**SnipCart intergration with .NET Core and VS Code**

**Goal** : The goal of this document is to list the steps to integrate a functional cart system into any website using SnipCart.

https://docs.snipcart.com/configuration/json-crawler

Most times, the value you'll specify for this attribute will be the unique URL where you're selling the item. However, some merchants could sometimes slightly more complex scenarios.

**First things first: Setting up our development environment**

* Install Visual Studio Code: <https://code.visualstudio.com/>
* Install .NET Core SDK 1.0 RC3 build 004530: <https://github.com/dotnet/core/blob/master/release-notes/rc3-download.md>
* Install Node: <https://nodejs.org/en/>
* Working directory: aspnetcoreapp/snipcart
* Get node packages: npm install –g yo bower grunt-cli gulp generator-aspnet
* Project creation with Yeoman: yo aspnet > Web Application Basic [without Membership and Authorization] > Bootstrap 3.3.7 > snipcart
* Project build with CLI (will run at http://localhost:5000):
  + cd snipcart
  + dotnet restore
  + dotnet build
  + dotnet run
* Get VS Code extensions: C# and .Net Core Project Manager (Nuget)

At this point, everything should be working and our development environment is finally set up.

Let the fun begin!

**Reflecting on life (not really): What kind of data will our website display?**

While starting a new development project is always very exciting, it’s also crucial to kick things off on the right foot.

A question that has to be answered early on is: *What kind of data is our website going to serve*? Even though the chances of getting the answer exactly right on our first shot is extremely slim, only having a general idea will help us go a long way.

Since the sole purpose of this document is to integrate an e-commerce shopping cart to a website, we can say with confidence that the chances of us using some kind of **product** are pretty good.

Knowing that we will be dealing with products in our project, now is the time to determine what information we will need to have displayed to the user. Usually, products will have these:

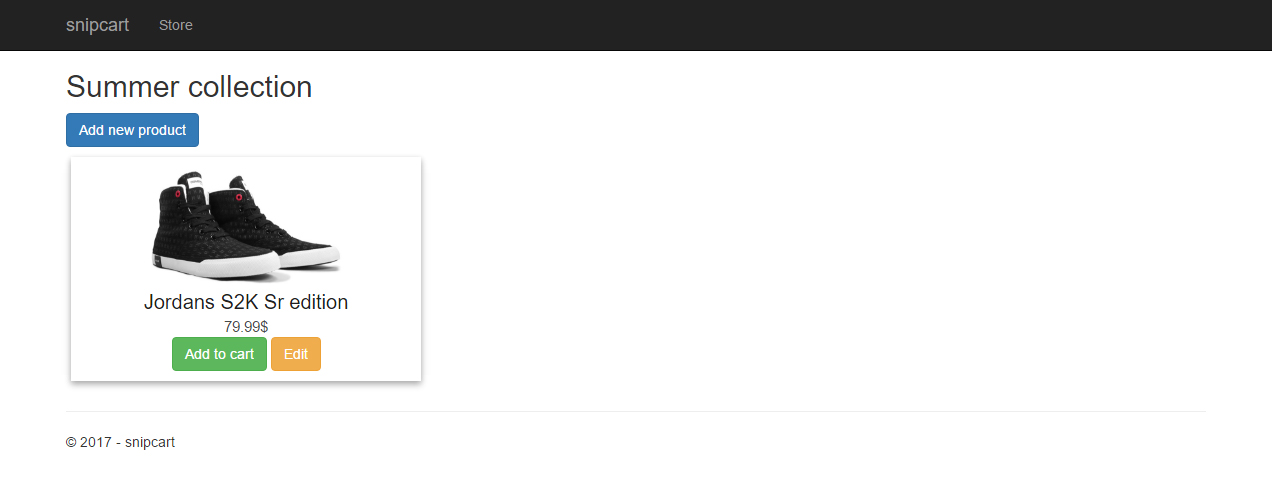
* A title
* A description
* A price
* Picture

Now that we know some of the info that every product will have, we can proceed to prototyping our storefront.

**Prototyping a nice static storefront**

Knowing that our website is simple and will only be showing some data about products, namely a title, description, price and one picture, we can quickly and efficiently start prototyping our storefront with dummy data.

We will also strategically insert placeholder buttons in the page to add and edit (and delete, which will appear when the edit window will open) products.



**Creating the model**

As we know, we will only be needing a single model for this project, which will be named “Product”. To set this up, we will create a new folder in the root of the project named “Models”, which we will then create a Product.cs file within it.

Our model will then need these attributes:

* Id (int)
* Title (string)
* Description (string)
* Price (double)
* Image (string)

Here is the model in code:

using System.Collections.Generic;

namespace snipcart.Models

{

public class Product

{

public int Id { get; set; }

public string Title { get; set; }

public string Description { get; set; }

public double Price { get; set; }

public string Image { get; set; }

public List<Product> Products { get; set; }

}

}

**EntityFrameWork Core In-Memory database**

For simplicity, we will be using an in-memory database, which will simulate calls to a real EntityFrameWork database.

To set it up, let’s create a file called InMemoryDB.cs within the Models folder and insert the following code:

using Microsoft.EntityFrameworkCore;

using snipcart.Models;

namespace snipcart

{

public class InMemoryDB: DbContext

{

public InMemoryDB (DbContextOptions<InMemoryDB> options)

: base(options)

{

}

public DbSet<Product> Products { get; set; }

}

}

Next, in the Startup.cs file of the root of the project, use our In-Memory database by adding the following code to the top of the file:

using snipcart;

Now, let’s tell our project to use this service by altering the Configure method to the following:

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)

{

loggerFactory.AddConsole(Configuration.GetSection("Logging"));

loggerFactory.AddDebug();

var context = app.ApplicationServices.GetService<InMemoryDB>();

AddTestData(context);

…

… And by adding the AddTestData() method at the end like so:

private static void AddTestData(InMemoryDB context)

{

var testProd1 = new snipcart.Models.Product

{

Id = 1,

Title = "Jordans S2K Sr edition",

Description = "The Jordans S2K Sr edition is the best bang for your buck.",

Price = 97.99,

Image = "http://simpleproductphotography.com/wp-content/uploads/2016/06/huf-converse-product-red-skidgrip-1.jpg"

};

var testProd2 = new snipcart.Models.Product

{

Id = 2,

Title = "Lamborghini Huracan",

Description = "The Lamborghini Huracan is definitely the best supercar for the money.",

Price = 278999.99,

Image = "http://1.bp.blogspot.com/-Gaj30dheGzE/VfGQL2uD0\_I/AAAAAAAAWJ4/IOomh6RXDpY/w800/lambo-huracan-roadster-rendering-ts-4.jpg"

};

context.Products.Add(testProd1);

context.Products.Add(testProd2);

context.SaveChanges();

}

In a nutshell, the code that we’ve just added in the Startup.cs file will do the following when the project starts up:

* Tell our project we want to use the snipcart namespace
* Tell our project to use the InMemoryDB service, which is our In-Memory database
* Create two dummy entries in our In-Memory database

At this point, we now have access to the InMemoryDB and its contents (exciting!).

**Using our database**

To use our database, we simply need to use the snipcart namespace like the following:

Using snipcart;

And create a member variable to access our database like so:

private readonly InMemoryDB \_context;

Now, we will want to initialize our \_context variable in the HomeController’s constructor so we can use it in our methods:

public HomeController(InMemoryDB context)

{

\_context = context;

}

Next, in any controller or method, we will be able to use the database in the following way:

var prodList = from p in \_context.Products select p;

or…

var prod = (from p in \_context.Products where p.Id.Equals(id) select p).FirstOrDefault();

**Controllers**

In this project, we will need to have view a certain product and have CRUD functions, which stands for Create, update and delete.

Knowing this, we will be creating three new controller actions in the Home controller (for simplicity reasons), one for each view. On top of that, we will be needing some GET methods to create, edit and delete our products. Here it goes:

public IActionResult AddNew()

{

return View();

}

public IActionResult Edit(int id)

{

var prod = (from p in \_context.Products where p.Id.Equals(id) select p).FirstOrDefault();

return View(prod);

}

public IActionResult Product(int id)

{

var prod = (from p in \_context.Products where p.Id.Equals(id) select p).FirstOrDefault();

return View(prod);

}

[HttpGetAttribute]

public string CreateNewProduct(string title, string desc, double price, string image){

var testProd = new snipcart.Models.Product

{

Title = title,

Description = desc,

Price = price,

Image = image

};

\_context.Products.Add(testProd);

\_context.SaveChanges();

return "done";

}

[HttpGetAttribute]

public string EditProduct(int id, string title, string desc, double price, string image){

var oldProd = (from p in \_context.Products where p.Id.Equals(id) select p).FirstOrDefault();

oldProd.Title = title;

oldProd.Description = desc;

oldProd.Price = price;

oldProd.Image = image;

\_context.SaveChanges();

return "done";

}

[HttpGetAttribute]

public string DeleteProduct(int id){

var prod = (from p in \_context.Products where p.Id.Equals(id) select p).FirstOrDefault();

\_context.Products.Remove(prod);

\_context.SaveChanges();

return "done";

}

**Views**

As for the views, we will need the AddNew.cshtml, Edit.cshtml and product.cshtml files.

**To populate the views with the data** from our database, we need to specify the model used at the top:

@model IEnumerable<snipcart.Models.Product>

Or if we’ve only got sent a single product from our controller:

@model snipcart.Models.Product

And then use the @Model.paramWanted to display the info needed. Alternatively, if more than one record has been sent, a may be used to iterate through the different products:

@foreach(var prod in Model){

… contents …

}

**To create a GET request to our methods**, we need to use JavaScript and JQuery AJAX to initiate these actions.

Here is an example of how any request will be handled:

$(".btn\_\_edit-product-btn").click(function(e){

e.preventDefault();

var title = $("#prod-title").val();

var description = $("#prod-desc").val();

var price = $("#prod-price").val();

var image = $("#prod-image").val();

$.ajax({

url: "/Home/EditProduct",

type: "GET",

datatype: 'json',

data: {

id: @Model.Id,

title: title,

desc: description,

price: price,

image: image

},

success: function (data, status, jqXHR) {

$(".form-edit-product").prepend(

'<div class="alert alert-success">' +

'<strong>Your product has been successfully modified!</strong>' +

'</div>'

);

},

error: function (jqXHR, status, err) {

$(".form-edit-product").prepend(

'<div class="alert alert-danger">' +

'<strong>An error has been detected in your request. Please try again.</strong>' +

'</div>'

);

}

})

});

In this example used for the Edit Product button, we can see that the request is done via the $.ajax block. Parameters are sent using the **data** object definition and the targeted method is set using the **url** object definition. In the above example, we want to call the EditProduct method from the Home controller and send it the id, title, desc, price and image parameters. We then have a **success** and **error** function callback that is called selectively if the request was successful or not. Data from the controller to the view can be retrieved on success or error if needed.

**Snipcart integration**

Integrating the Snipcart system is done fairly easily.

We only need to insert the JavaScript library , CSS styles and make sure that our project uses JQuery in our project, which will look like this:

CSS

<head>

…

<environment names="Development">

…

<link href="https://cdn.snipcart.com/themes/2.0/base/snipcart.min.css" type="text/css" rel="stylesheet" />

</environment>

<environment names="Staging,Production">

…

<link href="https://cdn.snipcart.com/themes/2.0/base/snipcart.min.css" type="text/css" rel="stylesheet" />

</environment>

</head>

JS

<body>

…

<environment names="Staging,Production">

…

<script

src="https://cdn.snipcart.com/scripts/2.0/snipcart.js"

id="snipcart"

data-api-key="my-api-key-here">

</script>

</environment>

…

</body>

The next step is to create buttons to initiate the Add To Cart action from Snipcart. To do this, once your products in your database successfully show on your screen, it’s only a matter of adding the attributes to your Add To Cart button’s html code with you dynamic data.

Here is an example of a button that adds the specified product to the cart:

<button

class="snipcart-add-item btn btn-success product-col\_\_product-add-to-cart-btn"

data-item-id="@Model.Id"

data-item-name="@Model.Title"

data-item-price="@Model.Price"

data-item-weight="0"

data-item-url="/Home/Product/@Model.Id"

data-item-image="@Model.Image"

data-item-description="@Model.Description">

Add to cart

</button>

Upon click of this button, the Snipcart cart will pop up and we will see the contents of the newly updated cart.

**Exciting!**

**Upgrades that could be made to this project**

Of course, this being a brief overview of the Snipcart e-commerce system integration, certain development “best practices” have been left out, namely:

* Adding a new product input validation (like price needs to be a “dot” ex: 9.95 and not 9,95)
* Editing a product input validation (like price **absolutely** needs to be entered with a “dot” ex. 9.95 and not 9,95). **When clicking the Edit button on an item, the pre-populated values will place a comma by default instead of a dot, so be warned that if it is not changed, the price will be changed to zero**.
* Deleting a product confirmation
* Setting up multiple controllers, one for every view. **This is important for maintainability.** Right now, only having a couple of views, it’s not really a problem. However we can imagine that if all our controller methods for every view were all in the same controller file, the file could end up very large and messy. **Normally, every view has its own controller**.