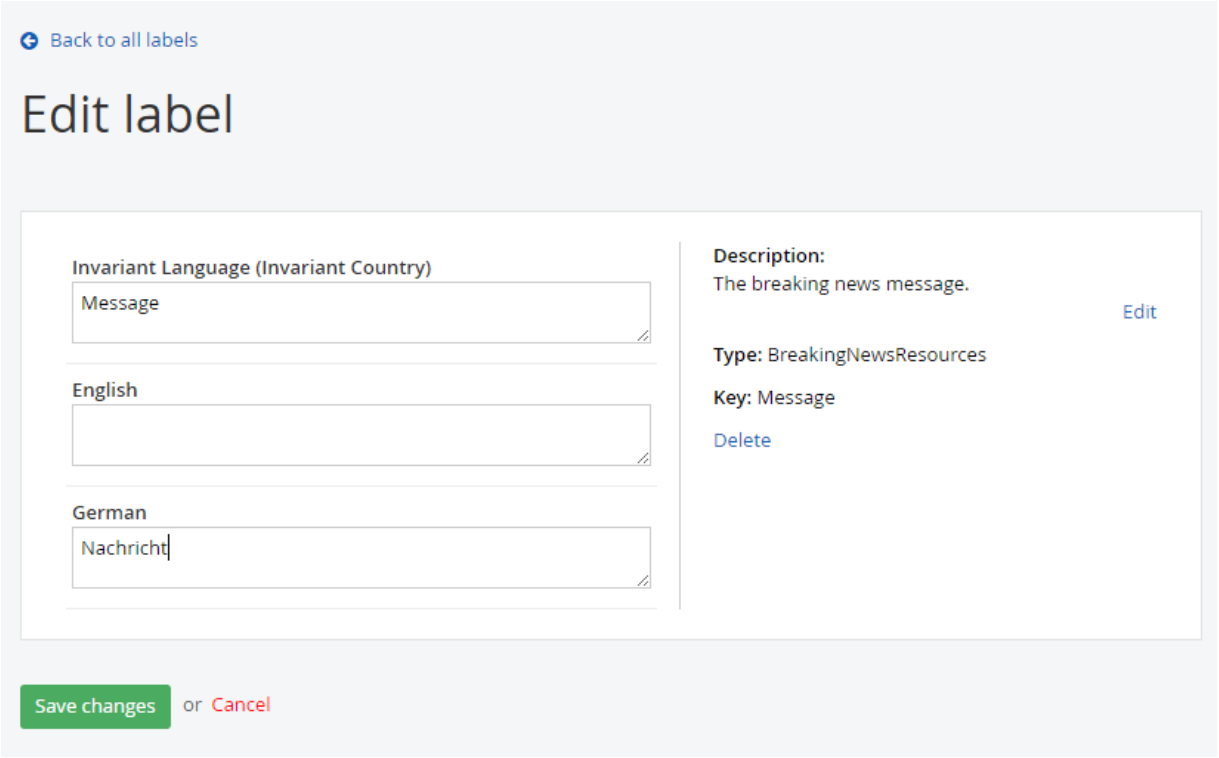
```
@Html.Resource("Message")
```

Don't forget to add the `using` for the Feather helper extension methods

```
@using Telerik.Sitefinity.Frontend.Mvc.Helpers
```

Demonstration: Localize your widget templates

1. Let's actually create a resource for our `BreakingNews` widget using the code snippets in the previous slide.
2. Let's add another culture (language) to our web application and see how we can enter the different language versions for our string resources.



github:gist

https://gist.github.com/ivaneftimov/41b385db3353960b201cf514ad48c4cf

Language Packs

- All of the backend of Sitefinity is written in a localizable way. Moreover all of the frontend widgets are localizable as well.
- Sitefinity allows you to import a language pack which can contain translations for all of the labels in the backend and built-in widgets, so you do not translate them one by one from the UI.
- There are language packs for the most common languages available here: http://www.sitefinity.com/developer-network/marketplace/publishers/telerik-inc
- If the language you want to use is not available on the Sitefinity CMS Marketplace, you must export and translate all labels and import them back again. For more details on how to export and import labels, see this article: http://docs.sitefinity.com/backend-languages.

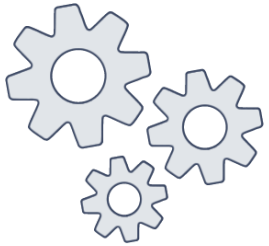
Code example: Localize client components

- Displaying localized text in Razor views was quite easy to implement.
- Displaying localized text in the widget component template (.html) is even easier.

- If you want to display localized content in the angular views of your components, which are normally `.html` files, you should:
  1. Change your file extension to `.sf-cshtml`
  2. Get the localized strings in the following way
     ```
     <h3>@(Res.Get<BreakingNewsResources>().Message)</h3>
     ```

❖ Do not forget that you should have previously created your resource class, in the same way as shown in the previous slides

Exercise: Localize the Webinar widget



- Localize the static text which appears on your Webinar widget
  - Add a second language for your website
  - Follow the steps from the demo you saw before, relating the BreakingNews widget
  - Add a language version for the new language of your string resources
  - Create a page with two language version, English and the new language you added before
  - Drop the Webinar widget on both versions
  - See the end result!

Lesson 7: Working with events

Subscribing and unsubscribing to events

- Sitefinity has a powerful event system which let's you plug in custom logic at certain stages of content management. For example, every time some content is created or modified, Sitefinity fires an event.
- You can subscribe to these events and execute your custom logic when they are fired.
- The `EventHub` service is a central place for interacting with events. It exposes simple ways for subscribing, unsubscribing and even firing events, in case you have your own custom events which you want to fire.

Code example: Introducing the EventHub service

- The EventHub service is a static helper class
- Subscribing to an event:

```
EventHub.Subscribe<IMediaContentDownloadedEvent>(this.MediaContent_Downloaded);
```

and the handler:

```
private void MediaContent_Downloaded(IMediaContentDownloadedEvent evt)
{
    // your code here
}
```

- Unsubscribing from an event:

```
EventHub.Unsubscribe<IMediaContentDownloadedEvent>(this.MediaContent_Downloaded);
```

The events

- “Before” and “after” events
  - Events that are fired before an action are suffixed with “-ing”, while events that fire after an action are sufixed with “-ed”. For example, the “UserCreating” event will fire before a user is created, while the “UserCreated” event will fire after the user has been created.
- The list of events which Sitefinity fires is quite long. Take a look at it here: http://docs.sitefinity.com/for-developers-list-of-events
- IDataEvent – a contract for event notification containing minimal information about modified items. It will be thrown on the Create, Update, and Delete actions for all content. Inside the event handler you can get information about the:
  - Event Action (Create, Update, or Delete)
  - ItemType
  - ItemId
  - ProviderName

This data is sufficient to allow you to load the corresponding manager and retrieve the actual item.

Code example: Creating custom events

- Although Sitefinity CMS provides many events raised by the built-in modules, you can also create your custom events providing hooks to your code. In order to implement a custom event, you need to follow 3 steps:

1. Create an interface – inherit from `IEvent`

```
public interface ICustomEvent : IEvent
{
    string MyCustomMessage { get; set; }
}
```

2. Implement your interface

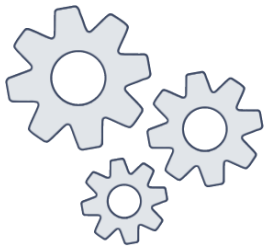
```
public class CustomEvent : ICustomEvent
{
    public string Origin { get; set; } // Origin comes from the IEvent interface
    public string MyCustomMessage { get; set; }
}
```

3. Fire your event using the EventHub

```
EventHub.Raise(new CustomEvent { MyCustomMessage = "Event info." });
```

Now everyone is able to subscribe to your event using the EventHub, like the native Sitefinity events!

Exercise: Subscribe to an event when a Blog post is being created



- Use the event to add a legal disclaimer at the end of the blog post

github:gist https://gist.github.com/ivaneftimov/63ca2a92fc9520c655b7e9f0322b35e2

Unsubscribe from events

- Unsubscribing from events is an important part of the application lifecycle
  - Sitefinity is capable of doing “soft restarts”, which may lead to subscribing to events more than once if you do not unsubscribe to your events at the end of the application lifecycle. For example, if you only subscribe to your events during initialization process of Sitefinity, and you do not unsubscribe, if Sitefinity performs a soft restart, your subscription will be executed one more time after the soft restart. This will lead to multiple executions of your handler when one event is fired.
  - Apart form the `Global.asax` file, the Initialize stage (method) of the Sitefinity modules is one of the most common places where event subscriptions are carried on. That’s why, it is important to unsubscribe from those events when the module is deactivated.
  - If the event subscription happened in Application_Start method in Global.asax, then the unsubscribing is naturally expected to happen on Application_End method.

Review of Data Layer concepts

Quick review of data layer concepts

- You bring content to the presentation layer using either the Native API or the Fluent API.
- The Native API gives you most control and flexibility; the Fluent API provides simplicity.
- The REST Services API enables you to expose Sitefinity content to client applications.
- You can use Providers to connect to different external data sources.
- You can localize content using language packs.
- Sitefinity has an event system. Every time some content is modified Sitefinity fires an event.

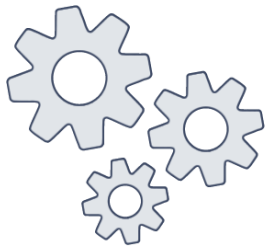
So far you have learned how to develop the presentation layer and work with content and data. You should now be able to develop complete websites and web applications in Sitefinity. Next, you will learn about a few advanced topics such as optimizing your Sitefinity application, testing, and managing configurations.

Lesson 8: Optimizing the performance of your Sitefinity application

Lesson objectives

- By the end of this lesson, you should be able to:
  - Design your code to avoid N+1 problems.
  - Implement lazy loading.
  - Describe when to use `SiteMapNode` vs `PageNode`.
  - Warm up your application.
  - Configure Sitefinity to use a CDN for scripts.

Exercise: Get pairs of Webinars and Events



Exercise: You have the Webinars module. Assume that the organization will create a new Event content item for each Webinar using the same title to promote the Webinar. You need to implement a requirement to enhance the Webinars widget to display the associated event for each Webinar.

1. Make sure you create dummy data – events and webinars.
2. Modify the WebinarsModel class with business logic to pull events with the same name.
3. Use the `EventsManager` to pull events.
4. Use a `ForEach` loop.

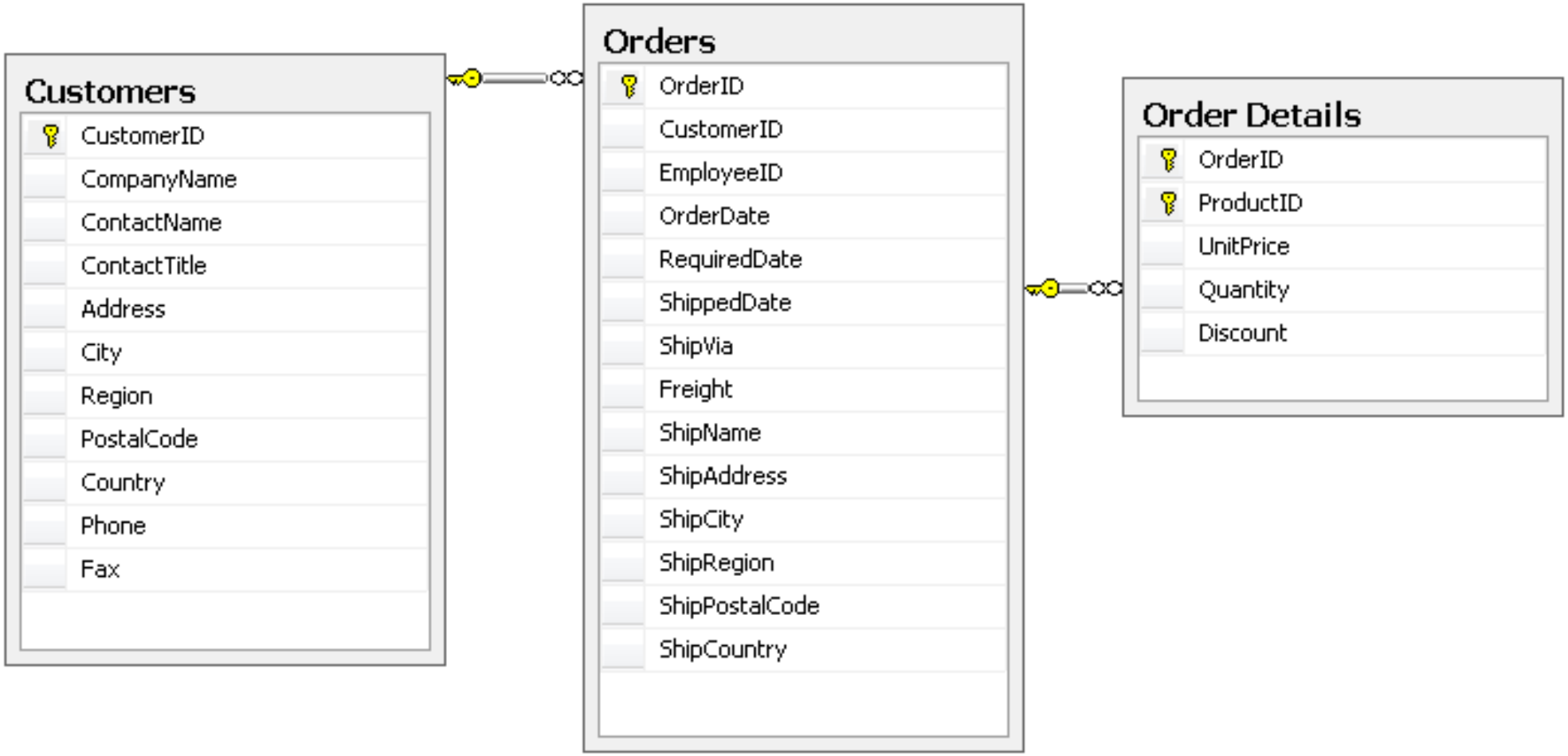
You are done?

Let's get look at the code and think for a moment

What is an N+1 problem?

- In some cases, because lazy loading is the default behavior, you may run into an N+1 problem.
- **N+1** is a data access anti-pattern where the database is accessed in a suboptimal way. For example, traversing a collection of related objects would lead to the execution of an additional query for each object in the collection. While this would still work, it is highly inefficient.
- Suppose you have a list of users and every user has related orders to itself, and every order has order details related to itself. Now if you want to display all the users with their related orders and order details, you need to traverse through all the users and access the related data. Whenever you request a related data, another query will be executed against the database. This is because the ORMs (DataAccess in case of Sitefinity), will not load user relations when you request the list of users. This behavior has its benefits, but in this case it is not optimal.
- The solution is **eager loading!**

Imagine the following data model



Code example: Code that has the N+1 problem

- If we want to display all the customers, with all their orders and order details, the most intuitive implementation would be:

```
IQueryable<Customer> customers = GetCustomers();

foreach (Customer customer in customers)
{
    Console.WriteLine("Customer Id: {0}", customer.CustomerID);
    foreach (Order order in customer.Orders)
    {
        Console.WriteLine("===Order date: {0}", order.OrderDate.ToString("d"));
        foreach (OrderDetail orderDetail in order.OrderDetails)
        {
            Console.WriteLine("Unit price: {0}", orderDetail.UnitPrice.ToString("c"));

        }
    }
}
```

- However, this code has a N+1 problem.
  - For every Customer, a subsequent call to the database will be made to retrieve its orders.
  - Also, for every order, another query will be executed to get the order details.

Code example: Avoiding an N+1 problem

- As we hinted few slides before, the solution for the N+1 problem is **eager loading**.
- The rule that *one query returning 100 rows is always faster than 100 queries returning one row each*, applies perfectly in this case. When we load the customers, we have to explicitly load the objects related to each customer.
- The easiest way to enforce eager loading is to *Include the relations*:

```
IQueryable<Customer> customers = GetCustomers()
        .Include<Order>(c => c.Order);
```

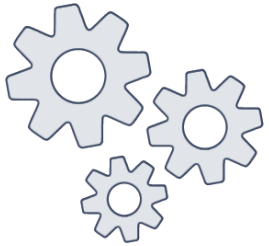
- This query will explicitly load the customers and all the orders related to them. The same principle should be followed about the `OrderDetail` objects.

- The `Inlclude` method is supported by DataAccess, which means it can be used in the context of Sitefinity.

Exercise: Fix the N+1 with the Webinars and Events

Now use Join statement to fix the sample. How would you do it?

Exercise: Use related data to associate Webinars and Events



- You can use related data to make your data modelling more flexible.
- Use the Related data to introduce entity association.
- Once done refactor the code use the related data API.

Demonstration: Avoiding an N+1 problem with Sitefinity related data

- One possible way to end up with a N+1 issue in the context of Sitefinity is when you are accessing Related Data through the API.
- Assume you have news items with relation to blog post items. If you want to list all the news items, along with their related blog post items, the query will most probably look like this:

```
foreach (var newsItem in news)
{
    var relatedPosts = newsItem.GetRelatedItems<BlogPost>("RelatedBlogPosts");
}
```

- As you may have guessed, this code has an N+1 problem.
- As you may have guessed, the solution is eager loading.
- How to implement it? – the Include method will not work in this case. Why?

Demonstration: Avoiding an N+1 problem with Sitefinity related data, continued

- The `Include()` method will not work because related data in Sitefinity is not implemented as direct relations between the objects, like in the example with the customers and orders. This means there is no navigation property List<BlogPost> of the NewsItem.
- Behind the scenes, data relations in Sitefinity are achieved through content links. They are objects which store a relation between two items, called parent and child, in order to explicitly state the relation direction.
- So, implementing something this:

```
var news = newsManager.GetNewsItems()
     .Include<BlogPost>(n => n.GetRelatedItems<BlogPost>("RelatedBlog"));
```

  will not work.
- The workaround for this issue is a query which looks scary, but the results will outwait the fear ☺.

Demonstration: Avoiding an N+1 problem with Sitefinity related data, continued

- The implementation which will explicitly load all the news items and their related data, in this case, blog post items, looks like this:
- First we get the content links which connect these two types of items:

```
var contentLinksManager = ContentLinksManager.GetManager();
var links = contentLinksManager.GetContentLinks()
     .Where(cl => cl.ParentItemType == typeof(NewsItem).FullName &&
                  cl.ComponentPropertyName == " RelatedBlogPosts");
```

- Then we join both item collections according to the content links in the previous step:

```
var news = newsManager.GetNewsItems();
var blogPosts = blogsManager.GetBlogPosts();
var relatedData = blogPosts
              .Join(links, (bp) => bp.Id, (cl) => cl.ChildItemId, (bp, cl) => new { bp, cl })
              .Join(news, (bpcl) => bpcl.cl.ParentItemId, (n) => n.Id, (bpcl, n) => new { n, bpcl.bp });
```

- The end result will be a collection of anonymous types containing related news and blogpost items.

❖ With Sitefinity 9.2 we will introduce a simpler way to do eager loading for related data.

Lazy loading and eager loading

- Lazy loading is deferring initialization of an object until it is actually needed. In the context of Sitefinity, this has two dimensions:
  - Loading a related object
  - Loading long lists of objects
- Eager loading is explicitly loading all objects, including relations.

! Lazy loading can lead to an N+1 problem in some cases.

When is lazy loading preferred?

- **Lazy loading** is a design pattern that delays the initialization of an object (or related object) until the point at which it is needed. It can contribute to efficiency in the program's operation if properly and appropriately used. For instance:
  - Loading a long list of items – better to load consequent subsets of the list as needed, because all of the items cannot fit on the screen anyway. So why to load them all at once – load them on demand. Example: paging, endless scroll
  - Loading related data – if your list items have related data, for examples details, load the details on demand, if they are not needed in the list view. Example: Master -> Detail view in Sitefinity – News widget loads list of news with title and dates, and subsequently loads additional data if you click on a news item in the list.

Code example: Lazy loading for collections

- Paging
  - Get the first 50 items first
    ```
    var manager = NewsManager.GetManager();
    var news = manager.GetNewsItems().Skip(0).Take(50);
    ```
  - Then on the next page, get the next 50 items
    ```
    var news = manager.GetNewsItems().Skip(50).Take(50);
    ```

Demonstration: Widget behavior with lazy loading and without lazy loading

PageNode and PageSiteNode

- **PageNode** is the data model of a Sitefinity page. It contains all the properties of a page and its data. You can use it to create a page or manipulate its properties.
- **PageSiteNode** contains a subset of the properties of a PageNode. When Sitefinity initializes, it instantiates the sitemap, which contains a PageSiteNode for every PageNode. You can use a PageSiteNode to quickly get the most common page properties without having to access the database.
  - For example, you can use the PageSiteNode to create a custom page selector where you only need the page title and page ID.
- You can only use a PageSiteNode for published pages since the sitemap only contains published pages.

Code example: Using the `PageSiteNode`

- The sitemap that Sitefinity creates during initialization is an ASP.NET structure. Sitefinity just provides the data for this structure.
- In order to use this sitemap structure, you need to use the Sitemap provider:

```
SiteMapProvider siteMapProvider = SiteMapBase.GetSiteMapProvider("FrontendSiteMap");
SiteMapNode currentPage = siteMapProvider.CurrentNode; // SiteMapNode is .NET object

// Cast is needed to get Sitefinity specific properties
PageSiteNode currentPage = siteMapProvider.CurrentNode as PageSiteNode;
```

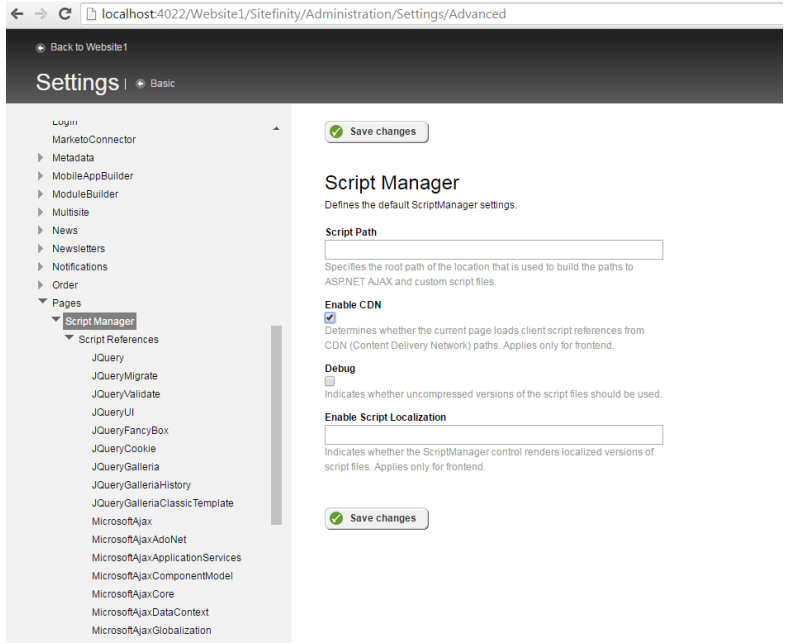
Warming up your application before deployment

- When ASP.NET pages are first loaded, they are compiled and their data is retrieved from the database. The output of these operations is stored in the server cache, if it is enabled.
- So the first request will take time to load.
- As a best practice, you should request your pages* after deployment or after application restart. This will enable the page to be served quickly (from the cache) for subsequent requests.

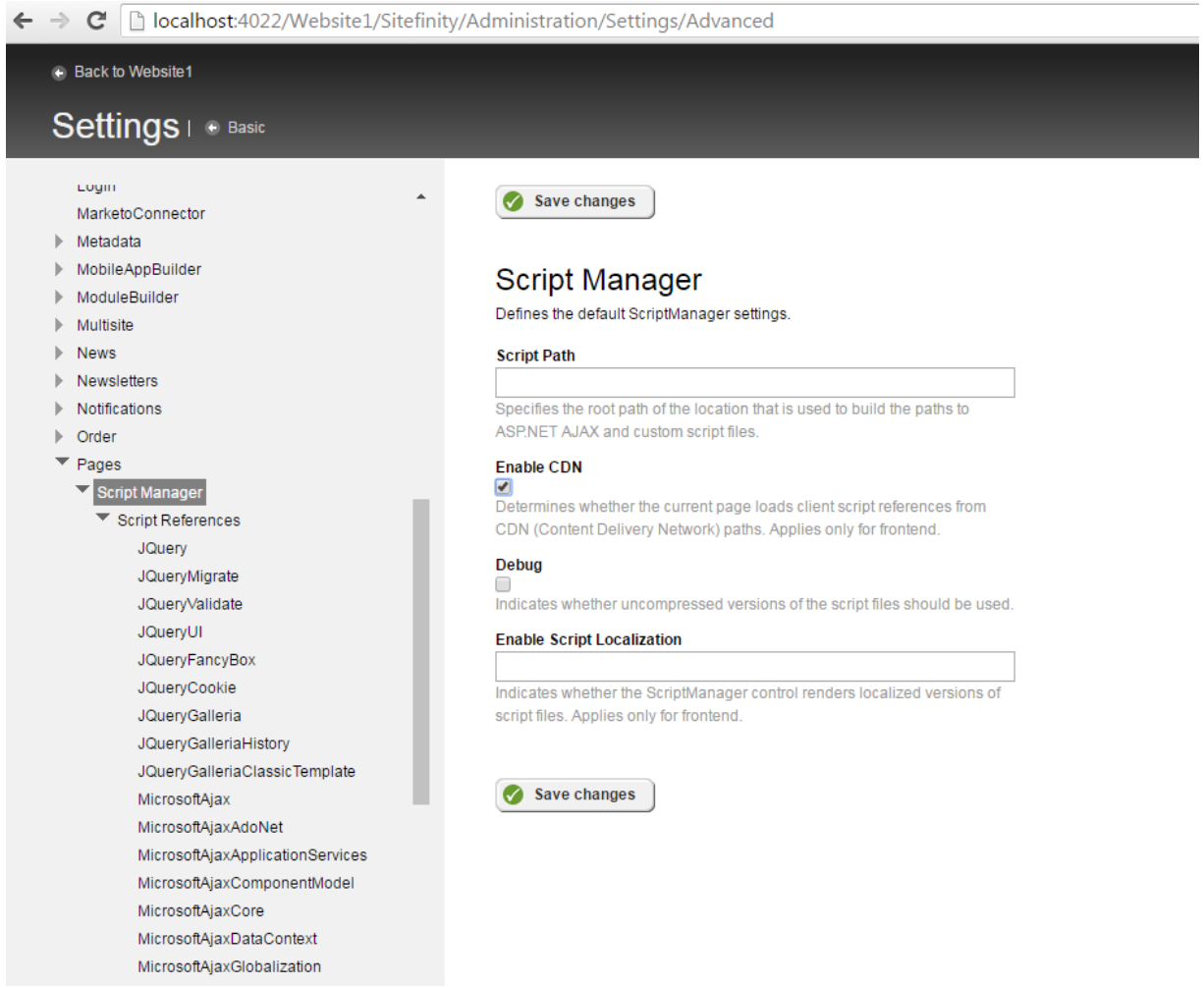
* Generally, you would want to warm up the most commonly used pages.

Configuring Sitefinity to use a CDN for scripts

- You can configure scripts that are commonly used by Sitefinity to be served from a CDN instead of your application.
- In this way, you can reduce the number of requests sent to your application and improve its performance.
- To set up Sitefinity to use a CDN instead of locally stored scripts, use these settings:



Demonstration: Configuring Sitefinity to use a CDN for scripts



Demonstration: Configuring Sitefinity to use a CDN for content libraries

Lesson 9: Managing configurations

Lesson objectives

- By the end of this lesson, you should be able to:
  - Use the `ConfigManager` to get or set configurations.
  - Use the `Config` helper class to get configurations.
  - Create and register a custom configuration.

Code example: Using the `ConfigManager` to get or set configurations

- The `ConfigManager` syntax is the same as the other content managers such as `NewsManager`, `BlogsManager`, etc:

```
var configManager = ConfigManager.GetManager();
```

- To get a configuration (`ConfigSection`) use the `GetSection` method:

```
var newsConfig = configManager.GetSection<NewsConfig>();
```

- To update a configuration, you need to first get it, make your changes, and then save it using the `SaveSection` method:

```
var newsConfig = configManager.GetSection<NewsConfig>();
newsConfig.DefaultProvider = "ExternalNews";
configManager.SaveSection(newsConfig);
```

Code example: `Config` helper class for easy access of common operations

- Here is a simpler way to get configurations using the `Config` helper class in case you only need to read the configurations:

```
var newsConfig = Config.Get<NewsConfig>();
```

- Helper method for registering a custom configuration (you'll learn more about this soon):

```
Config.RegisterSection<MyConfiguration>();
```

Code example: Creating a custom configuration

- Inherit from `ConfigSection` class
- Add your configuration properties

```
public class MyConfiguration : ConfigSection
{
    [ConfigurationProperty("name", DefaultValue = "Arial", IsRequired = true)]
    public string Name
    {
        get { return (String)this["name"]; }
        set { this["name"] = value; }
    }
}
```

Registering a custom configuration

- There are two ways to register your custom configuration
  1. During application start
  2. During your custom module initialization*

Usually a configuration is related to a module. For example, the News module has `NewsConfig`. Often, when you have your own custom module that encompasses some logic—for example a custom module that creates your custom dashboard in Sitefinity—you will have a custom configuration associated with this module.

Code example: Registering a custom configuration

- Option 1: During application start
  - Register your configuration when Sitefinity initializes. When Sitefinity initializes all its core components and modules, it will fire the Initialized event. You can subscribe to that event and register your configuration when it is fired.
    - First, subscribe for the Initialized event on Application_Start in the Global.asax file:

```
protected void Application_Start(object sender, EventArgs e)
{
      Bootstrapper.Initialized += this.Bootstrapper_Initialized;
}
```

    - Then, register your configuration in the `Bootstrapper_Initialized` handler:

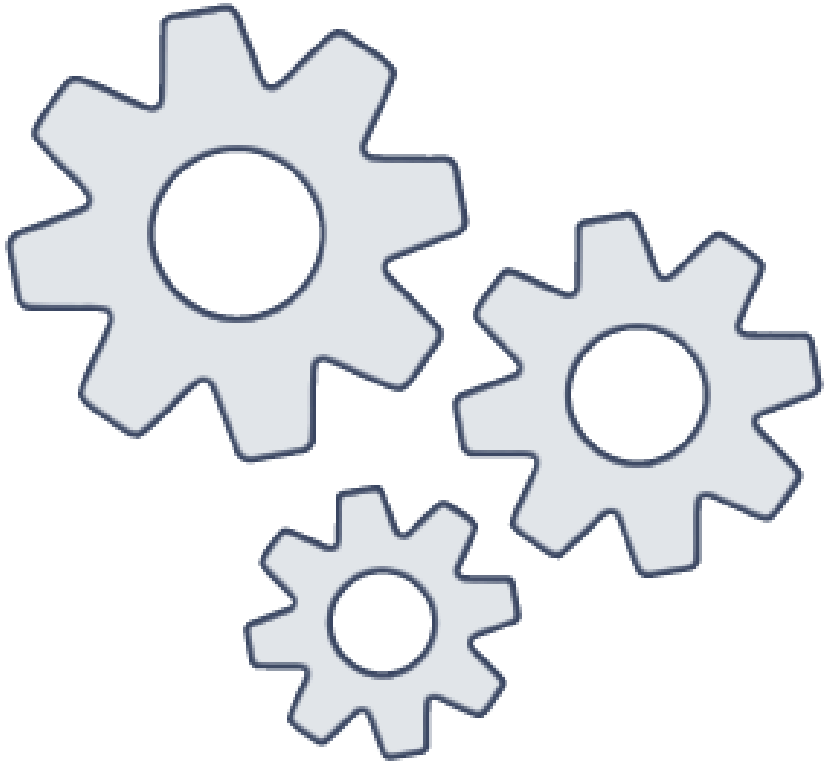
```
private void Bootstrapper_Initialized(object sender, ExecutedEventArgs e)
{
      Config.RegisterSection<MyConfiguration>();
}
```

Code example: Registering a custom configuration

- Option 2: During your custom module initialization
  - Register your configuration during the initialization phase of your custom module. `Initialize` is an abstract method that every Sitefinity module can override in order to execute its initialization logic. Registering a configuration should part of that initialization logic:

```
public override void Initialize(ModuleSettings settings)
{
    base.Initialize(settings);

    App.WorkWith()
    .Module(Constants.ModuleName)
    .Initialize()
    .Configuration<MyConfiguration>();
}
```

Excercise: Creating and registering a custom configuration



Lesson 10: Testing your code

Lesson objectives

- By the end of this lesson, you should be able to:
  - Understand the Sitefinity Service Locator mechanism.
  - Use the Dependency Injection pattern for your MVC controllers.
  - Mock Sitefinity content managers.
  - Mock content item classes.
  - Test your widget.

❖ This lesson will focus on unit testing. We will not use a mocking framework for this course—we will create the mocks ourselves. However, you may use one when you write your tests.

Sitefinity *Service Locator* mechanism

- Sitefinity uses the Service Locator pattern for most of its components.
- We use an abstraction over Microsoft's Unity Container for this purpose.
  - The advantage of using a container is that it provides the ability to register and resolve dependencies that might be used by your custom code as well as by Sitefinity.
- The entry point for interacting with the Sitefinity unity container is the `ObjectFactory` class.
- `ObjectFactory` allows you to initialize your own container.
  - This gives us the ability to register our own implementation of Sitefinity components.
  - In terms of testing, it gives us the ability to register mocks for the dependencies in our code and inject our custom container when the tests are running.
  - For example, we can create mocks of the Sitefinity managers used in our code, register them in a custom container and run our code with this container.
  - This way we can abstract the Sitefinity API logic and test our own implementations.

Code example: Use your version of a manager when running a test

- First, you need to create an instance of our Unity container

```
UnityContainer mockedContainer = new UnityContainer();
```

- Register your own implementation of a manager – in this case the `NewsManager`

```
mockedContainer
    .RegisterType<NewsManager, MyNewsManager>("ActualProviderName". ToUpperInvariant());
```

- Run your code (testing code) with the custom container you just created

```
ObjectFactory.RunWithContainer(mockedContainer, () =>
{
   // your testing code goes here
});
```

- You should resolve your manager in the following way, even in the actual code in the controllers

```
var manager = ObjectFactory.Resolve<NewsManager>("ActualProviderName". ToUpperInvariant());
```

! The standard way of getting manager is not working with the Unity container

```
var newsManager = NewsManager.GetManager();
```

Code example: Create a manager mock

- Now that we know how to inject our own custom manager, let's see the important aspects of creating that custom version (the mock):
  - Naturally, we need to inherit the Sitefinity manager that we want to mock:
    ```
    public class MyNewsManager : NewsManager
    ```
  - Override the default constructor:
    ```
    public MyNewsManager() : base(MyNewsManager.ProviderName) { }
    ```
    - `MyNewsManager.ProviderName` is just a constant holding provider name
  - Override the `Initialize` method *and make it do nothing.*
  - Override the `SetProvider` method *and make it do nothing.*
    - These two methods are executed when the manager is initialized, but since we don't want an actual initialization of the manager, we should make them do nothing when invoked.
  - Mock the methods which we use in our code:
    - For example the `GetNewsItems()` method

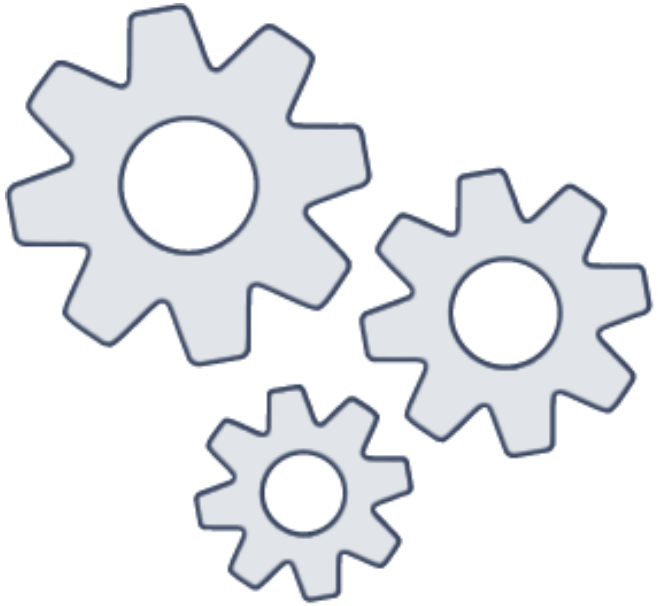
Code example: Create a content item mock

- Content items are in fact data model classes, so why we need to mock them?
- Content items in Sitefinity are a lot more complex than a simple data model, mostly due to the `Lstring` properties, such as Title, `UrlName`, etc.
  - `Lstring` properties are complex data models that allow us to store strings in different cultures.
  - The `Lstring` type is a necessity for Sitefinity's multilingual capabilities, but it makes testing challenging. This is why we need to mock the content items that we use in the tests, so that we can override `Lstring` properties and simplify their implementation—for example by making them return regular strings. Here is an example of a mock of NewsItem, which overrides the `Title` property and makes it suitable for testing.

```
public class MyNewsItem : NewsItem
{
    public override Lstring Title
    {
        get { return this.title; }
        set { this.title = value; }
    }
    private string title;
}
```

Guided exercise: Let's combine the previous three slides and create our first test

- We will test `Index` action of the `BreakingNewsController`.
  - *We will simplify the `Index` action a bit – it will only get the first item and assigns its title to the model*



github:gist https://gist.github.com/ivaneftimov/6180fb91e6ac3e43f4a5e431d89a70c7

Dependency Injection in MVC Controllers

- Sitefinity allows you to use non-default constructors for your MVC controllers and inject their dependencies using Ninject.
  - Ninject is an open source dependency injector for .NET.
- By decoupling the dependencies for the controller, you can easily mock them and test the code of your controller.

Dependency Injection in MVC Controllers

- ASP.NET MVC uses `DefaultControllerFactory` to instantiate and invoke its controllers.
  - This factory works only with the default constructors of the controllers
  - The simplified flow of invoking an MVC controller may be represented as follows:

```
Request --> Routing System --> Controller Factory --> Invoke Controller
```

- ASP.NET MVC allows you to plug in your own controller factory.
- Sitefinity Feather takes advantage of this option in order to plug in the `FrontendControllerFactory.`
- Sitefinity Feather uses the Service Locator mechanism of Sitefinity in order to register this factory into your Sitefinity application on runtime.
- The `FrontendControllerFactory` uses **Ninject** in order to resolve the controllers, hence the ability to inject the controllers' dependencies also using **Ninject**. *Moreover these dependencies may be injected through a non-default constructor of the controllers*.

Demonstration: Test the BreakingNewsController using DI

- Let's try to implement what we have discussed before – write a test for `BreakingNewsController`'s `Index()` action with help of **DI** and **Ninject**.
  1. Create a custom controller factory inheriting from `FrontendControllerFactory` and override the `GetControllerInstance` method
  2. Add From->To type mappings for the dependencies `Bind<INewsManagerWrapper>().To<NewsManagerWrapper>();`
  3. Register your new controller factory in Sitefinity's Bootstrapped event (using the `ObjectFactory`), resolve it, and set it as a controller factory to be used by your application.

     ? Why register and then resolve? Why not just create a new instance?
  4. Create a new constructor for the `BreakingNewsController` which will allow injecting of the dependencies

github:gist https://gist.github.com/ivaneftimov/1a4018e4396d856eadc71402c5a76d5b

Test the BreakingNewsController using DI | continued

? Anything wrong with the BreeakingNewsController constructor?

```
public BreakingNewsController(NewsManager newsManager)
{
      this.newsManager = newsManager;
}
```

This is not how we initialize the news manager.

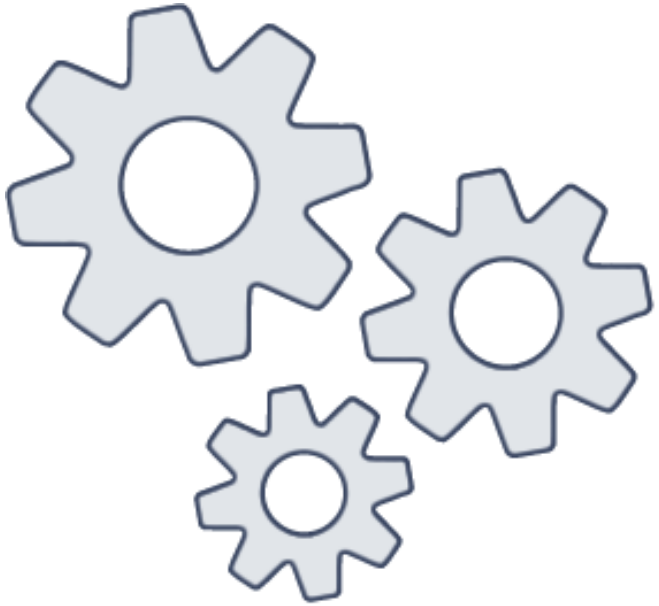
? *How do we use the `GetManager` method when using DI?*

- Create wrappers for Sitefinity managers

github:gist https://gist.github.com/ivaneftimov/dd93bb043f1ce853a2f88dbc64ce474c

Exercise: Test the `WebinarController` using DI

- Follow the same steps as for the `BreakingNewsController`



Review and Q&A

Advanced Sitefinity Development

Thank You!