ExtCore Documentation

Release 0.1.0

Dmitry Sikorsky

Jun 13, 2021

Contents

ExtCore is free, open source and cross-platform framework for creating modular and extendable web applications based on ASP.NET Core. It is built using the best and the most modern tools and languages (Visual Studio 2019, C# etc). Join our team!

CHAPTER 1

Contents

1.1 Introduction

ExtCore is free, open source and cross-platform framework for creating modular and extendable web applications based on ASP.NET Core. It is built using the best and the most modern tools and languages (Visual Studio 2019, C# etc). Join our team!

ExtCore allows you to build your web applications from the different independent reusable modules or extensions. Each of these modules or extensions may consist of one or more ASP.NET Core projects and each of these projects may include everything you want as any other ASP.NET Core project. You don't need to perform any additional actions to make it all work: any ASP.NET Core project can be used as an ExtCore-based web application extension by default. Controllers, view components, views (precompiled), static content (added as resources) are resolved automatically. These projects may be then added to the web application in two ways: as direct dependencies (as source code or NuGet packages) or by copying compiled DLLs to the Extensions folder. ExtCore supports both of these options out of the box and at the same time.

Furthermore, any project of the ExtCore-based web application is able to discover the types that are defined inside all the projects (optionally using the predicates for assemblies filtering) and to get the implementations or instances of that types.

Any module or extension can execute its own code during the web application initialization and startup. You can use priorities to specify the correct order of the calls. This feature might be used for configuration, to register services etc.

ExtCore consists of two general packages and four optional basic extensions.

1.1.1 General Packages

ExtCore general packages are:

- ExtCore.Infrastructure;
- ExtCore.WebApplication.

ExtCore.Infrastructure

This package describes such basic shared things as `IExtension` interface and its abstract implementation – `ExtensionBase` class. Also it contains `ExtensionManager` class – the central element in the ExtCore types discovering mechanism. Most of the modules or extensions need this package as dependency in order to be able to discover types, extensions etc.

ExtCore.WebApplication

This package describes basic web application behavior with `Startup` abstract class. This behavior includes modules and extensions assemblies discovering, `ExtensionManager` initialization etc. Any ExtCore web application must inherit its `Startup` class from `ExtCore.WebApplication.Startup` class in order to work properly. Also this package contains `IAssemblyProvider` interface and its implementation – `AssemblyProvider` class which is used to discover assemblies and might be replaced with the custom one.

1.1.2 Basic Extensions

ExtCore basic extensions are:

- ExtCore.FileStorage;
- ExtCore.Data;
- ExtCore.Mvc;
- ExtCore.Events.

ExtCore.FileStorage

This extension allows developer to work with a file storage through the abstraction layer and easily replace, let's say, file system storage with the Dropbox or Azure Blob Storage ones without changing any code.

ExtCore.Data

By default, ExtCore doesn't know anything about data, but you can use ExtCore.Data extension to have unified approach to working with data and the single data storage context among all the extensions. Data storage might be represented by a database, a web API, a file structure or anything else.

Currently ExtCore.Data supports MySQL, PostgreSql, SQLite, and SQL Server with Dapper or Entity Framework Core as ORM. You can add your own database or ORM support.

ExtCore.Mvc

By default, ExtCore web applications are not MVC ones. MVC support is provided for them by ExtCore.Mvc extension. This extension initializes MVC, makes it possible to use controllers, view components, views (added as resources and/or precompiled), static content (added as resources) from other extensions etc.

Also, it allows extension to register custom routes in a specific order.

ExtCore.Events

It can be used by the extension to notify the code in this or any other extension about some events.

1.2 Getting Started

1.2.1 How to Start Using the ExtCore Framework

1. Add dependency on the ExtCore.WebApplication NuGet package to your main web application project.

2. Optionally (if your web application uses MVC), add dependency on the ExtCore.Mvc NuGet package as well.

3. Call the `services.AddExtCore(extensionsPath)` extension method inside the `ConfigureServices` method of your `Startup` class:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddExtCore(this.extensionsPath);
}
```

4. Call the `applicationBuilder.UseExtCore()` extension method inside the `Configure` method of your `Startup` class:

```
public void Configure(IApplicationBuilder applicationBuilder)
{
  applicationBuilder.UseExtCore();
}
```

Your modular and extendable web application is ready to use. Now you can create an extension or use the existing ones. ExtCore automatically discovers types, controllers, views, styles, scripts, any other static content. Entities and repositories are also discovered automatically. You can use events to notify other extensions about something important.

1.2.2 How to Create an Extension

1. Create an empty .NET Razor class library project.

2. Put some classes, controllers, views, styles, scripts, and any other static content you want there. Static content must be embedded into the resulting assembly as resources using the following line inside the project file (.csproj):

```
<EmbeddedResource Include="Styles\**;Scripts\**" />
```

3. Build your extension project and copy the resulting assembly's DLL file into the extensions folder of your main web application (or you can simply add implicit dependency on the created extension project, or on the NuGet package).

4. If your extension needs to execute some code inside the `ConfigureServices` method of the main web application's `Startup` class (for example, to register some service inside the DI), you can implement the `ExtCore.Infrastructure.Actions.IConfigureServicesAction` interface.

5. If your extension needs to execute some code inside the `Configure` method of the main web application's `Startup` class (for example, to configure the web application's request pipeline), you can implement the `ExtCore.Infrastructure.Actions.IConfigureAction` interface.

It is recommended to follow the unified extension structure when developing your own extensions.

Samples

Please take a look at our samples on GitHub:

- Full-featured ExtCore 6.0.0 framework sample web application;

- ExtCore framework 6.0.0 sample simplest web application;
- ExtCore framework 6.0.0 sample web application that uses file storage;
- ExtCore framework 6.0.0 sample MVC web application;
- ExtCore framework 6.0.0 sample web application that uses a database;
- ExtCore framework 6.0.0 sample web application that uses a Identity;
- ExtCore framework 6.0.0 sample web application with modular UI;
- ExtCore framework 6.0.0 advanced sample web application with modular UI;
- ExtCore framework 6.0.0 advanced sample accounting web application;
- ExtCore framework 6.0.0 sample web application that registers a service inside the extension;
- ExtCore framework 6.0.0 sample web application that uses the events;
- ExtCore framework 6.0.0 sample API web application.

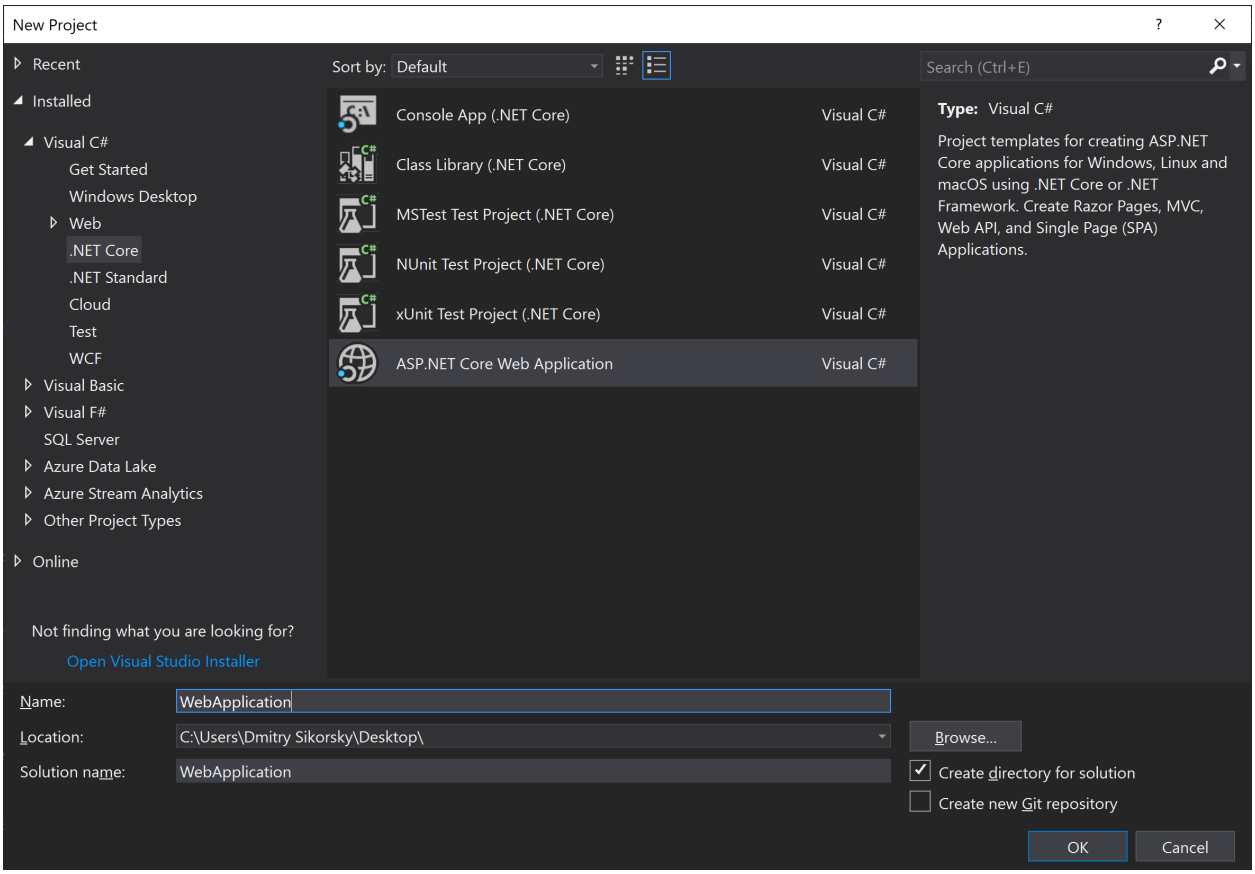
You can also download our ready to use full-featured sample. It contains everything you need to run ExtCore-based web application from Visual Studio 2019, including SQLite database with the test data.

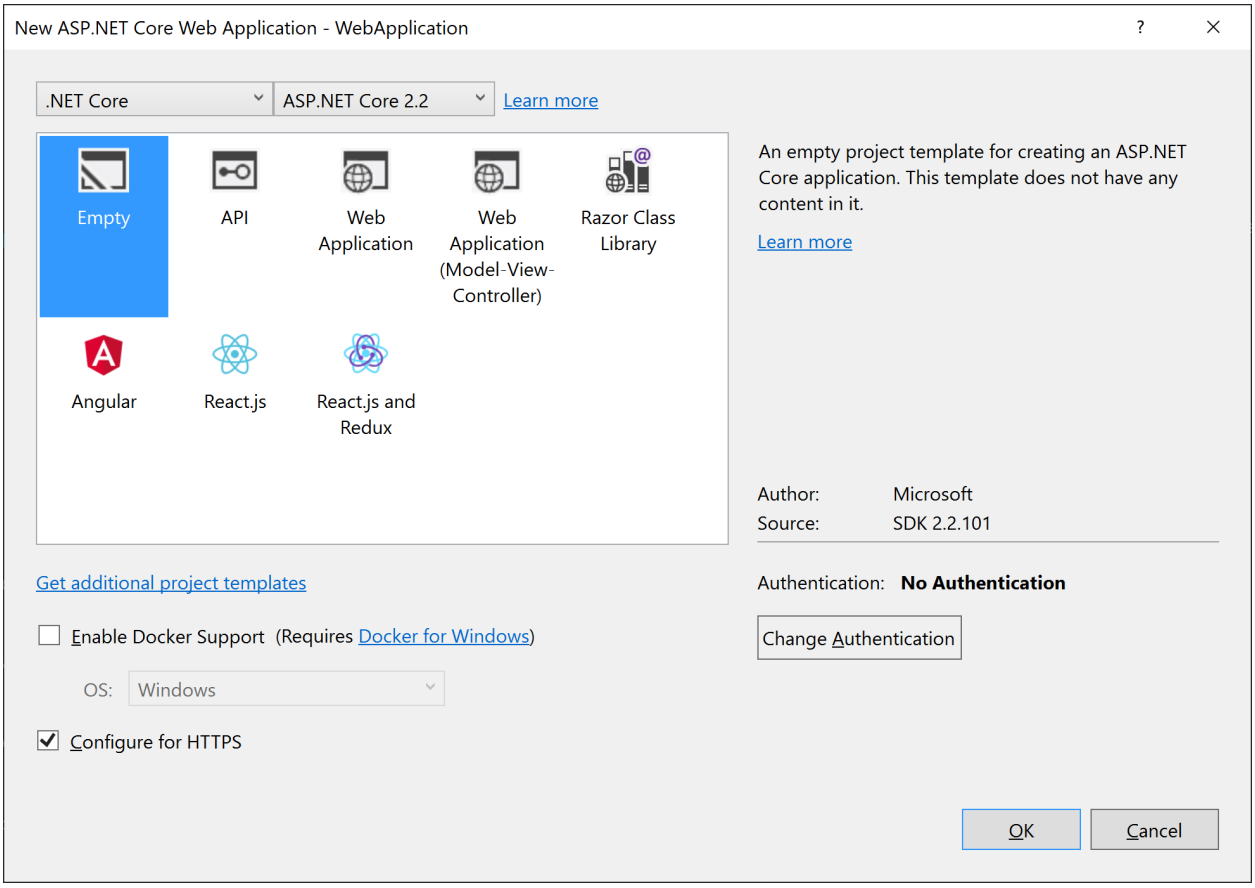
Tutorial: Create Simple ExtCore-Based Web Application

We are going to create simple modular and extendable ExtCore-based web application. First of all, if you are new to ASP.NET Core please visit this page. You will find there everything you need to start developing ASP.NET Core web applications.

Create Main Web Application

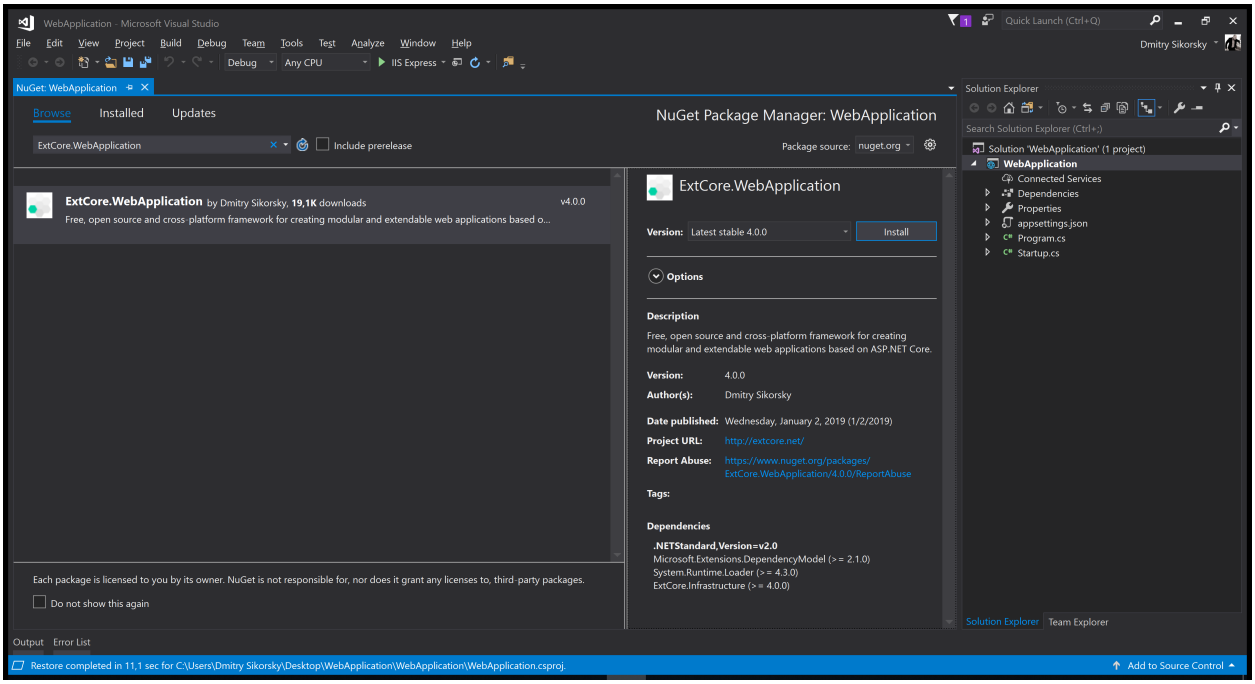
Now let's start Visual Studio and create new ASP.NET Core project:


New Project
Recent
Installed
Visual C#
Get Started
Windows Desktop
Web
.NET Core
.NET Standard
Cloud
Test
WCF
Visual Basic
Visual F#
SQL Server
Azure Data Lake
Azure Stream Analytics
Other Project Types
Online
Not finding what you are looking for?
Open Visual Studio Installer
Sort by: Default
Console App (.NET Core) Visual C#
Class Library (.NET Core) Visual C#
MSTest Test Project (.NET Core) Visual C#
NUnit Test Project (.NET Core) Visual C#
xUnit Test Project (.NET Core) Visual C#
ASP.NET Core Web Application Visual C#
Search (Ctrl+E)
Type: Visual C#
Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core or .NET Framework. Create Razor Pages, MVC, Web API, and Single Page (SPA) Applications.
Name: WebApplication
Location: C:\Users\Dmitry Sikorsky\Desktop\
Solution name: WebApplication
Browse...
Create directory for solution
Create new Git repository
OK
Cancel



Empty project is created.

Right click on your project in the Solution Explorer and open NuGet Package Manager. Switch to Browse tab and type ExtCore.WebApplication in the Search field (be sure that Include prerelease checkbox is checked). Click Install button:



You can get the same result manually by opening the WebApplication.csproj file and adding next line into it:

```
<ItemGroup>
  <PackageReference Include="ExtCore.WebApplication" Version="6.0.0" />
</ItemGroup>
```

Create the appsettings.json file in the project root. We will use this file to provide configuration parameters to ExtCore (such as path of the extensions folder). Now it should contain only one parameter `Extensions:Path` and look like this:

```
{
  "Extensions": {
    // Please keep in mind that you have to change '\' to '/' on Linux-based systems
    "Path": "\\Extensions"
  }
}
```

Open Startup.cs file. Inside the `ConfigureServices` method call `services.AddExtCore` one. Pass the extensions path as the parameter. Inside the `Configure` method call `applicationBuilder.UseExtCore` one with no parameters.

Now your `Startup` class should look like this:

```
public class Startup
{
  private string extensionsPath;

  public Startup(IHostingEnvironment hostingEnvironment, IConfiguration configuration)
  {
    this.extensionsPath = hostingEnvironment.ContentRootPath + configuration[
↪"Extensions:Path"];
  }

  public void ConfigureServices(IServiceCollection services)
```

(continues on next page)

(continued from previous page)

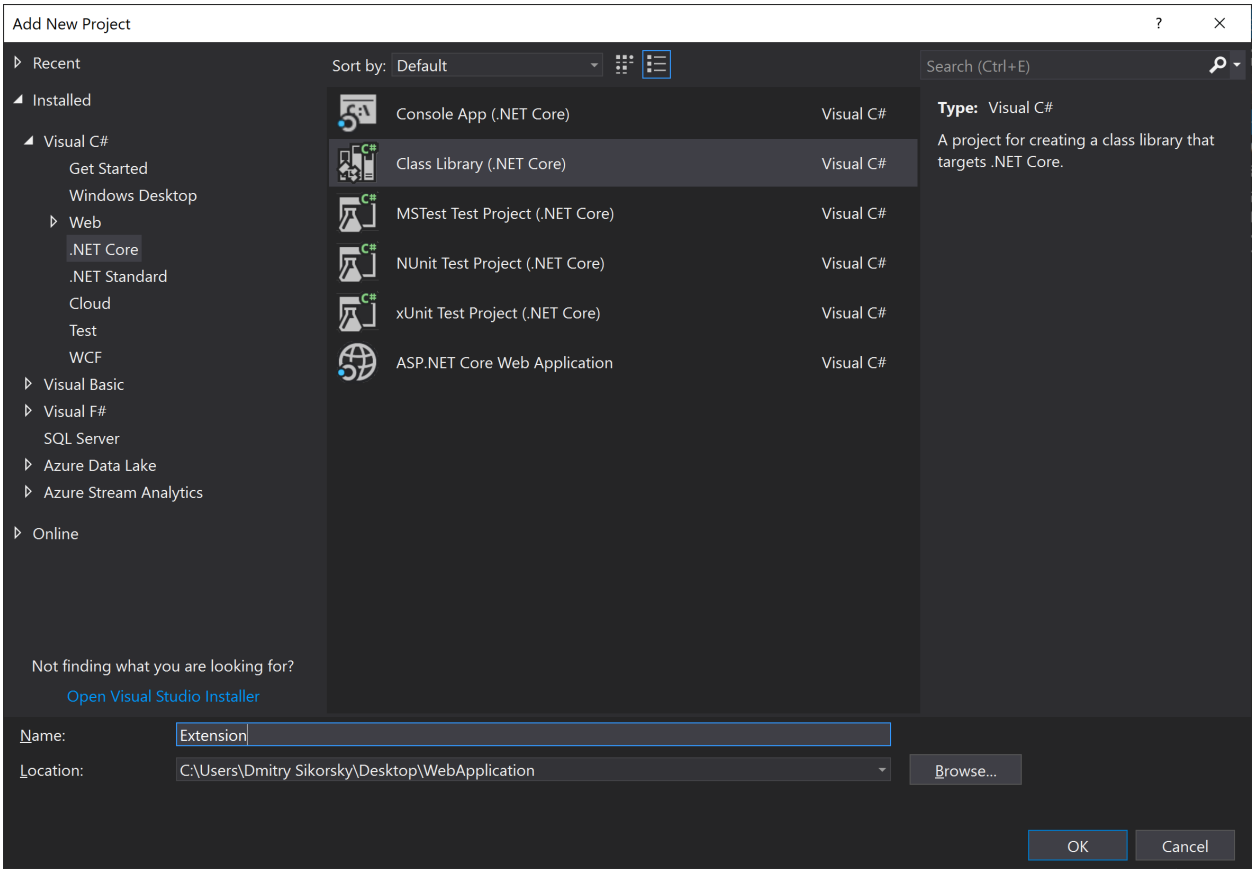
```
  {
    services.AddExtCore(this.extensionsPath);
  }

  public void Configure(IApplicationBuilder applicationBuilder)
  {
    applicationBuilder.UseExtCore();
    applicationBuilder.Run(async (context) =>
    {
      await context.Response.WriteAsync("Hello World!");
    });
  }
}
```

That's all, you now have ExtCore-based web application.

Now we need to create some extension project to show how ExtCore types discovering works.

Create Extension

Create new .NET Core class library project:



Open NuGet Package Manager and add dependency on the ExtCore.Infrastructure package.

Create `Extension` class and inherit it from `ExtCore.Infrastructure.ExtensionBase`. Override `Name` property in this way:

```
public override string Name
{
  get
  {
    return "Some name";
  }
}
```

It is enough for now.

Put it Together

We have two options to make our extension available in main web application:

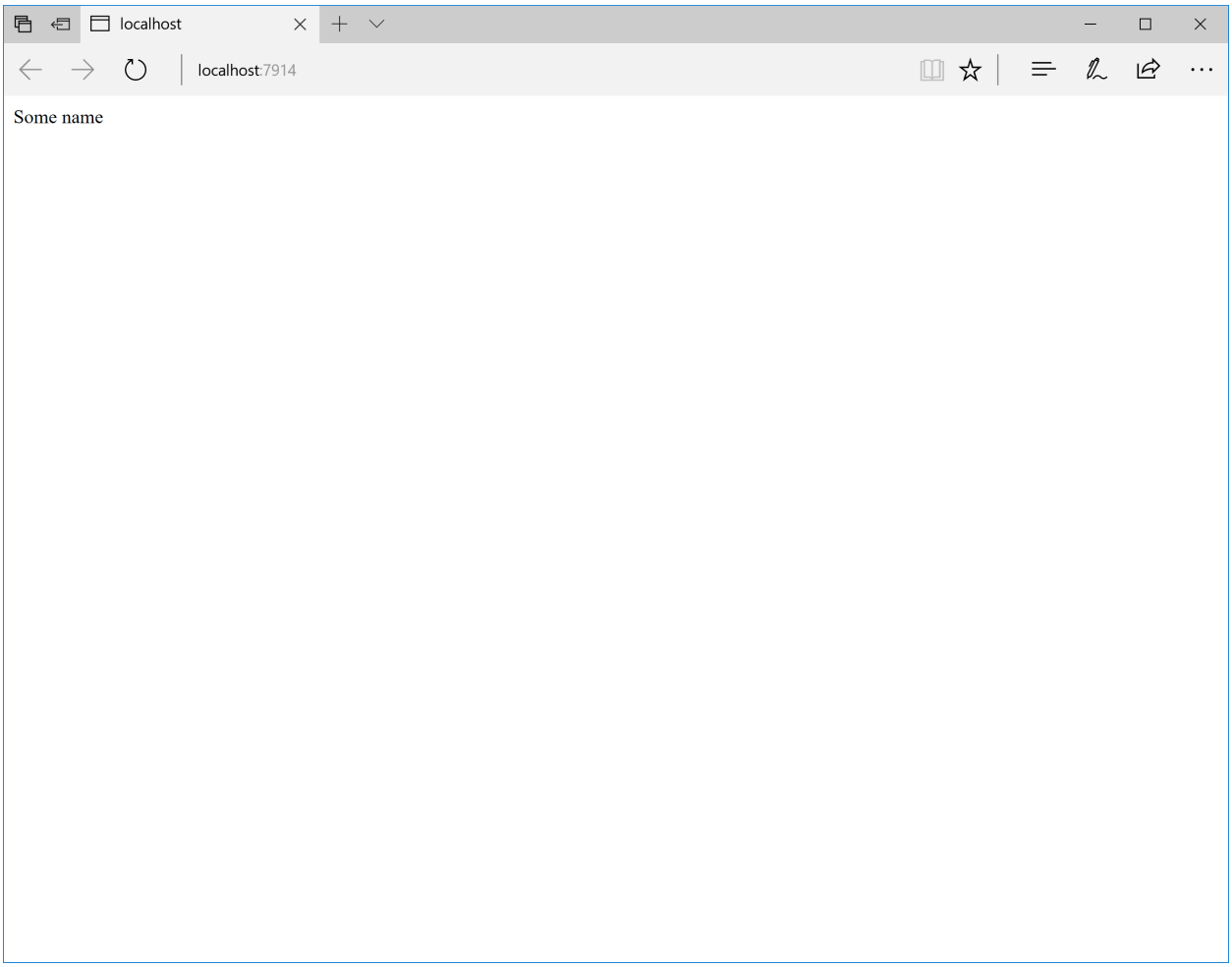
- add direct dependency on Extension in the WebApplication;
- put compiled Extension.dll file to extensions folder of the WebApplication that is configured in appsettings.json file.

While the first option is too obvious let's try the second one. Copy the Extension.dll file to the extensions folder of the WebApplication and modify `Configure` method of `Startup` class in next way:

```
public void Configure(IApplicationBuilder applicationBuilder)
{
  applicationBuilder.UseExtCore();
  applicationBuilder.Run(async (context) =>
  {
    await context.Response.WriteAsync(ExtensionManager.GetInstance<IExtension>().
↪Name);
  });
}
```

It will search for the implementation of the `IExtension` interface, create instance of found type, and write its `Name` property value on every request.

If we run our web application we will have the following result:



It may not look very impressive, but it's only the beginning! In the next tutorials we will see how extensions may execute their own code insite the `ConfigureServices` and `Configure` methods, how to use MVC and how to work with a storage.

You can find the complete source of this sample project on GitHub: ExtCore framework 6.0.0 sample simplest web application.

Tutorial: Create ExtCore-Based MVC Web Application

We are going to create modular and extendable ExtCore-based MVC web application. Please follow this tutorial to create simple ExtCore-based web application first. We will use it as a base.

So, we have the main web application and extension projects. They work but currently don't support MVC. We know that it is quite simple to add MVC support to ASP.NET Core web application using the `AddMvc` and `UseMvc` extension methods. With the ExtCore.Mvc extension it is even a bit easier.

Modify Main Web Application

Open NuGet Package Manager and add dependency on the ExtCore.Mvc package.

Now open Startup.cs file and remove the `applicationBuilder.Run` method calling from the `Configure` one, we don't need it anymore.

Now your `Startup` class should look like this:

```
public class Startup
{
  private string extensionsPath;

  public Startup(IHostingEnvironment hostingEnvironment, IConfiguration configuration)
  {
    this.extensionsPath = hostingEnvironment.ContentRootPath + configuration[
↪"Extensions:Path"];
  }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddExtCore(this.extensionsPath);
  }

  public void Configure(IApplicationBuilder applicationBuilder)
  {
    applicationBuilder.UseExtCore();
  }
}
```

Good. Now let's add some MVC-related things like controller and views. Also we will add CSS file too. It would be too obvious to add it to the main web application, so we will do that in the extension.

Modify Extension

First of all, replace dependency on ExtCore.Infrastructure with dependency on ExtCore.Mvc.Infrastructure (same version). The easiest way to do that is manually edit Extension.csproj file:

```
<ItemGroup>
  <PackageReference Include="ExtCore.Mvc.Infrastructure" Version="6.0.0" />
</ItemGroup>
```

Create Actions folder inside the project and create `UseEndpointAction` class inside it. Actions is ExtCore feature that allows extensions to execute some code inside the `ConfigureServices` and `Configure` methods of the web application. This class should look like this:

```
public class UseEndpointAciton : IUseEndpointsAction
{
  public int Priority => 1000;

  public void Execute(IEndpointRouteBuilder endpointRouteBuilder, IServiceProvider␣
↪serviceProvider)
  {
    endpointRouteBuilder.MapControllerRoute(name: "Default", pattern: "{controller}/
↪{action}", defaults: new { controller = "Default", action = "Index" });
  }
}
```

With this code our extension registers the default route for the web application which will use it. Each extension may register its own routes and make them have special order using the priorities.

Now we are ready to create controller and views.

Create `DefaultController` class and inherit it from `Controller`. Add simple `Index` action:

```
public class DefaultController : Controller
{
  public ActionResult Index()
  {
    return this.View();
  }
}
```

Create /Views/Shared/_Layout.cshtml and /Views/Default/Index.cshtml views.

_Layout.cshtml:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>@Html.Raw(this.ViewBag.Title as string)</title>
</head>
<body>
  @RenderBody()
</body>
</html>
```

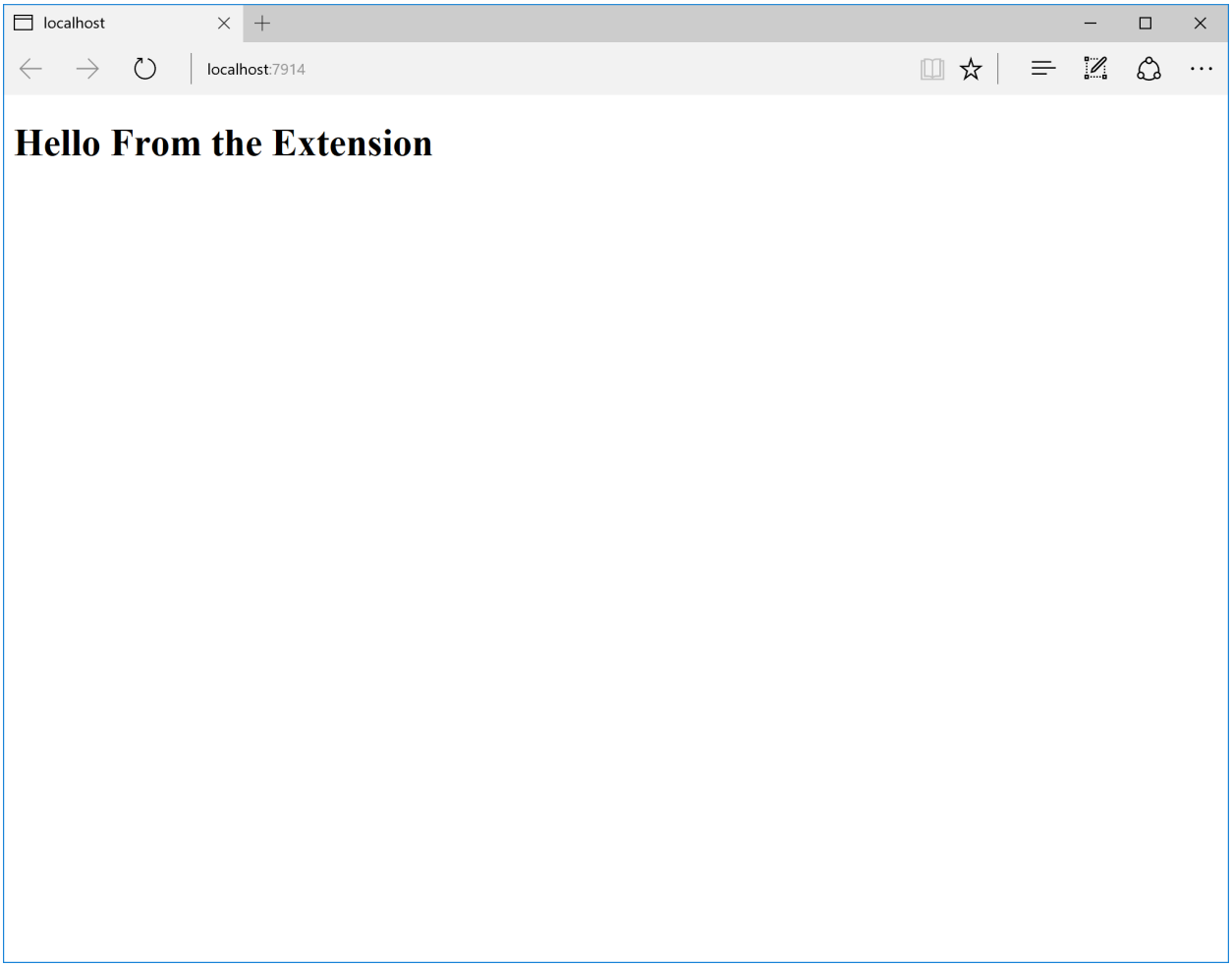
Index.cshtml:

```
<h1>Hello From the Extension</h1>
```

We need to tell the compiler to compile these views as resources to be able to use it later. Open the Extension.csproj file and add following lines there:

```
<ItemGroup>
  <EmbeddedResource Include="Views\**" />
</ItemGroup>
```

It is enough for now. Rebuild the solution and copy Extension.dll file to the extensions folder of the WebApplication. Run the web application:



We can see that controller and views are resolved. Cool! Now let's add some style to the our views. Create default.css file inside the /Styles folder (you need to create it too):

```
body {
  color: red;
}
```

Modify the Extension.csproj file again to tell the compiler to compile the styles too:

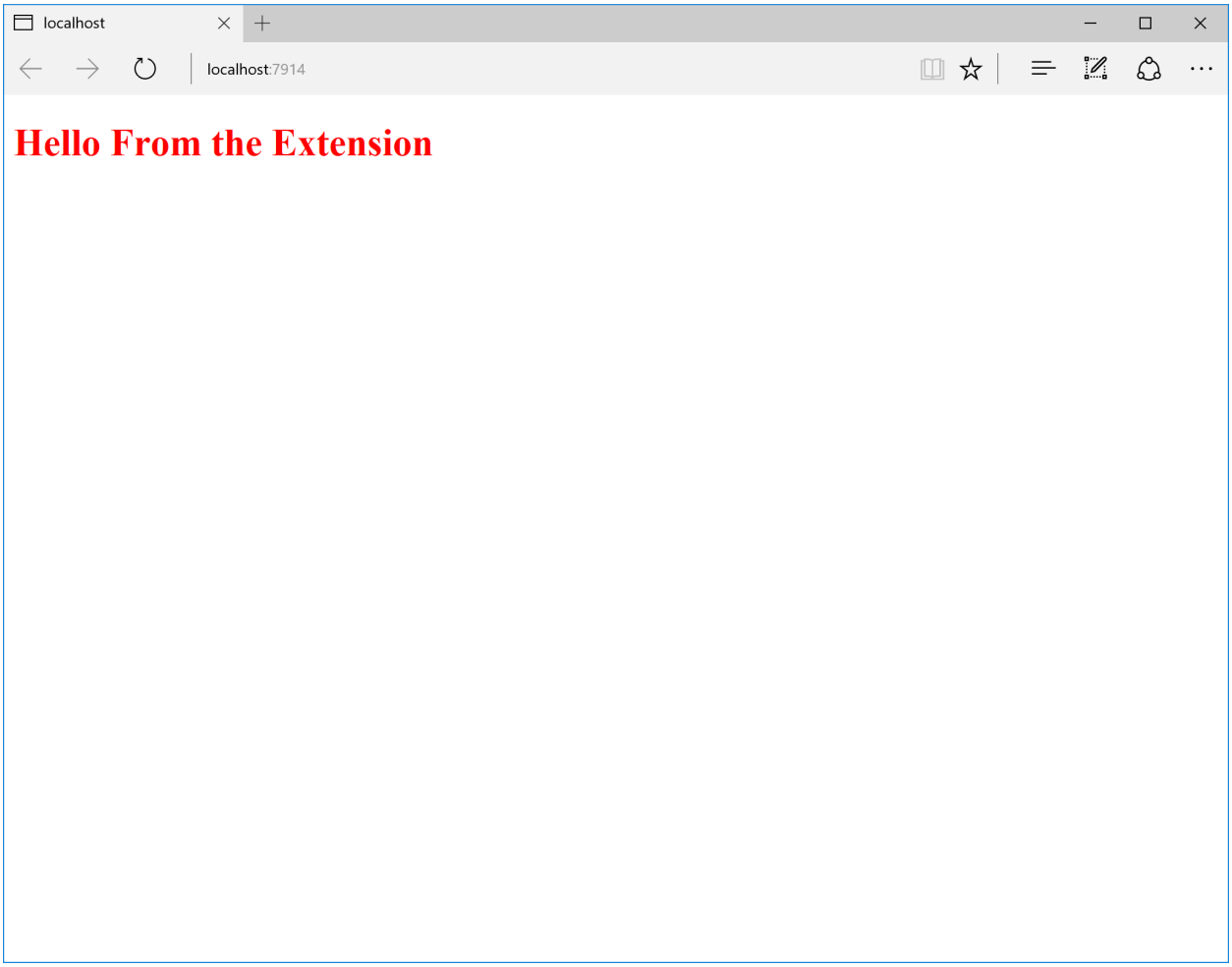
```
<ItemGroup>
  <EmbeddedResource Include="Styles\**;Views\**" />
</ItemGroup>
```

Finally, add the link to the CSS file to the Index.cshtml view:

```
<link href="Styles.default.css" rel="stylesheet" />
```

Note that resources have flat structure inside the assemblies so we need to replace / with . (dot) in the path to the CSS file.

Rebuild the solution again and replace ExtCoreExtension.dll file, run the web application:



As we can see, the text turns red. It means that everything works as expected. In the next tutorials we will see how to work with the storage.

You can find the complete source of this sample project on GitHub: ExtCore framework 6.0.0 sample MVC web application.

Tutorial: Create ExtCore-Based Web Application with Storage

We are going to create modular and extendable ExtCore-based web application with data. Please follow this tutorial to create ExtCore-based MVC web application first. We will use it as a base.

So, we have the main web application and extension projects. They work but currently don't know anything about data and storage. Let's assume that we want our extension to display some data (the list of persons for example) from the storage (SQLite database for example) in its view. The ExtCore.Data and ExtCore.Data.EntityFramework extensions will help us to achieve that goal.

In short, we should do the following:

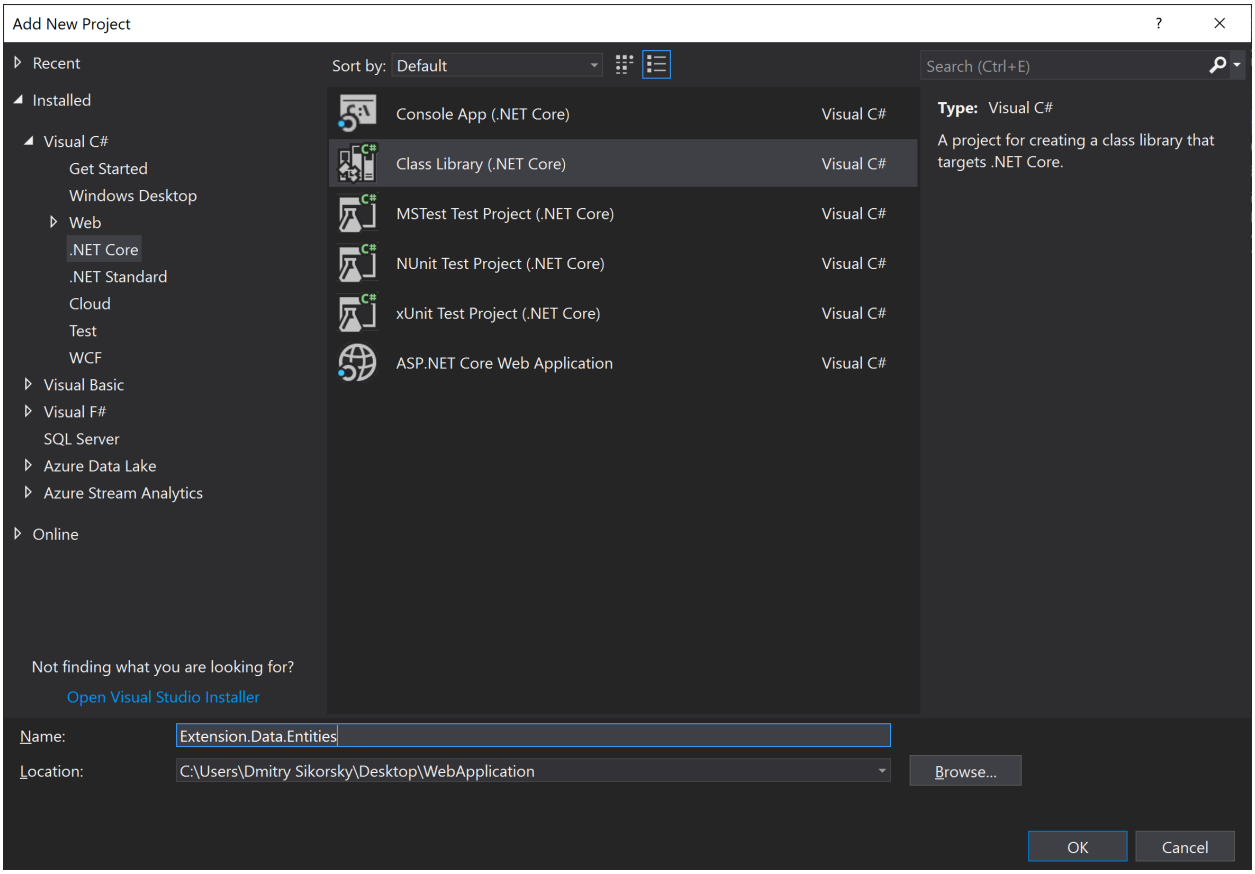
- create the storage (SQLite database in this case);
- describe the entities (the only one `Person` entity in this case);
- describe the repositories to work with that entities (abstraction and one implementation for each

of the supported storages; SQLite database support only in this case); * add logic for getting the persons from the database and displaying them.

Describe the Entities

To be able to share entities among different projects (and following the ExtCore.Data extension design pattern) we will describe entity classes in the separate project.

Create new .NET Core class library project:



Open NuGet Package Manager and add dependency on the ExtCore.Data.Entities.Abstractions version 6.0.0 package (be sure that Include prerelease checkbox is checked).

Create `Person` class that implements `ExtCore.Data.Entities.Abstractions.IEntity`. Create `Id` and `Name` properties. After that `Person` class should look like this:

```
public class Person : IEntity
{
  public int Id { get; set; }
  public string Name { get; set; }
}
```

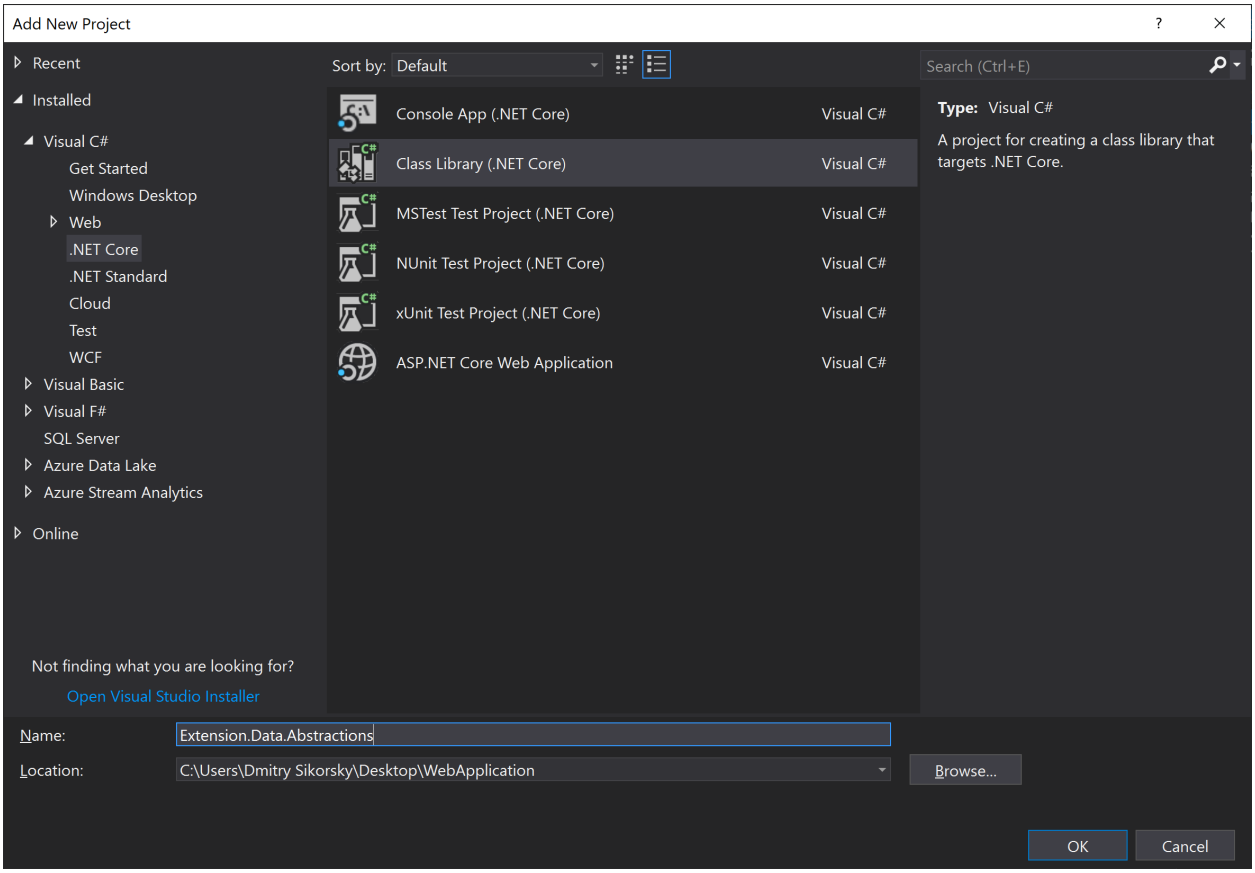
Describe the Repositories

ExtCore.Data extension implements Unit of Work and Repository design patterns. It means that all the work with the storage is performed in a single context, and it also means that every entity has its own repository that contains all methods you need to work with it. Repositories when created are resolved automatically, so everything we need to do is to create corresponding repositories.

To be able to share repositories among different projects (and following the ExtCore.Data extension design pattern) we will describe repository classes in the separate projects (one for abstractions and one for each of the supported storages).

Abstractions

Create new .NET Core class library project:



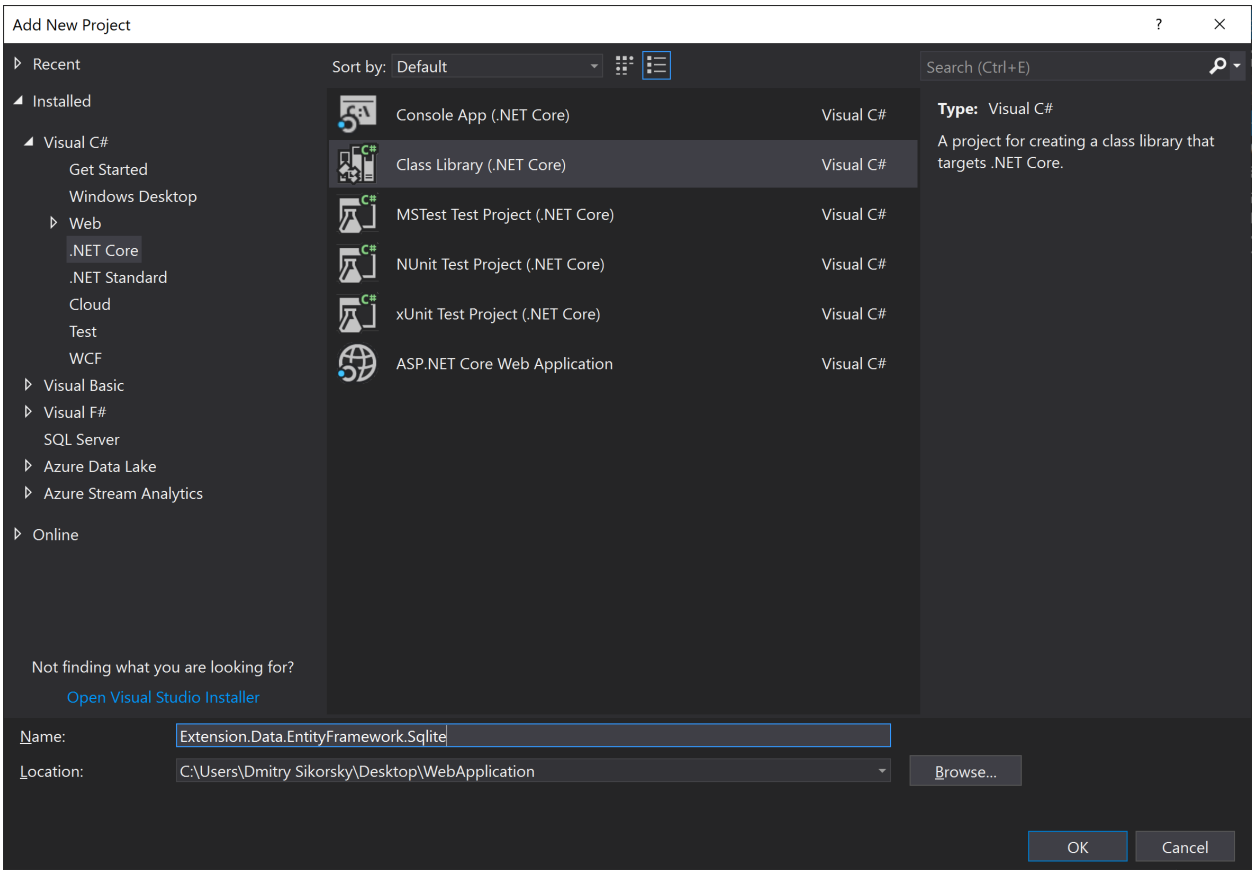
Open NuGet Package Manager and add dependency on the ExtCore.Data.Abstractions version 6.0.0 package. Also add dependency on your local Extension.Data.Entities project.

Create `IPersonRepository` interface that implements the `ExtCore.Data.Abstractions.IRepository` one. Create `All` method there. After that the `IPersonRepository` interface should look like this:

```
public interface IPersonRepository : IRepository
{
  IEnumerable<Person> All();
}
```

SQLite Storage Support

Create one more .NET Core class library project:



Open NuGet Package Manager and add dependency on the ExtCore.Data.EntityFramework.Sqlite version 6.0.0 package. Also add dependency on your local Extension.Data.Abstractions project.

Create `EntityRegistrar` class that implements the `ExtCore.Data.EntityFramework.IEntityRegistrar` interface. Override `RegisterEntities` method in this way:

```
public void RegisterEntities(ModelBuilder modelbuilder)
{
  modelbuilder.Entity<Person>(etb =>
    {
      etb.HasKey(e => e.Id);
      etb.Property(e => e.Id);
      etb.ToTable("Persons");
    }
  );
}
```

Now create `PersonRepository` class that implements `Extension.Data.Abstractions.IPersonRepository` interface and inherit it from the `ExtCore.Data.EntityFramework.RepositoryBase<Person>` class. Create `All` method there. After that `PersonRepository` class should look like this:

```
public class PersonRepository : RepositoryBase<Person>, IPersonRepository
{
  public IEnumerable<Person> All()
  {
    return this.dbSet.OrderBy(p => p.Name);
  }
```

(continues on next page)

(continued from previous page)

```
}
```

Modify Main Web Application

Now when we have everything we need to work with data and storage let's display the list of persons in the view.

First of all create the SQLite database with one Persons (pay attention to the case of the characters) table and few rows. You can use SqliteBrowser for that.

The second step is to add `ConnectionStrings:Default` parameter to the appsettings.json file:

```
"ConnectionStrings": {
  // Please keep in mind that you have to change '\' to '/' on Linux-based systems
  "Default": "Data Source=..\\..\\..\\db.sqlite"
}
```

Finally, open NuGet Package Manager and add dependencies on the ExtCore.Data version 6.0.0 and ExtCore.Data.EntityFramework.Sqlite version 6.0.0 packages.

Modify Extension

Add dependency on your local Extension.Data.Abstractions project.

Modify your `DefaultController` class to make it get parameter of type `IStorage` from the DI in the constructor and save that object to the private variable:

```
public DefaultController(IStorage storage)
{
  this.storage = storage;
}
```

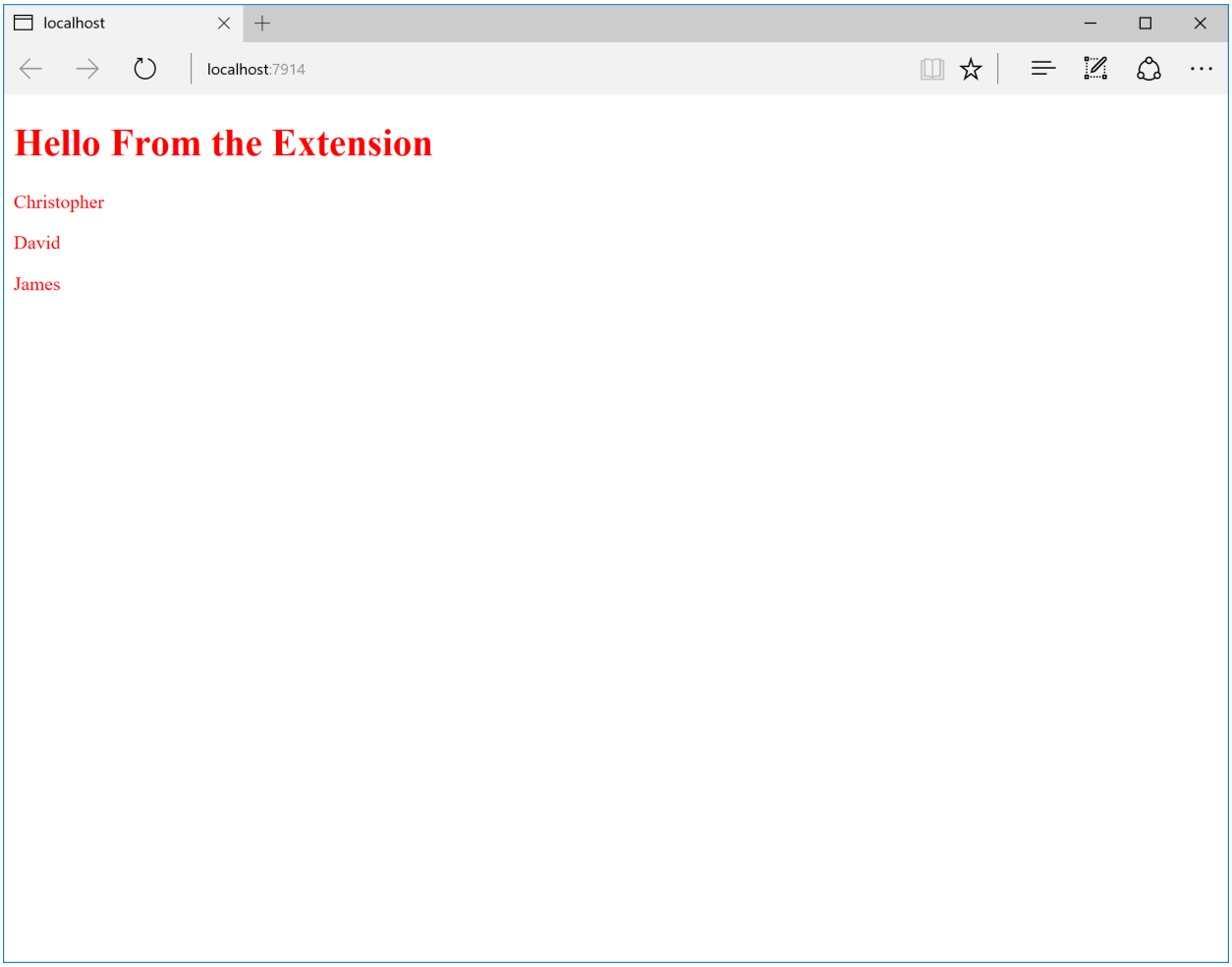
Now modify your `Index` action to get persons from the database and put them to the view:

```
public ActionResult Index()
{
  return this.View(this.storage.GetRepository<IPersonRepository>().All());
}
```

Now open your /Views/Default/Index.cshtml view and modify it in following way:

```
@model IEnumerable<Extension.Data.Entities.Person>
<h1>Hello From the Extension</h1>
@foreach (var person in this.Model)
{
  <p>@person.Name</p>
}
```

Rebuild the solution, put files Extension.dll, Extension.Data.Entities.dll, Extension.Data.Abstractions.dll, and Extension.Data.EntityFramework.Sqlite.dll to the extensions folder of the WebApplication, run the web application:



As we can see, data from the database is displayed.

You can find the complete source of this sample project on GitHub: ExtCore framework 6.0.0 sample web application that uses a database.

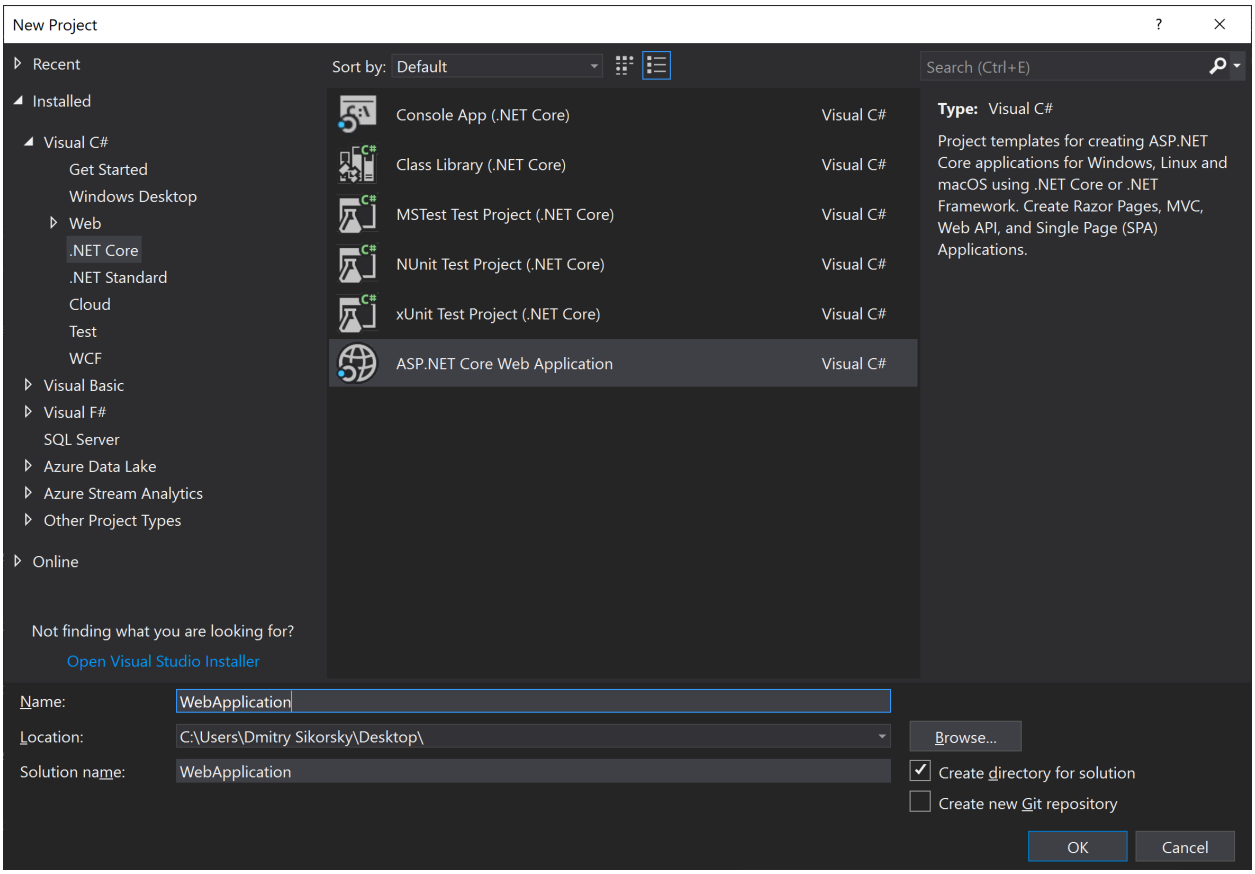
Tutorial: Registering and Using a Service Inside an Extension

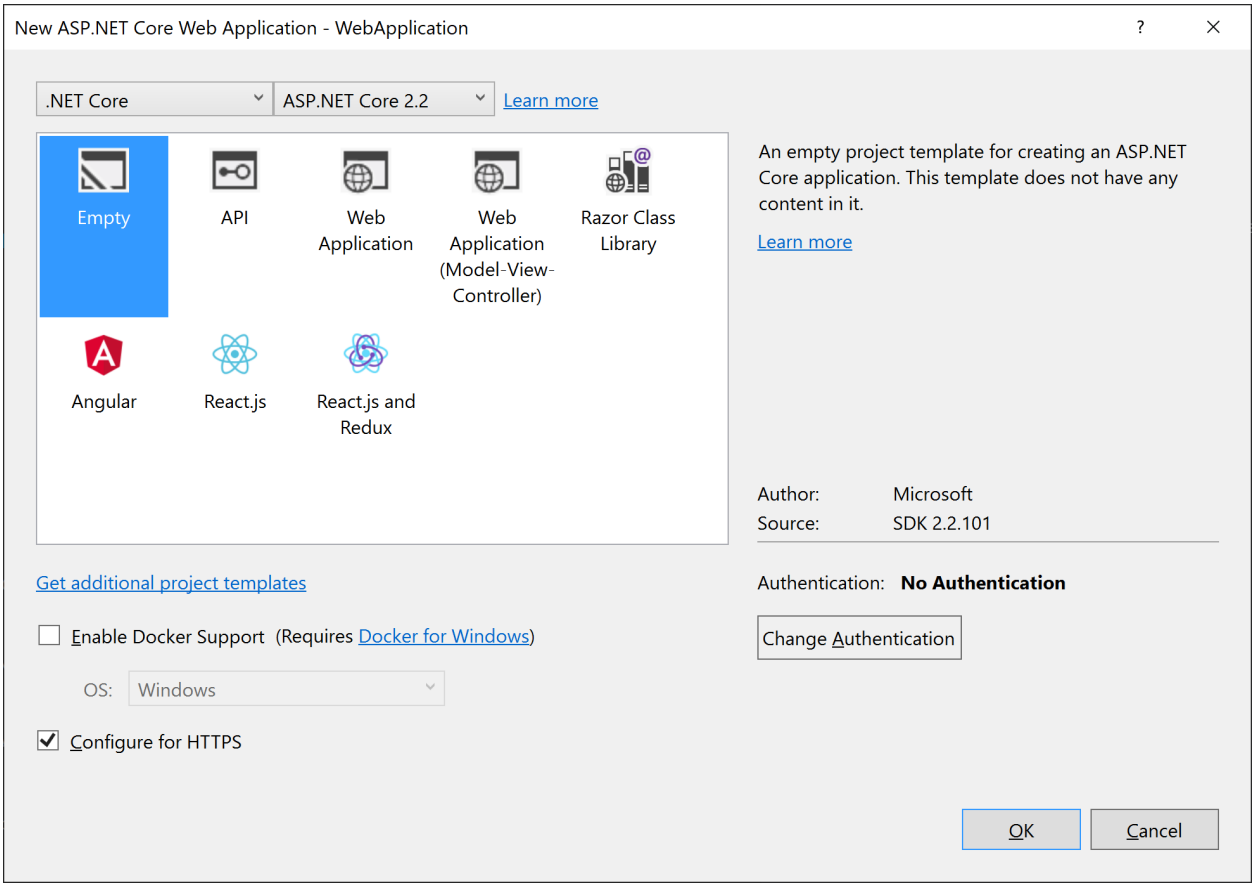
Often, we need to register a service inside an extension. For example, we may want to implement some interface in different ways and allow user to choose the implementation by using the specific extension. ExtCore.Data extension uses this approach to provide different storages support.

We will create the main web application project, one shared project that will contain the service interface, and 2 extension projects with the different implementations of that interface.

Create Main Web Application

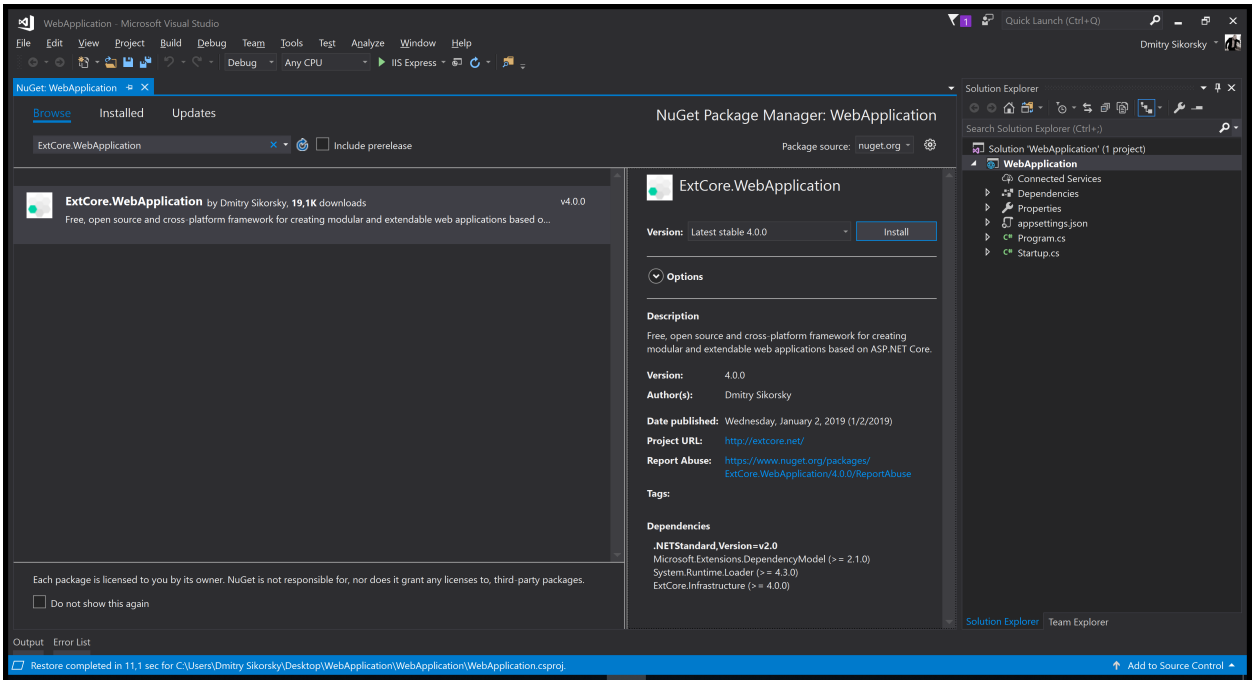
Now let's start Visual Studio and create new ASP.NET Core project:


New Project
Recent
Installed
Visual C#
Get Started
Windows Desktop
Web
.NET Core
.NET Standard
Cloud
Test
WCF
Visual Basic
Visual F#
SQL Server
Azure Data Lake
Azure Stream Analytics
Other Project Types
Online
Not finding what you are looking for?
Open Visual Studio Installer
Sort by: Default
Search (Ctrl+E)
Console App (.NET Core) Visual C#
Class Library (.NET Core) Visual C#
MSTest Test Project (.NET Core) Visual C#
NUnit Test Project (.NET Core) Visual C#
xUnit Test Project (.NET Core) Visual C#
ASP.NET Core Web Application Visual C#
Type: Visual C#
Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core or .NET Framework. Create Razor Pages, MVC, Web API, and Single Page (SPA) Applications.
Name: WebApplication
Location: C:\Users\Dmitry Sikorsky\Desktop\
Solution name: WebApplication
Browse...
Create directory for solution
Create new Git repository
OK
Cancel



Empty project is created.

Right click on your project in the Solution Explorer and open NuGet Package Manager. Switch to Browse tab and type ExtCore.WebApplication in the Search field (be sure that Include prerelease checkbox is checked). Click Install button:



Also add dependency on Microsoft.Extensions.Configuration.Json package.

Create the appsettings.json file in the project root. We will use this file to provide configuration parameters to ExtCore (such as path of the extensions folder). Now it should contain only one parameter `Extensions:Path` and look like this:

```
{
  "Extensions": {
    // Please keep in mind that you have to change '\' to '/' on Linux-based systems
    "Path": "\\Extensions"
  }
}
```

Open Startup.cs file. Inside the `ConfigureServices` method call `services.AddExtCore` one. Pass the extensions path as the parameter. Inside the `Configure` method call `applicationBuilder.UseExtCore` one with no parameters.

Now your `Startup` class should look like this:

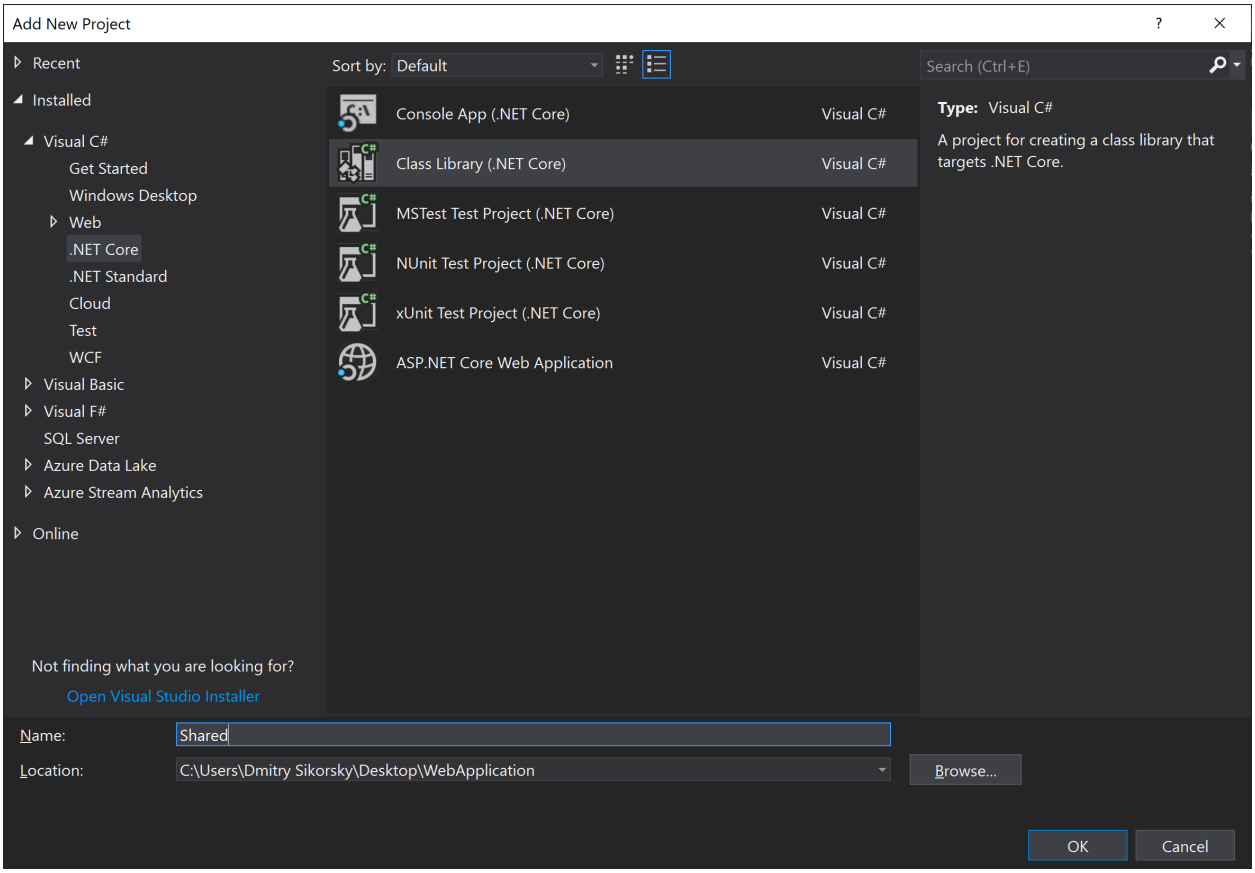
```
public class Startup
{
  private string extensionsPath;

  public Startup(IHostingEnvironment hostingEnvironment, IConfiguration configuration)
  {
    this.extensionsPath = hostingEnvironment.ContentRootPath + configuration[
↪"Extensions:Path"];
  }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddExtCore(this.extensionsPath);
  }

  public void Configure(IApplicationBuilder applicationBuilder)
```

(continues on next page)

(continued from previous page)

```
  {
    applicationBuilder.UseExtCore();
    applicationBuilder.Run(async (context) =>
    {
      await context.Response.WriteAsync("Hello World!");
    });
  }
}
```

That's all, you now have ExtCore-based web application. Now we need to create the shared project that will contain the service interface. Both the main web application and the extension projects will have explicit dependencies on this package:



Create the `IOperation` interface in this project:

```
public interface IOperation
{
  int Calculate(int a, int b);
}
```

Create Extensions

Now create 2 more projects for the service interface implementations: PlusExtension and MultiplyExtension. Add reference on the Shared project to both of them. After that, create corresponding classes in that projects:

```
public class PlusOperation : IOperation
{
  public int Calculate(int a, int b)
  {
    return a + b;
  }
}
```

```
public class MultiplyOperation : IOperation
{
  int Calculate(int a, int b)
  {
    return a * b;
  }
}
```

After that each extension needs to register its own implementation of the `IOperation` interface inside the ASP.NET Core DI. To do that we need to implement the `IConfigureServicesAction` interface (it is defined inside the ExtCore.Infrastructure package, don't forget to add the dependency). This is the example for the PlusExtension extension:

```
public class AddOperationAction : IConfigureServicesAction
{
  public int Priority => 1000;

  public void Execute(IServiceCollection services, IServiceProvider serviceProvider)
  {
    services.AddScoped(typeof(IOperation), typeof(PlusOperation));
  }
}
```

Good. We are ready for the final step.

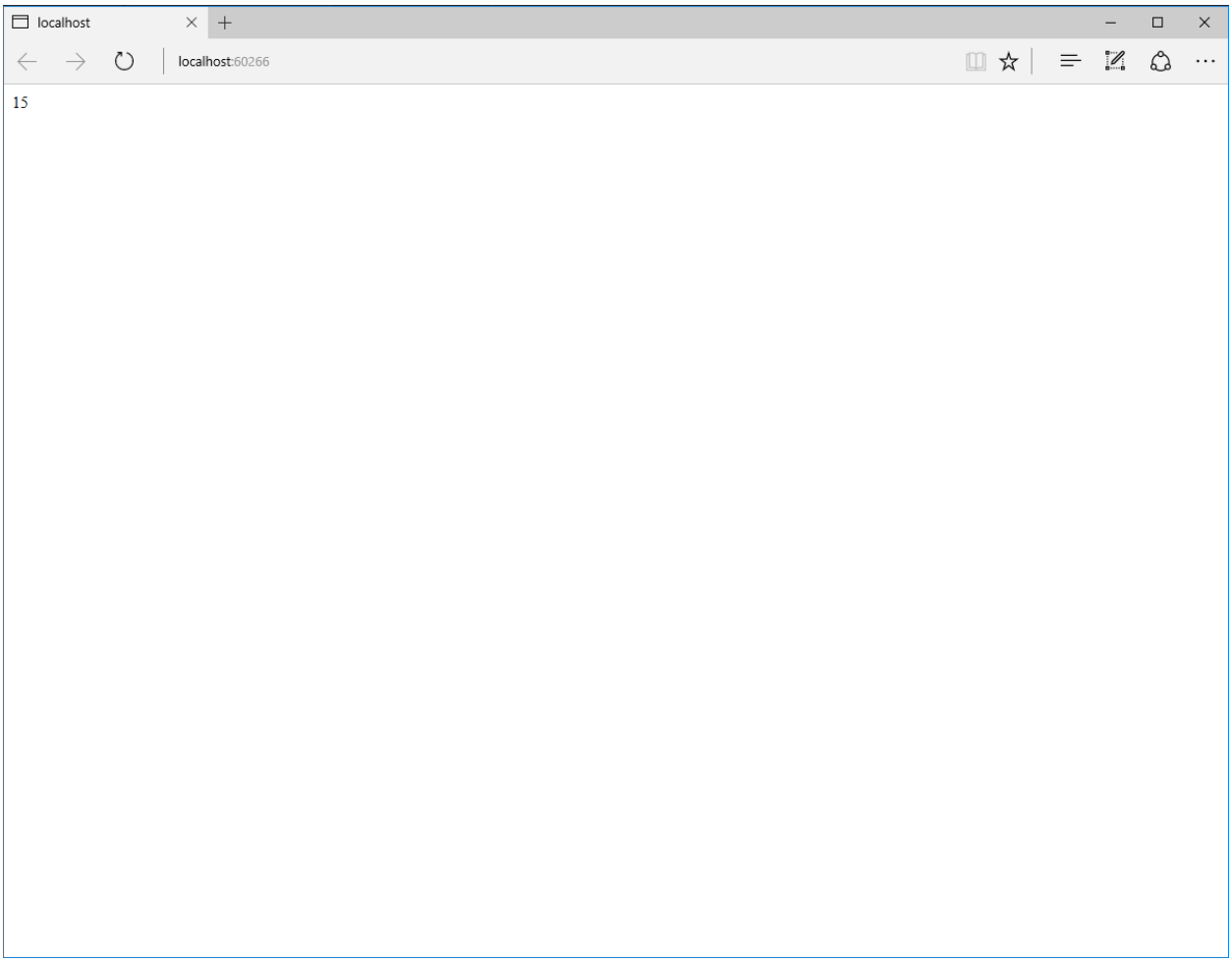
Put it Together

First of all, add reference on the Shared project to the main web application project. Now modify the `Configure` method in next way:

```
public void Configure(IApplicationBuilder applicationBuilder, IOperation operation)
{
  applicationBuilder.UseExtCore();
  applicationBuilder.Run(async (context) =>
    {

      await context.Response.WriteAsync(operation.Calculate(5, 10).ToString());
    }
  );
}
```

The implementation of the `IOperation` interface, which is used to calculate the final value, will be provided by the ASP.NET Core DI. Our code doesn't know which implementation is used, it is registered by the selected extension. To select the extension we need to copy its DLL file to the Extensions folder of the main web application, or add implicit reference on that project.

So, let's copy the PlusExtension.dll file to the Extensions folder and try to run our application:



Everything works as expected. We can replace the PlusExtension.dll with the MultiplyExtension.dll, restart the web application and the result will change.

You can find the complete source of this sample project on GitHub: ExtCore framework 6.0.0 sample web application that registers a service inside the extension.

Using Migrations

(It is only applicable for the ExtCore framework version 3.1.0-beta1 and higher.)

Migrations feature is only available if you are using Entity Framework Core as the ORM. To make it possible to use the Migrations tool, please, follow these steps:

1. Create the `DesignTimeStorageContextFactory` class (you can use any name) and inherit it from the ExtCore.Data.EntityFramework.DesignTimeStorageContextFactoryBase<T> one (here `T` is the type of the storage context you are using). You don't need to implement any methods in this class:

```
public class DesignTimeStorageContextFactory : DesignTimeStorageContextFactoryBase
↪<StorageContext>
{
}
```

2. Initialize the `StorageContextOptions.MigrationsAssembly` property with the name of the assembly (project) where the `DesignTimeStorageContextFactory` class is created:

```
services.Configure<StorageContextOptions>(options =>
  {
    options.ConnectionString = this.configurationRoot.GetConnectionString("Default");
    options.MigrationsAssembly = typeof(DesignTimeStorageContextFactory).
↪GetTypeInfo().Assembly.FullName;
  }
);
```

3. Call the `DesignTimeStorageContextFactory.Initialize()` static method after the `StorageContextOptions` class configuration:

```
DesignTimeStorageContextFactory.Initialize(services.BuildServiceProvider());
```

That's all, now you can use the Migrations tool.

Using Identity

(It is only applicable for the ExtCore framework version 3.1.0-beta2 and higher.)

Please, take a look at this sample.

1.3 Fundamentals

1.3.1 Working Principle

ExtCore is included into a project using two extension methods: `AddExtCore` and `UseExtCore`.

First of all, inside the `AddExtCore` method ExtCore discovers and loads the assemblies (using the implementation of the IAssemblyProvider interface). When the assemblies are discovered and loaded it caches it inside the ExtensionManager static class. This class is the entry point for the ExtCore type discovery mechanism, all extensions can use it to get the information about each other.

Then, using the `AddExtCore` and `UseExtCore` methods, ExtCore executes user actions (code fragments) from all the extensions inside the `ConfigureServices` and `Configure` methods. These actions are defined by the implementations of the IConfigureServicesAction and IConfigureAction interfaces and allows developer to specify the execution order by using the `Priority` property. This is one of the key features, because it allows the extensions to execute their own code during the web application initialization and startup. They can register services, add middleware etc.

ExtCore.Mvc extension also defines two more action interfaces which might be used to configure MVC: IAddMvcAction and IUseMvcAction ones.

ExtCore.Mvc extension discovers all the views (added as resources and/or precompiled) and static content (added as resources) and make it accessible using the custom implementation of the `IFileProvider` interface.

1.3.2 Initialization and Startup

ExtCore initialization and startup process consists of running two extension methods: `AddExtCore` and `UseExtCore`. These methods must be called inside the `ConfigureServices` and `Configure` methods of the web application's `Startup` class:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddExtCore("absolute path to your extensions");
}

public void Configure(IApplicationBuilder applicationBuilder, IHostingEnvironment␣
↪hostingEnvironment)
{
  if (hostingEnvironment.IsDevelopment())
  {
    applicationBuilder.UseDeveloperExceptionPage();
    applicationBuilder.UseDatabaseErrorPage();
  }

  applicationBuilder.UseExtCore();
}
```

`AddExtCore` Method

This method discovers and loads the assemblies and caches them into the ExtensionManager class. Then it executes code from all the extensions inside the `ConfigureServices` method using the implementations of the IConfigureServicesAction interface.

`UseExtCore` Method

This method executes code from all the extensions inside the `Configure` method using the implementations of the IConfigureAction interface.

1.3.3 Debugging ExtCore Extensions

To be able to debug an ExtCore extension you need to add the explicit project references to all of its projects to the main web application. When development process is complete, you may remove that references and use the extension as the DLL files.

1.3.4 Getting and Reading Logs

ExtCore writes logs while initialization and startup process so you can understand what is going on and where the problem is when something goes wrong. Also using logs you can see which assemblies have been discovered and loaded, which actions have been executed by extensions in the `ConfigureServices` and `Configure` methods etc.

Getting Logs

By default ASP.NET Core web application (as well as ExtCore-based one) doesn't have any logger configured, so logs are not shown. To configure the logger, you can follow this article or just add next line of code in the constructor of your Startup class:

```
public Startup(ILoggerFactory loggerFactory)
{
  loggerFactory.AddConsole();
}
```

This approach is used in our sample so you can take a look at it there. It is important to do that in the constructor and not in the `Configure` method, because otherwise you will miss the logs before the `Configure` method is called.

It is an excerpt from our sample application log:

```
info: ExtCore.WebApplication[0]
      Discovering and loading assemblies from path 'C:\Path\WebApplication\Extensions'
info: ExtCore.WebApplication[0]
      Assembly 'WebApplication.ExtensionA, Version=1.0.0.0, Culture=neutral,␣
↪PublicKeyToken=null' is discovered and loaded
info: ExtCore.WebApplication[0]
      Assembly 'WebApplication.ExtensionB.Data.Abstractions, Version=1.0.0.0,␣
↪Culture=neutral, PublicKeyToken=null' is discovered and loaded
info: ExtCore.WebApplication[0]
      Assembly 'WebApplication.ExtensionB.Data.Entities, Version=1.0.0.0,␣
↪Culture=neutral, PublicKeyToken=null' is discovered and loaded
info: ExtCore.WebApplication[0]
      Assembly 'WebApplication.ExtensionB.Data.EntityFramework.Sqlite, Version=1.0.0.
↪0, Culture=neutral, PublicKeyToken=null' is discovered and loaded
info: ExtCore.WebApplication[0]
      Assembly 'WebApplication.ExtensionB, Version=1.0.0.0, Culture=neutral,␣
↪PublicKeyToken=null' is discovered and loaded
info: ExtCore.WebApplication[0]
      Discovering and loading assemblies from DependencyContext
info: ExtCore.WebApplication[0]
      Assembly 'WebApplication, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
↪' is discovered and loaded
info: ExtCore.WebApplication[0]
      Assembly 'ExtCore.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=null'␣
↪is discovered and loaded
info: ExtCore.WebApplication[0]
      Assembly 'ExtCore.Data.Abstractions, Version=2.0.0.0, Culture=neutral,␣
↪PublicKeyToken=null' is discovered and loaded
info: ExtCore.WebApplication[0]
      Assembly 'ExtCore.Data.Entities.Abstractions, Version=2.0.0.0, Culture=neutral,␣
↪PublicKeyToken=null' is discovered and loaded
info: ExtCore.WebApplication[0]
      Assembly 'ExtCore.Data.EntityFramework, Version=2.0.0.0, Culture=neutral,␣
↪PublicKeyToken=null' is discovered and loaded
info: ExtCore.WebApplication[0]
      Assembly 'ExtCore.Data.EntityFramework.Sqlite, Version=2.0.0.0, Culture=neutral,
↪ PublicKeyToken=null' is discovered and loaded
info: ExtCore.WebApplication[0]
      Assembly 'ExtCore.Infrastructure, Version=2.0.0.0, Culture=neutral,␣
↪PublicKeyToken=null' is discovered and loaded
info: ExtCore.WebApplication[0]
      Assembly 'ExtCore.Mvc, Version=2.0.0.0, Culture=neutral, PublicKeyToken=null'␣
↪is discovered and loaded
info: ExtCore.WebApplication[0]
      Assembly 'ExtCore.Mvc.Infrastructure, Version=2.0.0.0, Culture=neutral,␣
↪PublicKeyToken=null' is discovered and loaded
info: ExtCore.WebApplication[0]
      Assembly 'ExtCore.WebApplication, Version=2.0.0.0, Culture=neutral,␣
↪PublicKeyToken=null' is discovered and loaded
info: ExtCore.WebApplication[0]
      Assembly 'Newtonsoft.Json, Version=9.0.0.0, Culture=neutral,␣
↪PublicKeyToken=30ad4fe6b2a6aeed' is discovered and loaded
info: ExtCore.WebApplication[0]
```

(continues on next page)

(continued from previous page)

```
      Assembly 'Remotion.Linq, Version=2.1.0.0, Culture=neutral,␣
↪PublicKeyToken=fee00910d6e5f53b' is discovered and loaded
info: ExtCore.WebApplication[0]
      Executing ConfigureServices action 'ExtCore.Data.Actions.AddStaticFilesAction'
info: ExtCore.WebApplication[0]
      Executing ConfigureServices action 'ExtCore.Data.EntityFramework.Actions.
↪AddStaticFilesAction'
info: ExtCore.WebApplication[0]
      Executing ConfigureServices action 'ExtCore.Mvc.Actions.AddStaticFilesAction'
info: ExtCore.WebApplication[0]
      Executing ConfigureServices action 'ExtCore.Mvc.Actions.AddMvcAction'
info: ExtCore.WebApplication[0]
      Executing Configure action 'ExtCore.Mvc.Actions.UseStaticFilesAction'
info: ExtCore.WebApplication[0]
      Executing Configure action 'ExtCore.Mvc.Actions.UseMvcAction'
info: ExtCore.Mvc[0]
      Executing UseMvc action 'ExtensionA.Actions.UseMvcAction'
info: ExtCore.Mvc[0]
      Executing UseMvc action 'ExtensionB.Actions.UseMvcAction'
```

Reading Logs

Let's take a look at the log output.

The first 2 lines indicate that the process of the assemblies loading from the specific path has begun. The path is displayed too, so you can check it:

```
info: ExtCore.WebApplication[0]
      Discovering and loading assemblies from path 'C:\Path\WebApplication\Extensions'
```

Then we can see few lines that show the assemblies that are discovered and loaded.

The next 2 lines indicate that the process of the assemblies loading from the `DependencyContext` has begun:

```
info: ExtCore.WebApplication[0]
      Discovering and loading assemblies from DependencyContext
```

Discovered and loaded assemblies are displayed again.

After the assemblies are discovered and resolved, user actions inside the `ConfigureServices` and `Configure` methods are executed:

```
info: ExtCore.WebApplication[0]
      Executing ConfigureServices action 'ExtCore.Data.Actions.AddStaticFilesAction'
info: ExtCore.WebApplication[0]
      Executing ConfigureServices action 'ExtCore.Data.EntityFramework.Actions.
↪AddStaticFilesAction'
info: ExtCore.WebApplication[0]
      Executing ConfigureServices action 'ExtCore.Mvc.Actions.AddStaticFilesAction'
info: ExtCore.WebApplication[0]
      Executing ConfigureServices action 'ExtCore.Mvc.Actions.AddMvcAction'
info: ExtCore.WebApplication[0]
      Executing Configure action 'ExtCore.Mvc.Actions.UseStaticFilesAction'
info: ExtCore.WebApplication[0]
      Executing Configure action 'ExtCore.Mvc.Actions.UseMvcAction'
info: ExtCore.Mvc[0]
```

(continues on next page)

(continued from previous page)

```
      Executing UseMvc action 'ExtensionA.Actions.UseMvcAction'
info: ExtCore.Mvc[0]
      Executing UseMvc action 'ExtensionB.Actions.UseMvcAction'
```

It is easy to understand what is going on and what is executed and to check the execution order.

Initialization and startup process is now finished.

1.3.5 Default Assembly Provider

ExtCore uses the default implementation of the IAssemblyProvider interface to discover and load the assemblies.

The instance of that class is created and set in the constructor of the Startup class. All the ExtCore-based web applications must inherit their `Startup` classes from this one.

There is another one overloaded constructor which gets the second parameter of the `IAssemblyProvider` type. You can use it to specify your custom assembly provider. Also, you can use the existing one but change the IsCandidateAssembly or IsCandidateCompilationLibrary filter.

1.3.6 Unified Extension Structure

Here X is your application name and Y is your extension name:

- X.Y;
- X.Y.Data.Entities;
- X.Y.Data.Abstractions;
- X.Y.Data.SpecificStorageA;
- X.Y.Data.SpecificStorageB;
- X.Y.Data.SpecificStorageC;
- X.Y.Frontend;
- X.Y.Backend;
- etc.

For example, we can take a look at ExtCore.Data extension structure:

- ExtCore.Data;
- ExtCore.Data.Entities.Abstractions;
- ExtCore.Data.Abstractions;
- ExtCore.Data.EntityFramework;
- ExtCore.Data.EntityFramework.PostgreSql;
- ExtCore.Data.EntityFramework.Sqlite;
- ExtCore.Data.EntityFramework.SqlServer.

The idea is that your extension is logically split into the projects. Entities are just entities (Data.Entities project), they don't need to have any links to the different extension's projects (but entities project may have dependency on the entities project of the different extension to be able to work with them; for example, your Statistics extension Report entity may need to be able to work with a Product entity from the Ecommerce extension).

Repository abstractions are put inside the Data.Abstractions. This project needs to know about the Data.Entities one, and that's all. All the code that needs to work with the entities should do that using the repository abstractions, without any explicit links on the concrete implementations.

If your extension has some services, split them into the abstractions (Services.Abstractions) and the default implementations (Services.Default).

Main extension project shouldn't contain any shared content, like utility classes etc. None of the other extension projects should have dependencies on it. The purpose of the main extension's project is to keep extension metadata and to resolve and register services. It should look for the existing implementations of the given services using the ExtensionManager class and register them in this way, without having an explicit dependency on the implementations project (please look how it is done in the ExtCore.Data extension). In this case it will be possible to replace the services implementations just by copying another DLL file into the extensions folder or adding some NuGet package.

1.3.7 Usage for authorization middleware

If you must use *Microsoft.AspNetCore.Authorization* in your application, the call must be made in an extension.

Extcore uses internal priorities that start at 10000.

You must therefore give a priority higher than this number when calling Authorize.

```
using System;
using ExtCore.Infrastructure.Actions;
using Microsoft.AspNetCore.Builder;

namespace Barebone.Actions
{
    public class UseAuthorizationAction : IConfigureAction
    {
        public void Execute(IApplicationBuilder applicationBuilder, IServiceProvider␣
↪serviceProvider)
        {
            applicationBuilder.UseAuthorization();
        }

        public int Priority => 10001;
    }
}
```

Otherwise, you will get an meaningful error stating that ASP.NET Core cannot find the middleware.

1.4 Extensions

1.4.1 ExtCore.FileStorage

This extension allows developer to work with a file storage through the abstraction layer and easily replace, let's say, file system storage with the Dropbox or Azure Blob Storage ones without changing any code.

Packages

- ExtCore.FileStorage;
- ExtCore.FileStorage.Abstractions;

- ExtCore.FileStorage.Dropbox;
- ExtCore.FileStorage.FileSystem.

1.4.2 ExtCore.Data

By default, ExtCore doesn't know anything about data, but you can use ExtCore.Data extension to have unified approach to working with data and the single data storage context among all the extensions. Data storage might be represented by a database, a web API, a file structure or anything else.

Currently ExtCore.Data supports MySQL, PostgreSql, SQLite, and SQL Server with Dapper or Entity Framework Core as ORM. You can add your own database or ORM support.

Packages

- ExtCore.Data;
- ExtCore.Data.Abstractions;
- ExtCore.Data.Entities.Abstractions;
- ExtCore.Data.Dapper;
- ExtCore.Data.Dapper.MySql;
- ExtCore.Data.Dapper.PostgreSql;
- ExtCore.Data.Dapper.Sqlite;
- ExtCore.Data.Dapper.SqlServer;
- ExtCore.Data.EntityFramework;
- ExtCore.Data.EntityFramework.MySql;
- ExtCore.Data.EntityFramework.PostgreSql;
- ExtCore.Data.EntityFramework.Sqlite;
- ExtCore.Data.EntityFramework.SqlServer.

1.4.3 ExtCore.Mvc

By default, ExtCore web applications are not MVC ones. MVC support is provided for them by ExtCore.Mvc extension. This extension initializes MVC, makes it possible to use controllers, view components, views (added as resources and/or precompiled), static content (added as resources) from other extensions etc.

Also, it allows extension to register custom routes in a specific order.

Packages

- ExtCore.Mvc;
- ExtCore.Mvc.Infrastructure.

1.4.4 ExtCore.Events

It can be used by the extension to notify the code in this or any other extension about some events.

Packages

- ExtCore.Events.

1.5 Migration

1.5.1 4.x.x to 5.x.x

There are 3 main differences between ExtCore 4.x.x and 5.x.x.

.NET Core 3.0 applications can't run on top of the full .NET Framework.

Razor runtime compilation is now (starting from .NET Core 3.0) turned off by default. So far (6.0.0), ExtCore does not support Razor runtime compilation at all, so views and pages added as resources won't be resolved anymore.

To make them work you have to convert all the projects containing Razor files to Razor class libraries.

4.x.x:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <EmbeddedResource Include="Views\**;Images\**" />
  </ItemGroup>

  ...

</Project>
```

5.x.x:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <AddRazorSupportForMvc>true</AddRazorSupportForMvc>
  </PropertyGroup>

  <ItemGroup>
    <EmbeddedResource Include="Images\**" />
  </ItemGroup>

  ...

</Project>
```

Routes are replaced with endpoints, so `ExtCore.Mvc.Infrastructure.Actions.IUseMvcAction` is replaced with `ExtCore.Mvc.Infrastructure.Actions.IUseEndpointsAction`.

1.5.2 5.x.x to 6.x.x

Migration only needs target framework to be changed from netcoreapp3.1 to net5.0.