Lesson 3 - How to create “CUSTOMER REVIEW” module for PLATFORM MANAGER

# Summary

Use this guide to create a “Customer Review” module for Virto Commerce Platform Manager

# Video

<https://web.microsoftstream.com/video/43fd5a0a-d482-4de9-93af-4e0ad0837601>

# Prerequisites

* Installed Virto Commerce Platform Manager
* Basic C# knowledge
* Visual Studio 2017 or higher

# Glossary

**VC** – Virto Commerce

**Platform Manager** – Virto Commerce Platform Manager

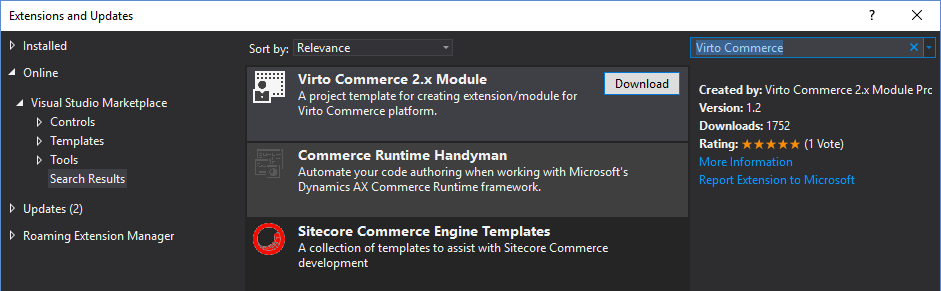
**JS** – Java Script

**VS** – Visual Studio

# Create a new module 03:32 - 12:20

New module should be created from a special VC module template in Visual Studio. The template is available as a Visual Studio extension online.

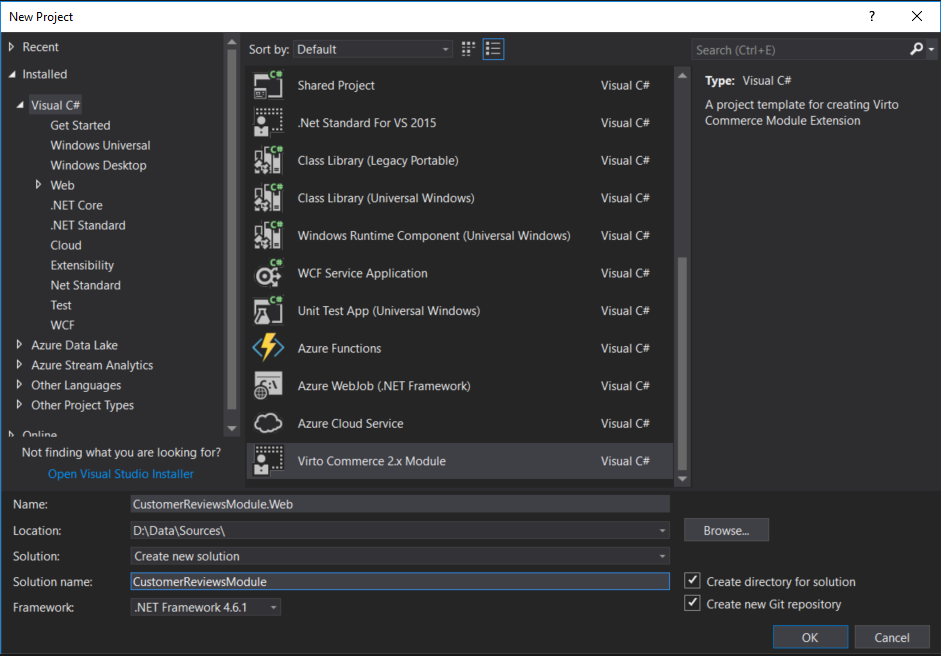
## Virto Commerce template

Open Visual Studio, go to **Tools > Extensions and Updates**. Search for **Virto Commerce 2.x Module**. 

Install it and restart Visual Studio.

Now, in Visual Studio click **New Project**, search for an existing **Virto Commerce 2.x Module** project. Name it, according to the naming convention. For example:

* “Name”: **CustomerReviewsModule.Web**
* “Solution name”: **CustomerReviewsModule**



After new module created, we should fill in title, description and authors attributes in module.manifest file:

|  |
| --- |
| <module>  ....  <title>Sample Customer reviews module</title>  <description>Sample module demonstrating best practices in a real life example.</description>  <authors>  <author>Egidijus Mazeika</author>  </authors>  ....  <module> |

*module.manifest* contains various attributes describing the module and its contents.

It is also necessary to correct workspace name from ‘**workspace.CustomerReviewsModule.Web’** to ‘**workspace.CustomerReviewsModuleWeb’** in *CustomerReviewsModule.Web\Scripts\CustomerReviewsModule.Web.js* file.

Now we are ready to compile and run the application.

## Connect new module with the platform

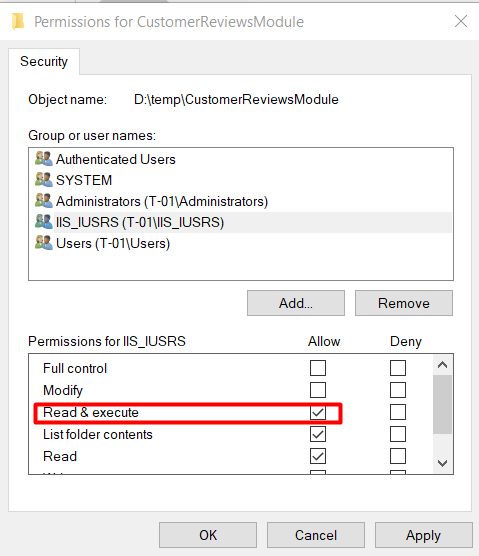
Now, we need to tell the platform that we added a new module. For that, we’ll connect our newly created project folder to the Platform Manager ~/Modules via the symbolic link:

1. Run Command Prompt as an administrator
2. Navigate to the physical location folder of Manager's ~/Modules virtual directory
3. Run the following command:

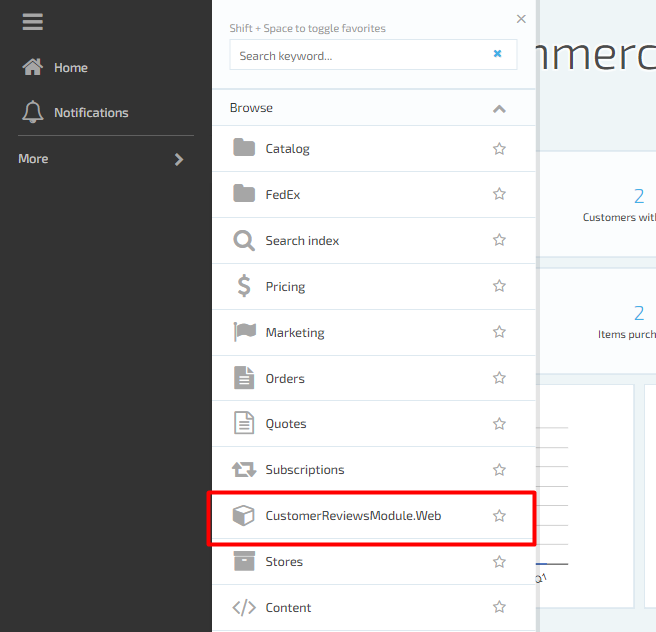
|  |
| --- |
| mklink /d **CustomerReviewsModule** **<full\_path\_to\_ CustomerReviewsModule \_project>** |

Also, we should check and set appropriate permissions for the newly created module folder, so that IIS can access it.

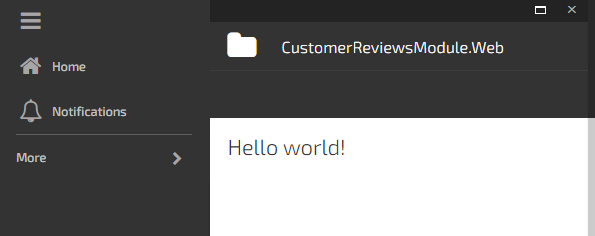
Here, for the “IIS\_IUSRS” group the required permission is “Read & execute”:



Compile solution and restart IIS (use iisreset.exe command). After that, our module should appear in Platform Manager. Open it in browser to check how new module looks like.



Click on **CustomerReviewsModule.Web** and we should see “Hello world” blade



## Compile and debug 12:20 - 17:08

When we run the code, we may notice some errors. To find out the cause, we should **debug** our code. For that we can use breakpoints.

To debug C# code at run-time we have to attach debugger to IIS instance

In Visual Studio:

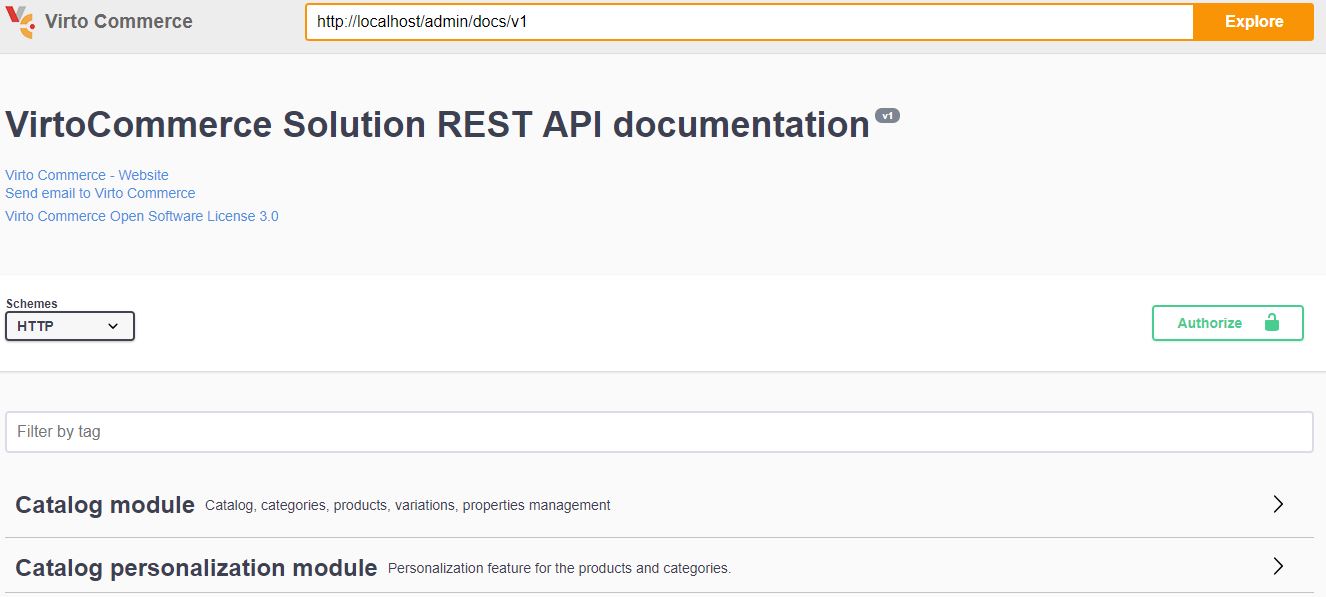
1. Click "Debug" from the menu bar
2. Click "Attach to Process"
3. Check the "Show processes from all users" checkbox in the bottom left corner
4. Select aspnet\_wp.exe, w3p.exe, or w3wp.exe from the process list
5. Click "Attach"

To debug JS code at run-time we use special debugging tools in browser. You can read more about Chrome debug tools and how to debug any JS issue in this [article](https://javascript.info/debugging-chrome)

## Debugging module Rest API

### Swagger API

We can test our API endpoints by using “REST API documentation” (Swagger) UI. Browse **[localhost/admin/docs/ui/index]** URL:

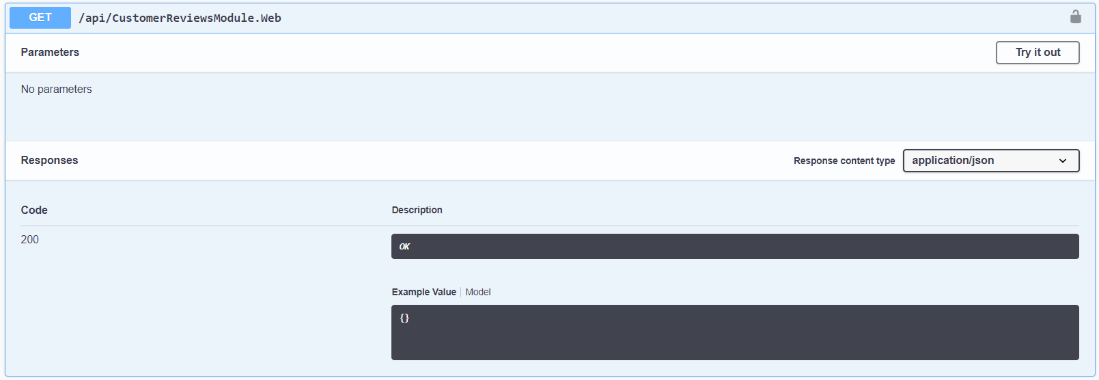


Here we can find all the API methods, exposed by Platform and installed modules as well.

### Testing module Rest API Endpoints

Click on “Sample Customer reviews module” to see the available endpoints.

When the new module is generated from a template, there is only one endpoint **api/CustomerReviewsModule.Web** included, returning “Hello, world!”:



# The recommended structure for VC module solution

“Customer review” solution consists of 4 logically divided parts (projects):

* **CustomerReviewsModule.Core** – this is where we keep the models and abstractions of our services. **CustomerReviewsModule.Core** projecthas following folder structure:
  + Model.
  + Services.
* **CustomerReviewsModule.Data** – here we can find all the service implementations, repositories, entity models, migration data and configurations. **CustomerReviewsModule.Data** projecthas following folder structure:
  + Migrations.
  + Model.
  + Repositories.
  + Services.
* **CustomerReviewsModule.Web** – contains the module definition, WEB API, Scripts and Localization resources. **CustomerReviewsModule.Web** projecthas following folder structure:
  + Controllers:
    - API.
  + Scripts:
    - blades.
    - Resources.
  + Security.
  + Content.
* **CustomerReviewsModule.Test** – for testing the service and repository layer methods with Unit test.

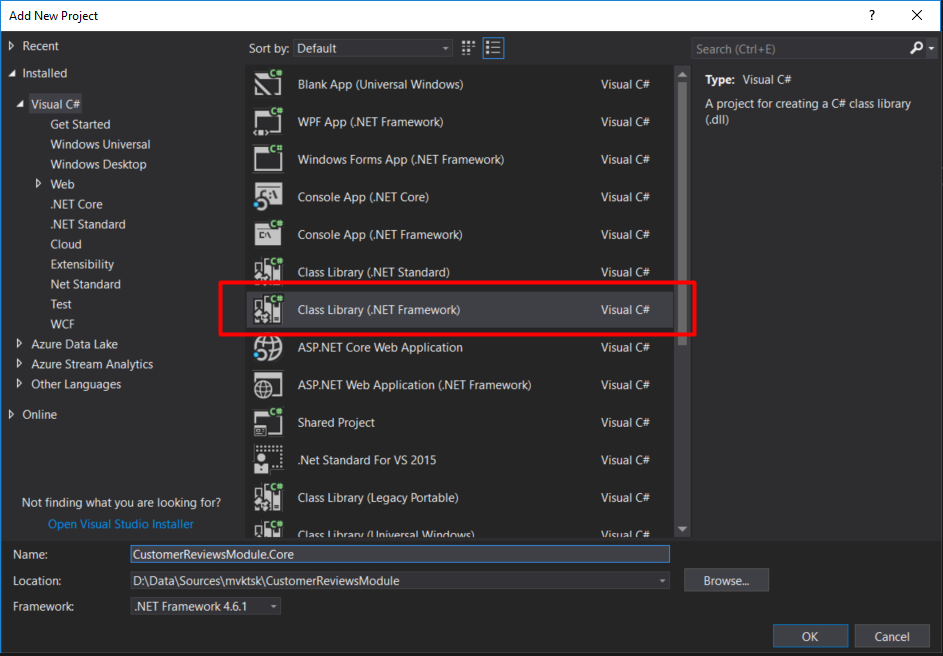
# Core project 17:22 - 23:56

All the abstractions are defined in the **Core** project.

## Prerequisites

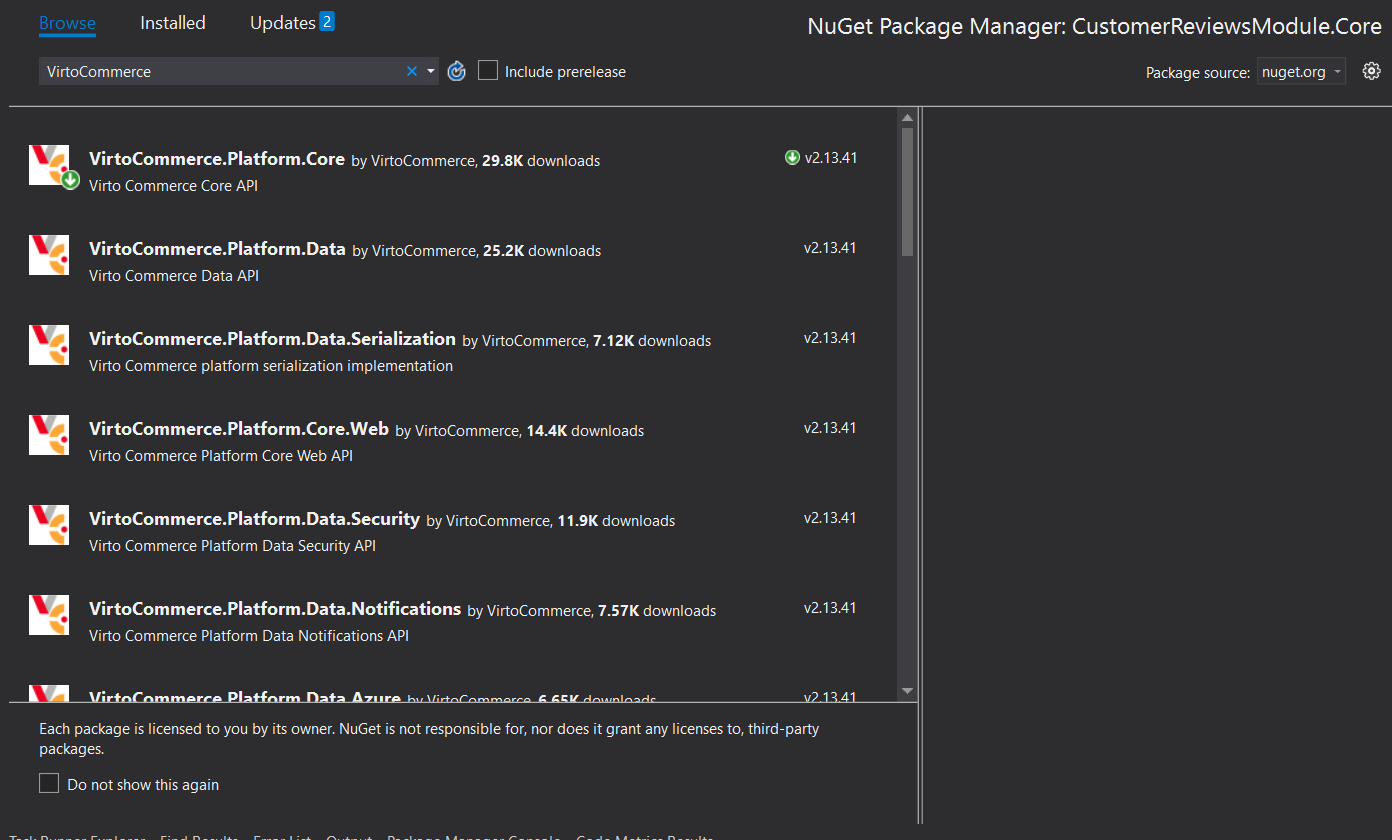
### Adding new project

Add new **CustomerReviewsModule.Core** project to the solution. In Visual Studio, go to **Solution Explorer,** choose **CustomerReviewsModule** solution,click right mouse button **Add > New Progect**. Search for **Class Library (.NET Framework)**.



### Installing NuGet packages

We need to install **VirtoCommerce.Platform.Core** and **VirtoCommerce.Domain** NuGet packages. In Visual Studio, go to **Solution Explorer,** choose **CustomerReviewsModule.Core** project, click right mouse button **Manage NuGet Packages**.



### Folder structure

And finally, we need to create **Model** and **Services** folders in **CustomerReviewsModule.Core** project. In Visual Studio, go to **Solution Explorer,** choose **CustomerReviewsModule.Core** project, click right mouse button **Add>New Folder**.

## Domain models

All domain models should be located in **CustomerReviewsModule.Core\Model** folder.

### Entity

As a base domain class, we have **CustomerReview** entity:

|  |
| --- |
| namespace CustomerReviews.Core.Model  {      public class CustomerReview : AuditableEntity      {          public string AuthorNickname { get; set; }          public string Content { get; set; }          public bool IsActive { get; set; }          public string ProductId { get; set; }      }  } |

It contains all the mandatory data related to customer’s review: The author’s nickname, review content, the status(active/inactive) and product ID which user reviewed.

### Search criteria

We should define some criteria to search for reviews such as **CustomerReviewSearchCriteria**:

|  |
| --- |
| public class CustomerReviewSearchCriteria : SearchCriteriaBase  {      public string[] ProductIds { get; set; }      public bool? IsActive { get; set; }  } |

In this case, the client can search by products and review status.

## Services

In order to use our models, we need services. In **CustomerReviewsModule.Core\Services folder** we’ll define the abstractions for customer’s review services.

For example, the **ICustomerReviewService** interface:

|  |
| --- |
| public interface ICustomerReviewService  {      CustomerReview[] GetByIds(string[] ids);        void SaveCustomerReviews(CustomerReview[] items);        void DeleteCustomerReviews(string[] ids);  } |

Here we define methods to get, save and delete customer reviews.

And a separate service is for the search. **ICustomerReviewSearchService**:

|  |
| --- |
| public interface ICustomerReviewSearchService  {      GenericSearchResult<CustomerReview> SearchCustomerReviews(CustomerReviewSearchCriteria criteria);  } |

# Data project (Persistence layer 24:10 – 26:39)

For the persistence or Data Access Layer (DAL) we have a separate project called **Data**. Here we implement all the interfaces defined in **Core** project. Moreover, it contains all the persistence and data access abstractions, and mappings as well.

## Prerequisites

### Adding new project

Add new **CustomerReviewsModule.Data** project to the solution. In Visual Studio, go to **Solution Explorer,** chooseCustomerReviewsModule solution,click right mouse button **Add > New Progect**. Search for **Class Library (.NET Framework)**.

### Installing NuGet packages

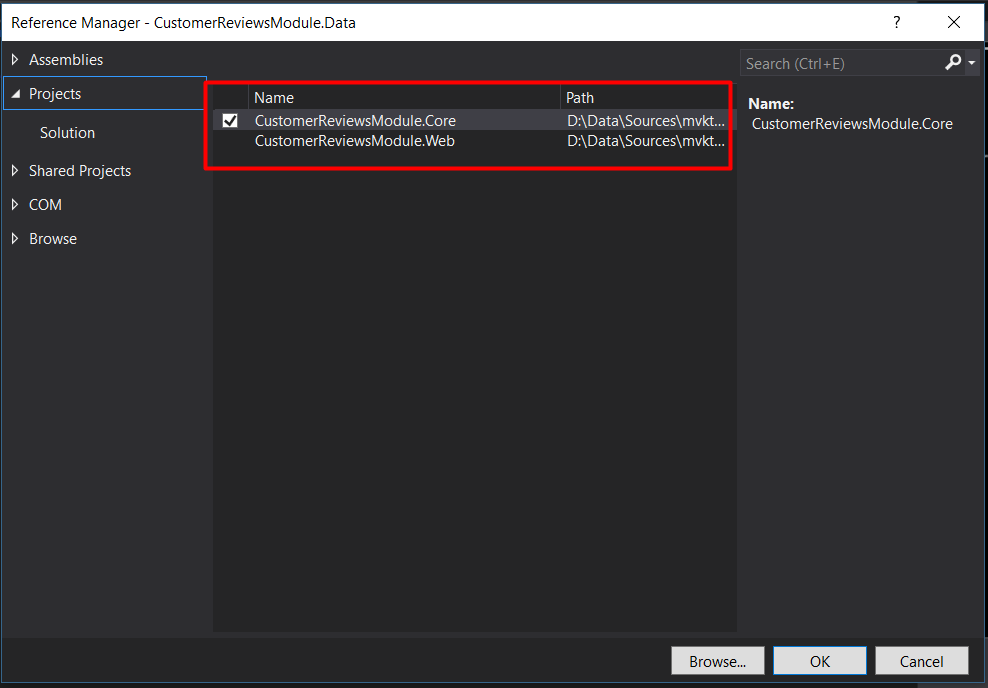
After that, install **VirtoCommerce.Platform.Core, VirtoCommerce.Platform.Data**  and **VirtoCommerce.Domain** NuGet packages. In Visual Studio, go to **Solution Explorer,** choose **CustomerReviewsModule.Data** project, click right mouse button **Manage NuGet Packages**.

### Folder structure

Then create 3 new folders in the project: **Migrations**, **Model,** **Repositories** and **Services**. In Visual Studio, go to **Solution Explorer,** choose **CustomerReviewsModule.Data** project, click right mouse button **Add>New Folder**.

### Project references

And finally add reference to **CustomerReviewsModule.Core** project. In Visual Studio, go to **Solution Explorer,** choose **CustomerReviewsModule.Data>References**, click right mouse button **Add Reference >Projects** check **CustomerReviewsModule.Core**.



## Entity and mapping

Now we need to create data access layer models and map it to the base domain class. All data access layer models should be located in **CustomerReviewsModule.Data\Model** folder. The example of the CustomerReview data access layer model implementation you can find in sample [repository](https://github.com/VirtoCommerce/vc-samples/tree/master/CustomerReviews).

## Repositories

As a new type of abstraction, Data project has repositories defined in it. For example, we can notice ICustomerReviewRepository used in CustomerReviewService.

|  |
| --- |
| public interface ICustomerReviewRepository : IRepository     {         IQueryable<CustomerReviewEntity> CustomerReviews { get; }           CustomerReviewEntity[] GetByIds(string[] ids);         void DeleteCustomerReviews(string[] ids);     } |

**ICustomerReviewRepository** interface contains only the declaration of the methods, properties. Implementation of the interface **ICustomerReviewRepository** uses Entity Framework under the hood. The Interface and the implementation can find in sample [repository](https://github.com/VirtoCommerce/vc-samples/tree/master/CustomerReviews).

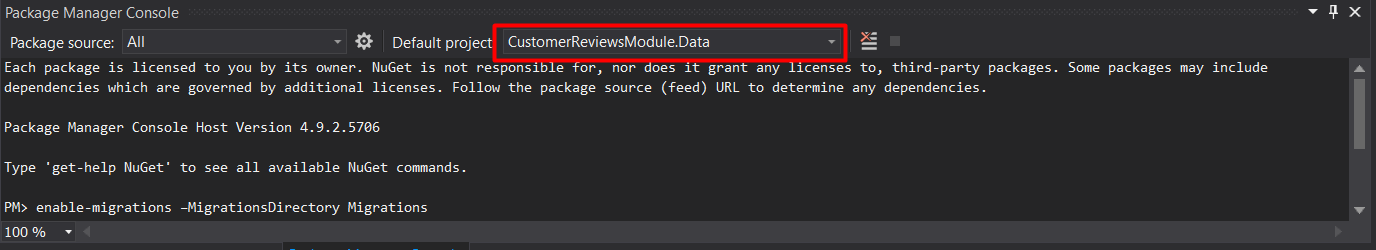
The interface and its implementation should be located in **CustomerReviewsModule.Data\Repositories** folder.

## Services

The implementations of the services that was defined early in **CustomerReviewsModule.Core** project should be located in **CustomerReviewsModule.Data\Services** folder. The example of the services implementation you can find in sample [repository](https://github.com/VirtoCommerce/vc-samples/tree/master/CustomerReviews).

## Migrations 34:47 – 36:39

In order to define the data access layer based on Entity Framework we’ll use the package manager tools in Visual Studio. From the VS top menu chooses **Tools>Module Package Manager>Package Manager Console**.



There is a command to enable migrations:

|  |
| --- |
| enable-migrations –MigrationsDirectory Migrations |

In the dropdown menu choose the **CustomerReviewsModule.Data** project. The command run will create a new migration configuration.

Note, that by default the automatic migrations are switched off:

|  |
| --- |
| AutomaticMigrationsEnabled **=** false |

### Initial migration

Now, let’s generate the initial migration. For that we’ll use this script:

|  |
| --- |
| Add-Migration Initial -ConnectionString "Data Source**=**(local);Initial Catalog**=**VirtoCommerce2;Persist Security Info**=**True;User ID**=**virto;Password**=**virto;MultipleActiveResultSets**=**True;Connect Timeout**=**420" -ConnectionProviderName "System.Data.SqlClient" |

This will generate the **dbo.CustomerReview** migration script. Keep in mind, that in case “Down migration” developer should take care of removing the module with all its dependencies.

Now the Data Access Layer is ready.

# Web project

## Prerequisites

### Installing NuGet packages

Install **VirtoCommerce.Platform.Core.Web, VirtoCommerce.Platform.Data**  and **VirtoCommerce.Domain** NuGet packages. In Visual Studio, go to **Solution Explorer,** chooseCustomerReviewsModule.Data project, click right mouse button **Manage NuGet Packages**.

### Folder structure

In **CustomerReviewsModule.Web** project we need to additionally create **Security** folder. In Visual Studio, go to **Solution Explorer,** choose **CustomerReviewsModule.Web** project, click right mouse button **Add>New Folder**.

### Project references

And finally add reference to **CustomerReviewsModule.Core** and **CustomerReviewsModule.Data** projects. In Visual Studio, go to **Solution Explorer,** choose **CustomerReviewsModule.Web>References**, click right mouse button **Add Reference >Projects** check **CustomerReviewsModule.Core** and **CustomerReviewsModule.Data**.

## WEB API layer 28:31 - 33:49

Finally, we need a way to access and use our services. For that reason, we define a **WEB API** layer. Search, CRUD and module specific operations are exposed there.

For example, let’s have a look at searching endpoint:

|  |
| --- |
| [HttpPost]  [Route("search")]  [ResponseType(typeof(GenericSearchResult<CustomerReview>))]  [CheckPermission(Permission = PredefinedPermissions.CustomerReviewRead)]  public IHttpActionResult SearchCustomerReviews(CustomerReviewSearchCriteria criteria)  {      GenericSearchResult<CustomerReview> result = \_customerReviewSearchService.SearchCustomerReviews(criteria);       return Ok(result);  } |

As we see, there isn’t much code in our method. We’re just calling the customer review search Service directly. Of course, here as well, we’ll use the ICustomerReviewRepository abstraction and not the service implementation itself.

The API endpoints are all accessed from the JavaScript code written in the CustomerReviewsModule.

## Authorization 44:20 – 46:25

There are predefined permissions coming from the module core. We keep the hard-coded list of permissions in PredefinedPermissions class:

|  |
| --- |
| public static class PredefinedPermissions  {      public const string CustomerReviewRead = "customerReview:read",                   CustomerReviewUpdate = "customerReview:update",                   CustomerReviewDelete = "customerReview:delete";  } |

With them we control the API access allowing to read, update or perform delete operation. We keep permission checks in frontend part as well.

## Declaring new permissions in module.manifest

After permissions had hard-coded in PredefinedPermissions class we have to declare them in module manifest.

|  |
| --- |
| <module>  ....      <permissions>          <group name="Customer Reviews">              <permission id="customerReview:read" name="Read Customer Reviews" />              <permission id="customerReview:update" name="Update Customer Review" />              <permission id="customerReview:delete" name="Delete Customer Review" />          </group>      </permissions>  </module> |

You can read more about permissions in [Working with platform security](https://virtocommerce.com/docs/vc2devguide/working-with-platform-manager/basic-functions/working-with-platform-security)

## WEB API endpoints protection

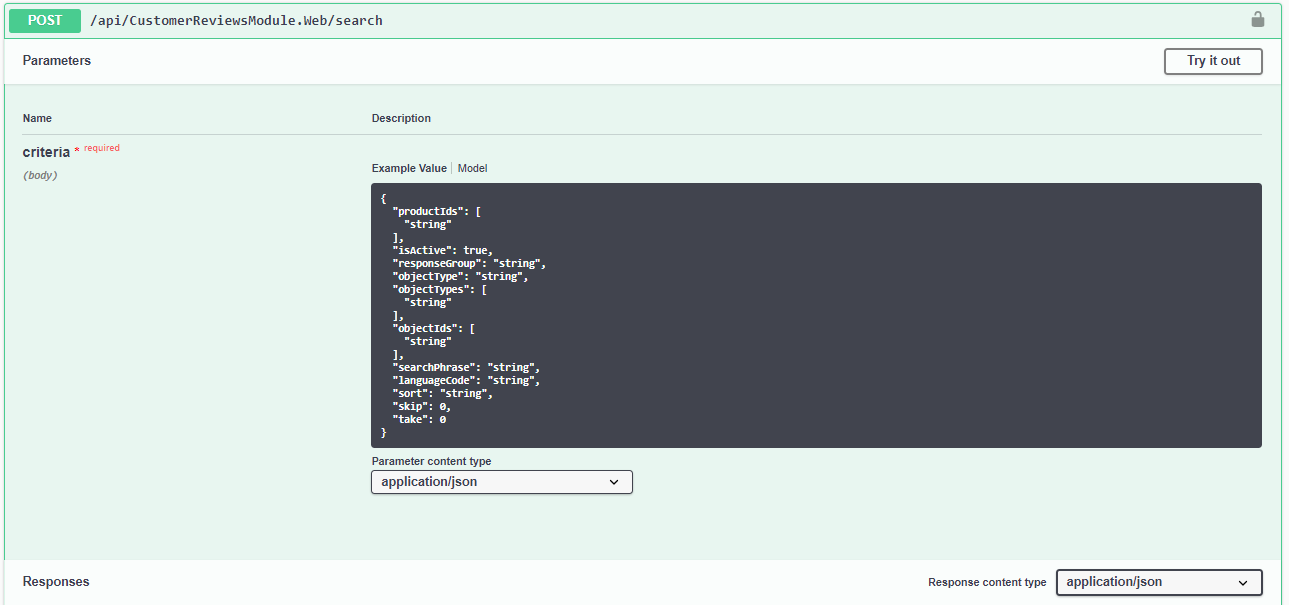
From the backend side we usually protect our methods with **CheckPermission**:

|  |
| --- |
| [CheckPermission(Permission = PredefinedPermissions.CustomerReviewRead)] |

## Testing Rest API Endpoints in Swagger

Besides the JavaScript, we can test our API endpoints, with the Swagger. Open the Swagger interface URL: **[localhost://admin/docs/ui/index]** and click on “Customer reviews module” to see the available endpoints.

We can test the search functionality for instance. Under the **api/CustomerReviewsModule.Web/search** endpoint we can create simple or nested, compound criteria. It accepts the criteria as a simple object. After providing the criteria hit the “try it out” button.



## The Main class 33:49 – 33:57

So far, we’re good because all the classes we used here are registered inside of a base class **Module**. Which is the base class for the entire module. In this case, it’s Customer Reviews module.

### Dependency Injection 38:05 - 39:31

Here, in our project we are using interfaces in our services. That’s because we always use a dependency injection. Inside of our Initialize() method in the Main class we register the Service implementations of the module and link them with the interfaces:

|  |
| --- |
| \_container.RegisterType<ICustomerReviewSearchService, CustomerReviewSearchService>();  \_container.RegisterType<ICustomerReviewService, CustomerReviewService>(); |

This is called a loose coupling mechanism.

### Module settings 46:28 – 48:08

We need module settings to check if a specific feature is enabled, or to determine which search engine to use in our module.

We define module settings with the Setting manager. First, we register the **ISettingsManager:**

|  |
| --- |
| var settingManager = \_container.Resolve<ISettingsManager>(); |

After, we can use it to create key, value associations:

|  |
| --- |
| settingManager.RegisterModuleSettings("VirtoCommerce.Store", storeSettings); |

## Define management UI 39:35 – 44:08

First, let’s open a product list in a catalog directory of Virto Commerce application UI. And click to one of the products in the products list. This will open the blade of product details. To add a review manually, we’ll use the Swagger API again. All we need is to copy the **product\_id** from here and make a request with **http:localhost/admin/api/CutomerReviewsModule** endpoint in Swagger UI.

Now, we can see the added customer review item in Virto Commerce platform menu.

We can add any JavaScript code to our module along with many other [types of extensions](https://virtocommerce.com/docs/vc2devguide/working-with-platform-manager/extending-functionality).

JavaScript code you can find in folder Scripts:

* CustomerReviews.Web.js Contains an AngularJS module definition.
* blades
  + helloWorld\_blade1.js Contains an AngularJS controller for the Hello World blade.
  + helloWorld\_blade1.tpl.html Contains a markup for the Hello World blade.
* resources
  + CustomerReviews.WebApi.js Contains definitions for the resources (services) available inside AngularJS module.

Let's add a new [widget](https://virtocommerce.com/docs/vc2devguide/working-with-platform-manager/basic-functions/widgets) to product details blade. This widget will display the count of Customer Reviews for the product.

First, let’s add available resources definition in CustomerReviews.WebApi.js

|  |
| --- |
| \_angular.module('CustomerReviews.Web')  .factory('CustomerReviews.WebApi', ['$resource', function ($resource) {      return $resource('api/customerReviews', {}, {          search: { method: 'POST', url: 'api/customerReviews/search' },          update: { method: 'PUT' }      });  }]); |

Then, crate new folder “widgets” in Scripts folder and add 2 files:

* customerReviewWidget.js Contains an AngularJS controller for the CustonerReviews widget.
* customerReviewWidget.tpl.html Contains a markup for the CustonerReviews widget.

Add actual code to the files.

Then, in folder blade rename files to reviews-list.js and reviews-list.tpl.html (do not forget add actual code).

Actual code for all files you can find in sample [repository](https://github.com/VirtoCommerce/vc-samples/tree/master/CustomerReviews).

### Localization

Virto Commerce uses English as a default language. To localize a module content, we need to create a basic translation file for one of the languages. The file should be placed in module’s Localizations subfolder. You can read more about localization in [Localization implementation](https://virtocommerce.com/docs/vc2devguide/working-with-platform-manager/localization-implementation) article.

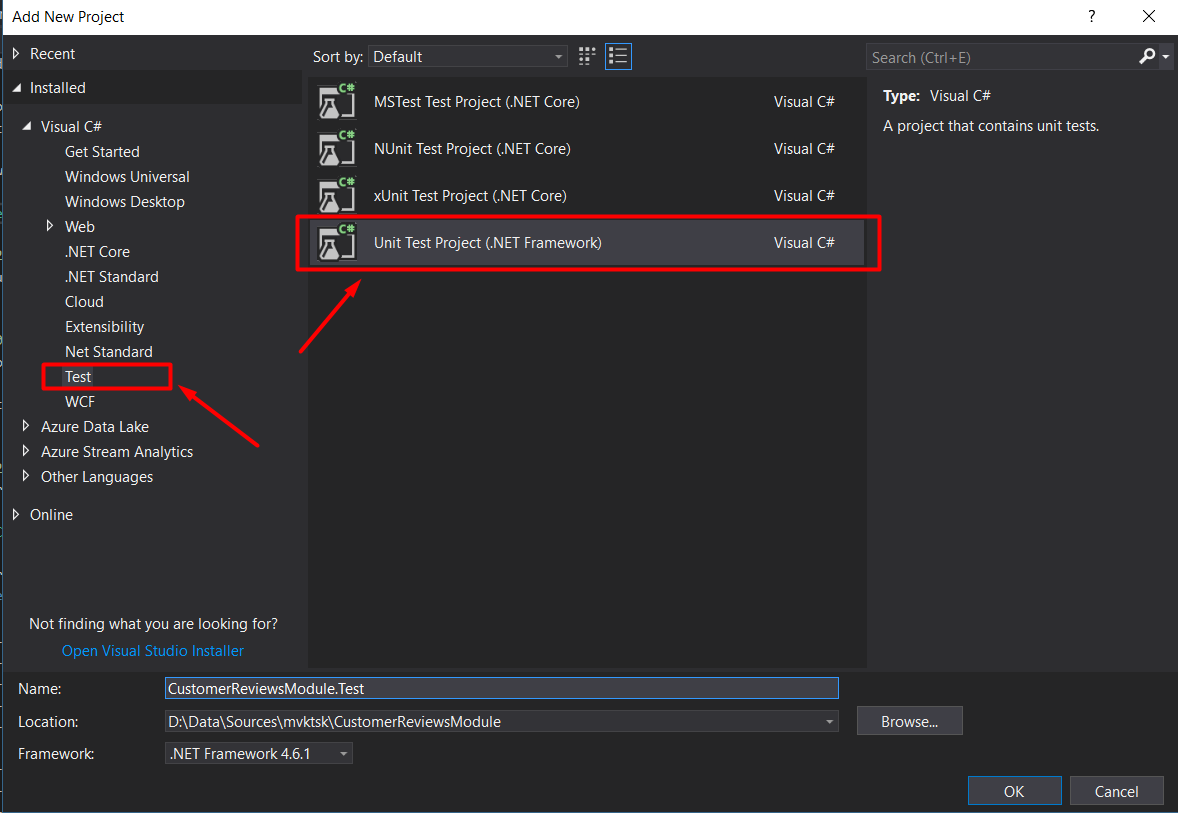
# Test project

Now we are ready to create test project for **CustomerReviewsModule.** We are using xUnit testing framework for test project.

## Prerequisites

### Adding new project

Add new **CustomerReviewsModule.Test** project to the solution. In Visual Studio, go to **Solution Explorer,** chooseCustomerReviewsModule solution,click right mouse button **Add > New Progect**. Search for **Unit test Project (.NET Framework)**.



### Installing NuGet packages

After that, install **VirtoCommerce.Platform.Core, VirtoCommerce.Platform.Data**  and **VirtoCommerce.Domain, VirtoCommerce.Platform.Testing, xunit, xunit.runner.console, xunit.analyzers** NuGet packages. In Visual Studio, go to **Solution Explorer,** choose **CustomerReviewsModule.Test** project, click right mouse button **Manage NuGet Packages**.

## Tests implementation

We recommend to use "Unit testing best practices with .NET Core and .NET Standard"

<https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices> in tests development.

Actual code for **CustomerReviewsModule.Test** project you can find in sample [repository](https://github.com/VirtoCommerce/vc-samples/tree/master/CustomerReviews).

# Pack and release/deployment 48:14 – 53:36

To deploy our module to other platforms we’ll need the package manager tool. The command for that is:

|  |
| --- |
| compress-module |

After execution, the command will create the .zip module package. That’s it! We can deploy it to the other environment.

Let’s try it in **admin-demo** application. Go to **Modules>Advanced>Install/update module from file**. Now upload the .zip file that we created with package manager. After checking, the upload details hit the **Install** button. **Restart** the application.

Under the hood, the application will unzip the file, check the structure, and copy all the required classes to the Modules folder of the current application.