Webben vNext

Lab 1 – Web development with .NET Core

# Intro

In this lab we will start creating the ASP.NET Core Web Application that we will continue to build and improve throughout the workshop. We will add the needed interface, implementation, controller and views to be able to save files locally, test the controller and render an ASP.NET Core MVC View.

Techniques and concepts you will use in this lab:

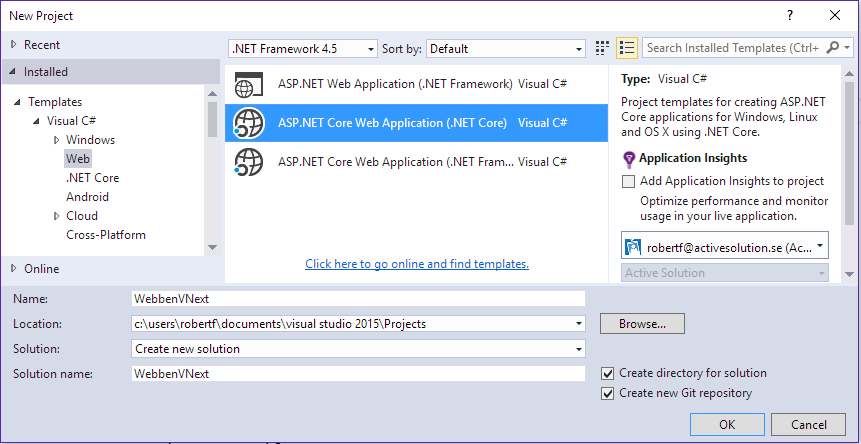
* ASP.NET Core and configuration
* Interfaces and dependency injection
* Inversion of Control (IOC) Container
* Mock framework – Moq

# Lab Instructions

## Create and explore the Web Application

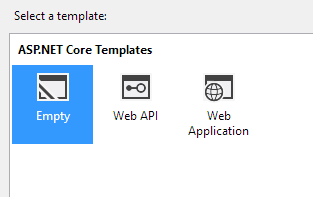
### Create the project

1. Open Visual Studio 2015 and choose File -> New Project and choose **ASP.NET Core Web Application (.NET Core)**.



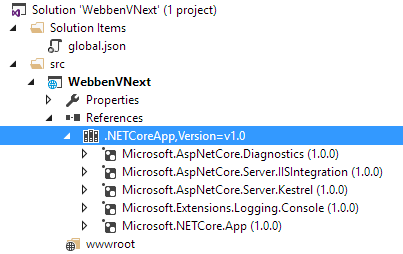
**Make sure that the option “Create new Git repository” is checked.**

(Note that there is also an option to create ASP.NET Core Web Applications for the full Framework).

1. Name the project **WebbenVNext**.
2. Choose the empty template to create an ASP.NET Core Application without any controllers or views.   
     
     
   We will not yet host the application in Azure so make sure you uncheck the **Host in the Cloud** checkbox under the Microsoft Azure choice.

### Examine the project

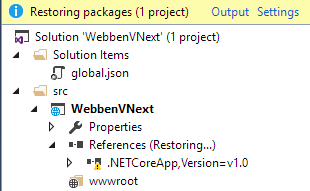
1. Open **Program.cs** and notice the calls to the different extension methods of the WebHostBuilder: UseKestrel, UseContentRoot, UseIISIntegration and UseStartup.
2. Expand the References node in the Solution Explorer and expand the .NET Core App, Version=v1.0 reference:



1. Open the **project.json** file and notice that this JSON structure defines the references displayed in Visual Studio.
2. Add the references to ASP.NET MVC, Razor Tools and ASP.NET Static Files to the project.json file. In the dependencies node, right after Microsoft.NETCore.App add the code marked with yellow:

|  |
| --- |
| {  "dependencies": {  "Microsoft.NETCore.App": {  "version": "1.0.0",  "type": "platform"  },  "Microsoft.AspNetCore.Mvc": "1.0.1",  "Microsoft.AspNetCore.Razor.Tools": {  "version": "1.0.0-preview2-final",  "type": "build"  },  "Microsoft.AspNetCore.StaticFiles": "1.0.0",  "Microsoft.AspNetCore.Diagnostics": "1.0.0",  "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",  "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",  "Microsoft.Extensions.Logging.Console": "1.0.0"  },  ... |

1. When you save the project.json Visual Studio should restore the new packages automatically:



1. Open Startup.cs and replace the code with the following:

|  |
| --- |
| using Microsoft.AspNetCore.Builder;  using Microsoft.AspNetCore.Hosting;  using Microsoft.Extensions.DependencyInjection;  using Microsoft.Extensions.Logging;  namespace WebbenVNext  {  public class Startup  {  public void ConfigureServices(IServiceCollection services)  {  services.AddMvc();  services.AddOptions();  }  public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)  {  app.UseStaticFiles();  app.UseMvcWithDefaultRoute();  }  }  } |

## Create the Interface, ViewModel, View and Controller for file handling

### Create the interface

Let’s start building the application by adding an interface for handling the saving and fetching of files from storage.

1. Add a new folder called **Storage**.

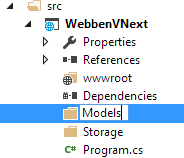


1. Add a new interface in the Storage folder called **IBlobs**:

|  |
| --- |
| using System.Collections.Generic;  using System.IO;  using System.Threading.Tasks;  namespace WebbenVNext.Storage  {  public interface IBlobs  {  Task Save(string name, Stream fileStream);  Task<IEnumerable<string>> GetAllBlobUrls();  }  } |

### Create the ViewModel that the view will use

1. Add a new folder called **Models**

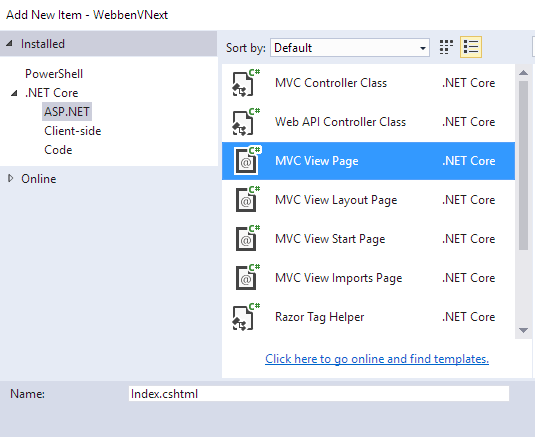


1. Add a new class called **HomeViewModel**

|  |
| --- |
| using System.Collections.Generic;  namespace WebbenVNext.Models  {  public class HomeViewModel  {  public HomeViewModel(IEnumerable<string> blobUrls)  {  BlobUrls = blobUrls;  }  public IEnumerable<string> BlobUrls { get; }  }  } |

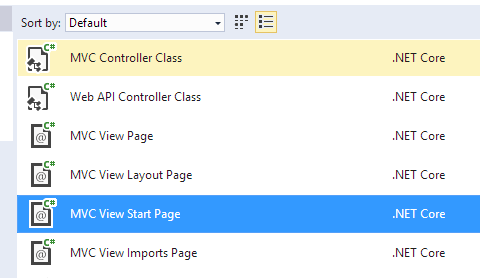
### Create the Views

1. Add a folder called **Views**.
2. In the Views folder – add a sub folder called **Home**.
3. In the Home folder add a **MVC View Page** using New -> New item -> MVC View Page. Leave the default name **Index.cshtml**.



Replace the template code with the following:

|  |
| --- |
| @model WebbenVNext.Models.HomeViewModel  <h1>Webben vNext</h1>  <div class="row">  <div class="col-md-6">  <h2>Ladda upp bild</h2>  <div class="card">  <div class="card-block">  <form method="post" asp-action="Index" asp-controller="Home" enctype="multipart/form-data">  <input type="file" name="files" multiple accept="image/\*" id="files" />  <input type="submit" value="Upload" />  </form>  </div>  </div>  </div>  <div class="col-md-6">  <h2>Bildbibliotek</h2>  @foreach (var url in Model.BlobUrls)  {  <div class="card">  <img src="@url" class="img-fluid" />  <div class="card-block">  <p class="card-text">@url</p>  </div>  </div>  }  </div>  </div> |

1. We will need a couple of additional view files for the MVC View Engine, first – in the View folder – add a **MVC View Start Page** with New -> New item:  
   

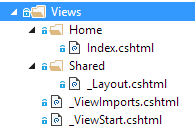
Leave the default name and content.

1. Again in the View folder - add a **MVC View Imports Page.** Leave the default name and add the following code to the page:

|  |
| --- |
| @using WebbenVNext  @addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers |

1. Create a new folder under Views called **Shared**.
2. In the Shared folder – add a **MVC View Layout Page**. Leave the default name and replace the template content with the following:

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8" />  <meta name="viewport" content="width=device-width, initial-scale=1.0" />  <title>Webben vNext</title>  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.5/css/bootstrap.min.css" />  </head>  <body>  <div class="container">  @RenderBody()  </div>  </body>  </html> |

1. Your Views folder structure should now look like this:  
     
   

### Create the Controller

1. Add folder called **Controllers.**
2. Add a class in the Controllers folder called **HomeController**

|  |
| --- |
| using System.Collections.Generic;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Http;  using Microsoft.AspNetCore.Mvc;  using WebbenVNext.Models;  using WebbenVNext.Storage;  namespace WebbenVNext.Controllers  {  public class HomeController : Controller  {  private readonly IBlobs \_blobs;  public HomeController(IBlobs blobs)  {  \_blobs = blobs;  }  public async Task<IActionResult> Index()  {  var blobUrls = await \_blobs.GetAllBlobUrls();  var viewModel = new HomeViewModel(blobUrls);  return View(viewModel);  }  [HttpPost]  public async Task<IActionResult> Index(ICollection<IFormFile> files)  {  foreach (var file in files)  {  if (file.Length > 0)  {  await \_blobs.Save(file.FileName, file.OpenReadStream());  }  }  return RedirectToAction("Index");  }  }  } |

Verify that the application builds.

## Create a local file storage implementation of the IBlob interface

In the storage folder add a new class named **LocalBlobs**:

|  |
| --- |
| using System.Collections.Generic;  using System.IO;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Hosting;  namespace WebbenVNext.Storage  {  public class LocalBlobs : IBlobs  {  private const string FilesRootDirectoryName = "uploads";  private readonly IHostingEnvironment \_environment;  public LocalBlobs(IHostingEnvironment environment)  {  this.\_environment = environment;  }  public async Task Save(string name, Stream file)  {  var filesFolder = Path.Combine(\_environment.WebRootPath, FilesRootDirectoryName);  if (!Directory.Exists(filesFolder))  {  Directory.CreateDirectory(filesFolder);  }  using (var fileStream = new FileStream(Path.Combine(filesFolder, name), FileMode.Create))  {  await file.CopyToAsync(fileStream);  }  }  public Task<IEnumerable<string>> GetAllBlobUrls()  {  var filesFolder = Path.Combine(\_environment.WebRootPath, FilesRootDirectoryName);  if (!Directory.Exists(filesFolder))  {  var empty = new string[] { };  return Task.FromResult(empty.AsEnumerable());  }  var directoryInfo = new DirectoryInfo(filesFolder);  var blobUrls = directoryInfo.GetFiles()  .OrderByDescending(p => p.CreationTime)  .Select(x => x.Name)  .Select(GetUrl);  return Task.FromResult(blobUrls);  }  private string GetUrl(string name)  {  var path = Path.Combine("/", FilesRootDirectoryName, name);  var url = path.Replace(@"\", "/");  return url;  }  }  } |

## Create a test project with a test using the Moq mocking framework

### Create the test project

1. Add a new solution folder called **test**.
2. In the test folder - add a new project with Add new project -> Visual C# -> Class Library (.NET Core). Name the project **WebbenVNext.Tests**
3. Open project.json and replace the generated code with the following:

|  |
| --- |
| {  "version": "1.0.0-\*",  "testRunner": "mstest",  "dependencies": {  "NETStandard.Library": "1.6.0",  "MSTest.TestFramework": "1.0.4-preview",  "dotnet-test-mstest": "1.1.1-preview",  "Moq": "4.6.38-alpha",  "WebbenVNext": "1.0.0-\*"  },  "frameworks": {  "netcoreapp1.0": {  "imports": [  "dnxcore50",  "portable-net45+win8"  ],  "dependencies": {  "Microsoft.NETCore.App": {  "version": "1.0.1",  "type": "platform"  }  }  }  }  } |

1. Rename the file Class1.cs to **HomeControllerTests.cs**
2. Replace the code in HomeControllerTests with the following:

|  |
| --- |
| using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;  using Microsoft.VisualStudio.TestTools.UnitTesting;  using Moq;  using WebbenVNext.Controllers;  using WebbenVNext.Models;  using WebbenVNext.Storage;  namespace WebbenVNext.Tests  {  namespace WebbenVNext.Tests  {  [TestClass]  public class HomeControllerTests  {  [TestMethod]  public async Task HomeIndexSholdReturnFilesFromBlobProvider()  {  var blobUrls = new[] { "/file1.png", "/file2.jpg" };  var mockedBlob = new Mock<IBlobs>();  mockedBlob  .Setup(x => x.GetAllBlobUrls())  .Returns(Task.FromResult(blobUrls.AsEnumerable()));  var homeController = new HomeController(mockedBlob.Object);  var homeIndexViewResult = await homeController.Index() as ViewResult;  var homeIndexViewModel = homeIndexViewResult?.Model as HomeViewModel;  var homeIndexViewModelBlobUrls = homeIndexViewModel?.BlobUrls.ToList();  Assert.IsNotNull(homeIndexViewModelBlobUrls);  Assert.AreEqual(blobUrls[0], homeIndexViewModelBlobUrls?[0]);  Assert.AreEqual(blobUrls[1], homeIndexViewModelBlobUrls?[1]);  }  }  }  } |

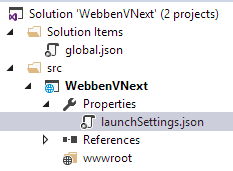
1. Put a breakpoint on the first line in the method HomeIndexSholdReturnFilesFromBlobProvider and debug the test with Test -> Debug. Notice that the test uses a mock of the file storage to be able to test the code of the controller in isolation
2. Verify that the test passes.

## Configure an Exception handling page for the application

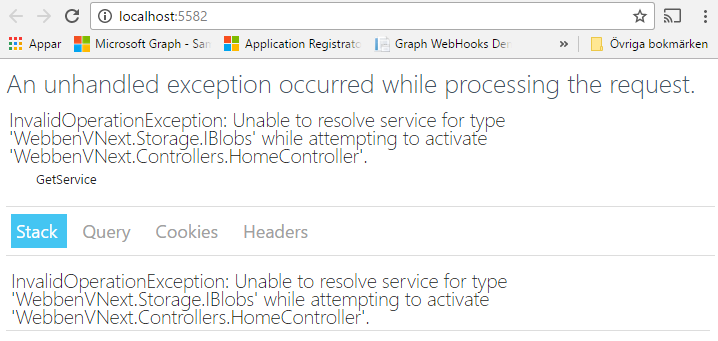
Right now the application will not show any error information page at all since it has not been configured yet. Let’s add an exception page for debugging purposes.

1. Edit the **startup.cs** and add the code in yellow *before* the existing code in the Configure method:

|  |
| --- |
| public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)  {  if (env.IsDevelopment())  {  app.UseDeveloperExceptionPage();  }  app.UseStaticFiles();  app.UseMvcWithDefaultRoute();  } |

1. The application will now show a developer friendly error message for debugging purposes if the env.IsDevelopment is set.
2. Examine the **launchSettings.json** file under the properties node in the project:  
    

This is where Visual Studio stores settings from the projects debug profiles – for example to handle different environment variables.

1. Now let’s hit F5 and run the application.
2. The following error page should show:  
   

This is because we have not yet told ASP.NET Core how to actually resolve the reference to the interface IBlobs.

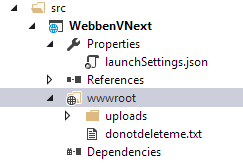
## Configure ASP.NET Core to resolve the dependency to LocalBlobs

1. Open **Startup.cs** and add the code in yellow below to configure the built in IOC (Inversion of Control) container to resolve IBlobs references to the LocalBlobs implementation:

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddMvc();  services.AddOptions();  // Local blobs  services.AddTransient<IBlobs, LocalBlobs>();  } |

AddTransient specifies the lifetime of the service to per request.

## Test the application

Before we can test the application there is one more thing we have to do. Since the **wwwroot** folder is empty so far, Visual Studio will not create the folder when we debug the project. So let’s add an empty text file called **donotdeleteme.txt** to wwwroot to prevent this:  
  


Run the application and verify that the application now works by uploading a picture.

# Next Step

In the next lab you will deploy your application to Microsoft Azure and store the images in Azure Blob Storage instead of locally.

## Challenges

In case you have time left, here are some challenges that you can try using the project you just built:

* Explore the command line tool for .NET Core. Navigate to the project folder for the application, run a command line prompt from there and examine the **dotnet** command (dotnet help will show you all the available options). Try dotnet run for example.
* Use the command line to run the test from the command line. Navigate to the test project from the command line and run **dotnet test**

## Related sessions during Techdays

The following sessions will cover some of the topics in this lab, allowing you dig even deeper into these areas:

Inblick, försmak, in I det nya Microsoft med ASP.NET Core, Fredrik Normén  
<http://tdswe.se/session/inblick-forsmak-in-i-det-nya-microsoft-med-net-core-level-200/>

ASP.NET Core Deep Dive, Rachel Appel,  
<http://tdswe.se/session/asp-net-core-deep-dive-level-300/>