Platformus Documentation

Release 4.0.0

Dmitry Sikorsky

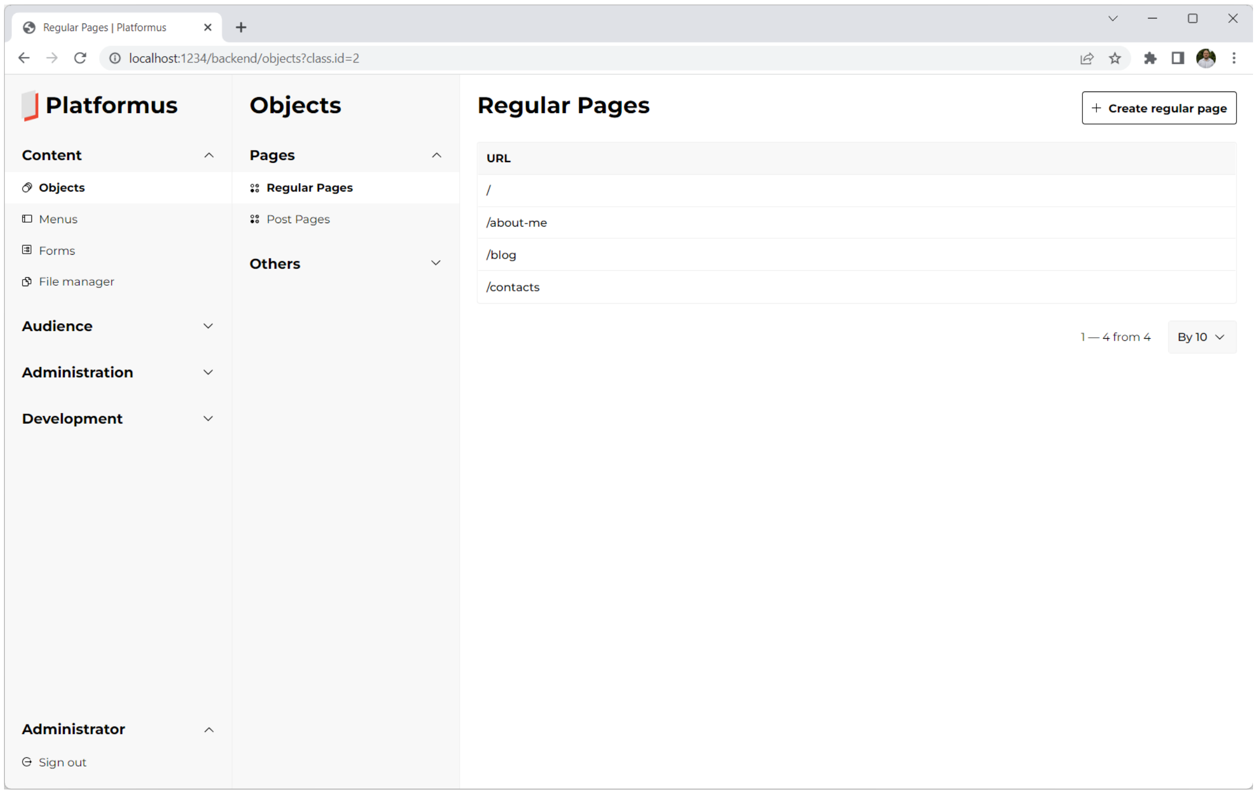
Feb 24, 2023

Contents

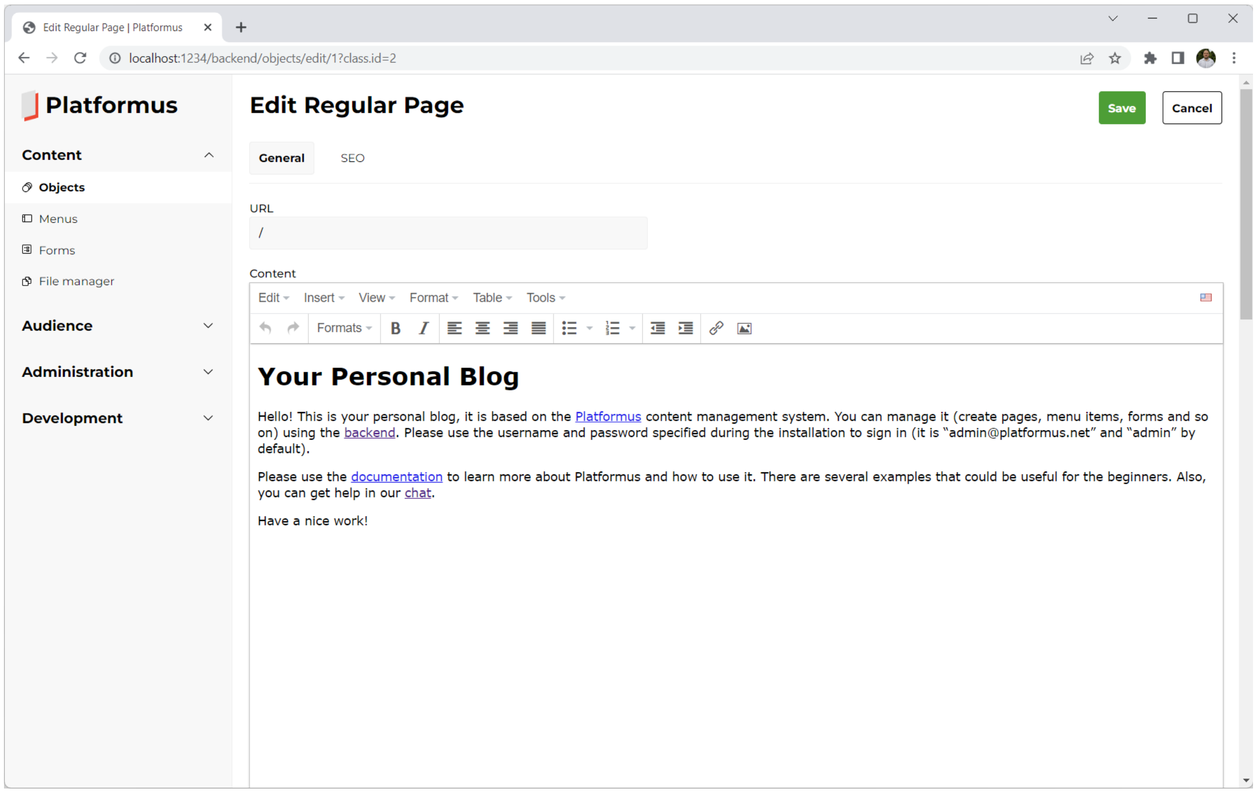
Platformus is free, open source, and cross-platform developer-friendly CMS based on ASP.NET Core, ExtCore framework, and Magicalizer.

It can be used for rapid development of the websites and admin panels for mobile and web applications. There are built-in features to describe, create, and deliver multilingual and multicultural content as HTML or JSON without writing code, but it is possible to develop custom extensions with the standardized user interface when performance is important. Wide range of the tag helpers makes it simpler and faster.

Platformus CMS is modular and extendable, it can run on different environments (Linux, Mac, Windows, and clouds) and supports different database types (PostgreSQL, SQLite, SQL Server).



URL
/
/about-me
/blog
/contacts



URL: /

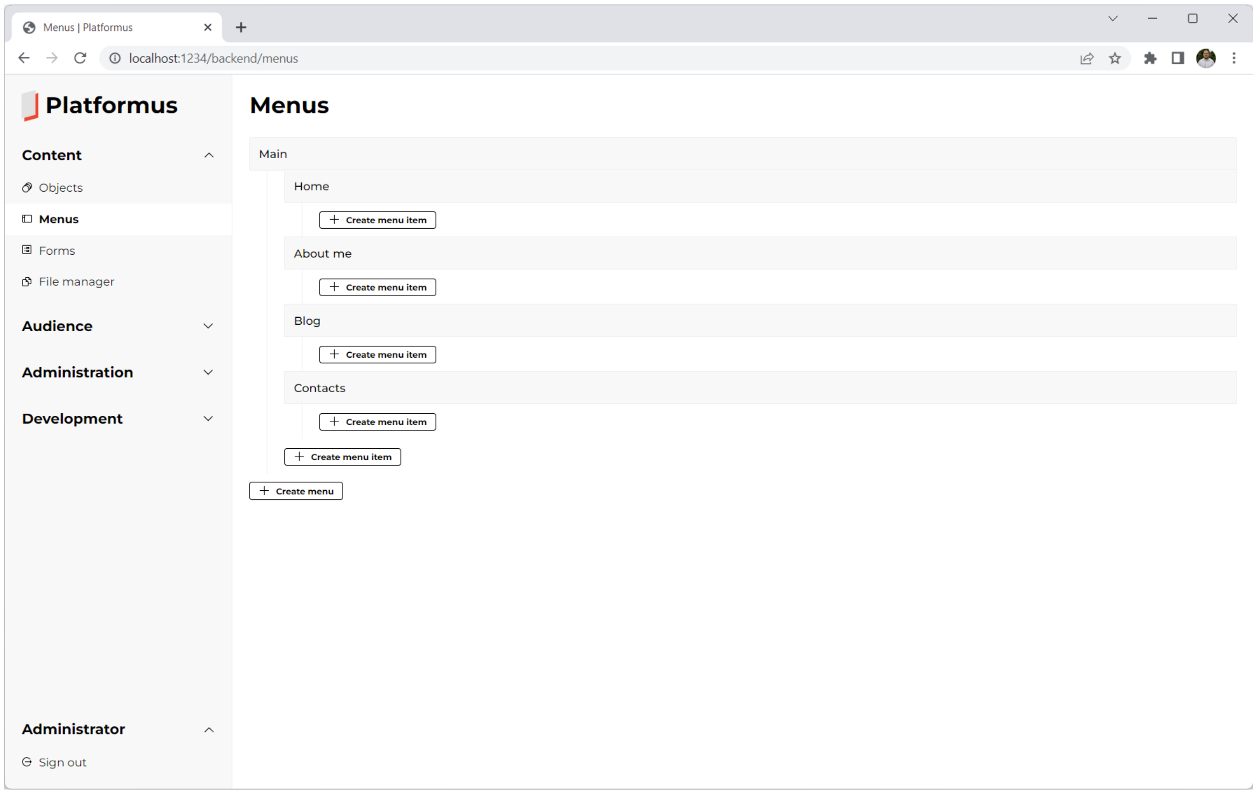
Content:

Your Personal Blog

Hello! This is your personal blog, it is based on the Platformus content management system. You can manage it (create pages, menu items, forms and so on) using the backend. Please use the username and password specified during the installation to sign in (it is "admin@platformus.net" and "admin" by default).

Please use the documentation to learn more about Platformus and how to use it. There are several examples that could be useful for the beginners. Also, you can get help in our chat.

Have a nice work!


Menus | Platformus
localhost:1234/backend/menus
Platformus
Content
Objects
Menus
Forms
File manager
Audience
Administration
Development
Administrator
Sign out
Menus
Main
Home
Create menu item
About me
Create menu item
Blog
Create menu item
Contacts
Create menu item
Create menu item
Create menu

CHAPTER 1

Contents

1.1 Getting Started

To start using Platformus CMS you need to either reference it as the NuGet packages (the preferred way) or the source code, prepare the database by executing the database scripts (with both schema and data), and create content.

1.1.1 Samples

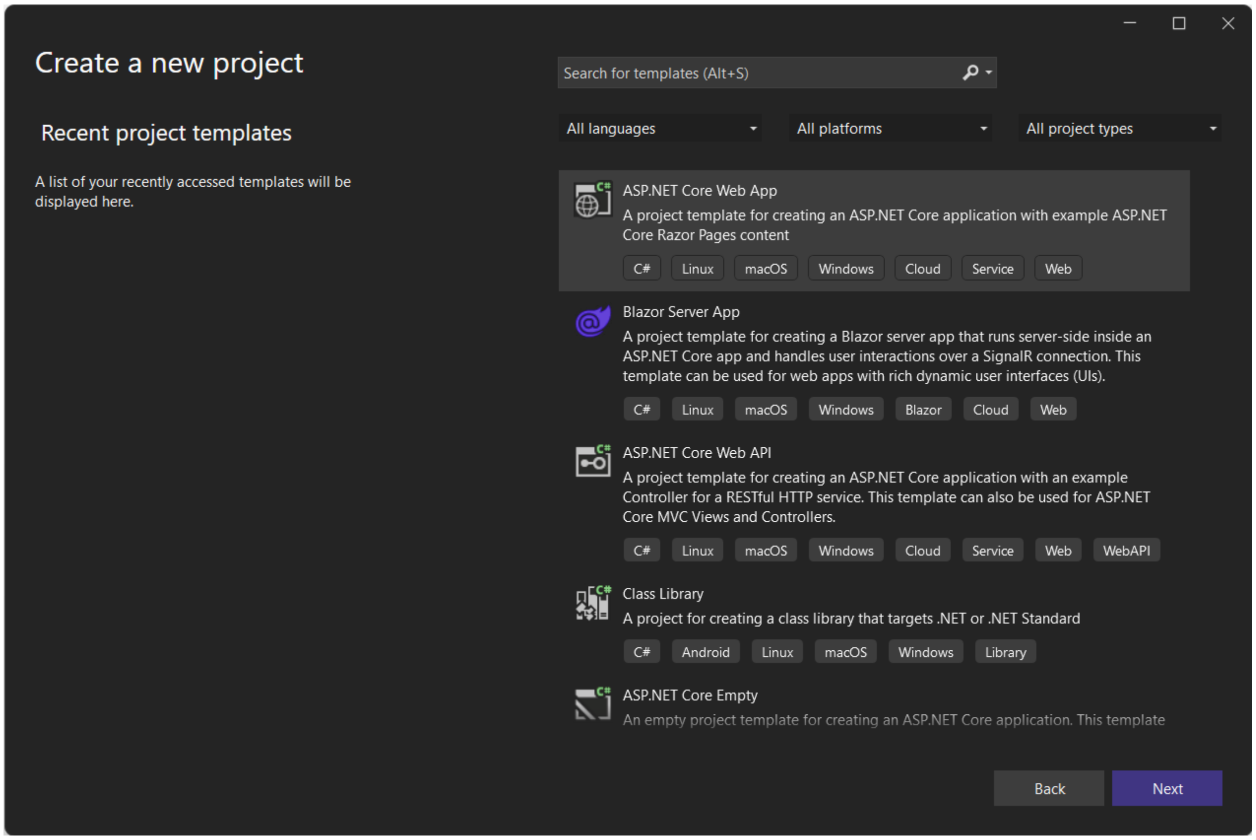
Please take a look at our samples on GitHub:

- Platformus-Sample-Personal-Website
- Platformus-Sample-Personal-Blog
- Platformus-Sample-Ecommerce
- Platformus-Sample-Mobile-App-Admin-Panel

1.1.2 Using as the NuGet Packages

Using Platformus CMS as the NuGet packages is the preferred way (unless you consider the source code only as a starting point for your own project and do not need any updates in the future). Your host web application can contain any application-specific code, and you can extend the built-in features and modify the default behaviors using the Platformus public API.

1. Create an ASP.NET Core host web application (or use an existing one):


Create a new project

Recent project templates

A list of your recently accessed templates will be displayed here.

Search for templates (Alt+S)

All languages | All platforms | All project types

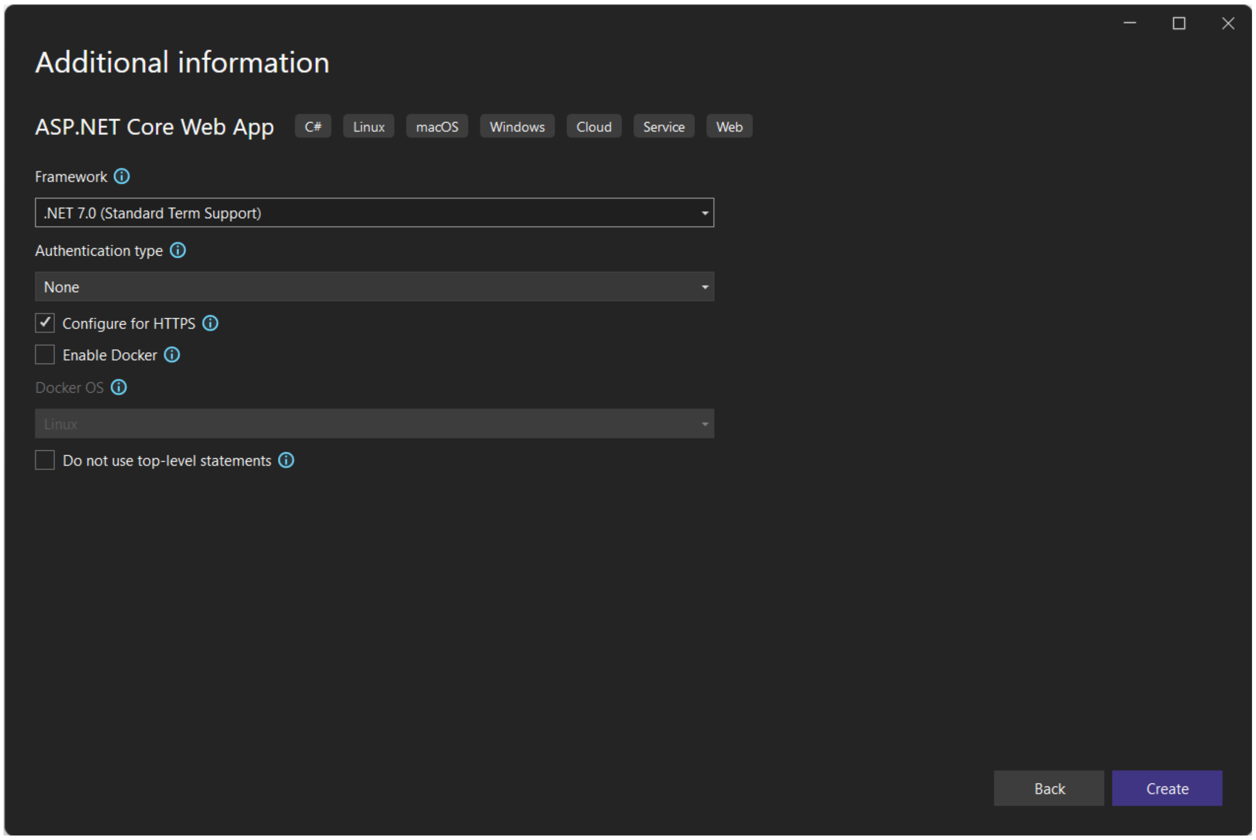
ASP.NET Core Web App
A project template for creating an ASP.NET Core application with example ASP.NET Core Razor Pages content
C# Linux macOS Windows Cloud Service Web

Blazor Server App
A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Blazor Cloud Web

ASP.NET Core Web API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.
C# Linux macOS Windows Cloud Service Web WebAPI

Class Library
A project for creating a class library that targets .NET or .NET Standard
C# Android Linux macOS Windows Library

ASP.NET Core Empty
An empty project template for creating an ASP.NET Core application. This template


Additional information

ASP.NET Core Web App C# Linux macOS Windows Cloud Service Web

Framework: .NET 7.0 (Standard Term Support)

Authentication type: None

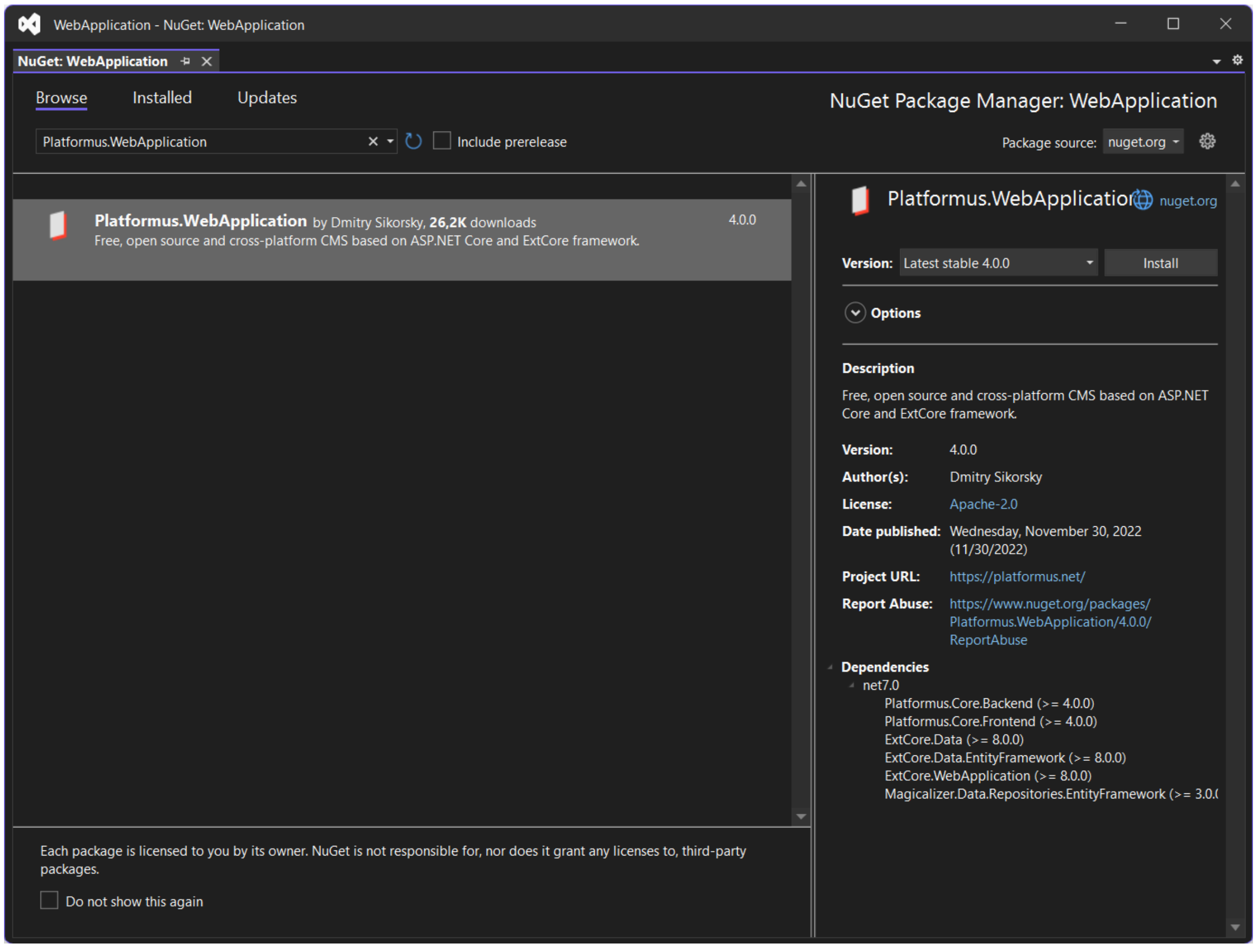
[✓] Configure for HTTPS

[ ] Enable Docker

Docker OS: Linux

[ ] Do not use top-level statements

2. Open NuGet Package Manager and add dependencies on the following Platformus packages:


WebApplication - NuGet: WebApplication

NuGet: WebApplication

Browse Installed Updates

NuGet Package Manager: WebApplication

Platformus.WebApplication

Include prerelease

Package source: nuget.org

Platformus.WebApplication by Dmitry Sikorsky, **26,2K** downloads 4.0.0
Free, open source and cross-platform CMS based on ASP.NET Core and ExtCore framework.

Platformus.WebApplication nuget.org

Version: Latest stable 4.0.0 Install

Options

Description

Free, open source and cross-platform CMS based on ASP.NET Core and ExtCore framework.

Version: 4.0.0
Author(s): Dmitry Sikorsky
License: Apache-2.0
Date published: Wednesday, November 30, 2022 (11/30/2022)
Project URL: https://platformus.net/
Report Abuse: https://www.nuget.org/packages/Platformus.WebApplication/4.0.0/ReportAbuse

Dependencies
net7.0
Platformus.Core.Backend (>= 4.0.0)
Platformus.Core.Frontend (>= 4.0.0)
ExtCore.Data (>= 8.0.0)
ExtCore.Data.EntityFramework (>= 8.0.0)
ExtCore.WebApplication (>= 8.0.0)
Magicalizer.Data.Repositories.EntityFramework (>= 3.0.(

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

Do not show this again

- Platformus.Core.Data.EntityFramework.Sqlite
- Platformus.Images
- Platformus.Website.Backend
- Platformus.Website.Data.EntityFramework.Sqlite
- Platformus.Website.Frontend
- Platformus.WebApplication

Or you can add them manually by editing .csproj file of your web application project:

```
<ItemGroup>
  <PackageReference Include="Platformus.Core.Data.EntityFramework.Sqlite" Version="4.
↪0.0" />
  <PackageReference Include="Platformus.Images" Version="4.0.0" />
  <PackageReference Include="Platformus.Website.Backend" Version="4.0.0" />
  <PackageReference Include="Platformus.Website.Data.EntityFramework.SqlServer"␣
↪Version="4.0.0" />
  <PackageReference Include="Platformus.Website.Frontend" Version="4.0.0" />
  <PackageReference Include="Platformus.WebApplication" Version="4.0.0" />
</ItemGroup>
```

3. Open your `Startup` class.

Add the `services.AddPlatformus()` extension method call inside the `ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddPlatformus();
}
```

Add the `StorageContextOptions` options class configuration inside the `ConfigureServices` method in order to provide the connection string (of course, you should take it from the application settings):

```
public void ConfigureServices(IServiceCollection services)
{
  services.Configure<StorageContextOptions>(options =>
    {
      options.ConnectionString = this.configuration.GetConnectionString("Default");
    }
  );

  services.AddPlatformus(this.extensionsPath);
}
```

Add the `applicationBuilder.UsePlatformus()` extension method call inside the `Configure` method:

```
public void Configure(IApplicationBuilder applicationBuilder, IWebHostEnvironment␣
↪webHostEnvironment)
{
  if (webHostEnvironment.IsDevelopment())
    applicationBuilder.UseDeveloperExceptionPage();

  applicationBuilder.UsePlatformus();
}
```

Don’t forget to include the `Platformus.WebApplication.Extensions` namespace in order these extension methods to be resolved.

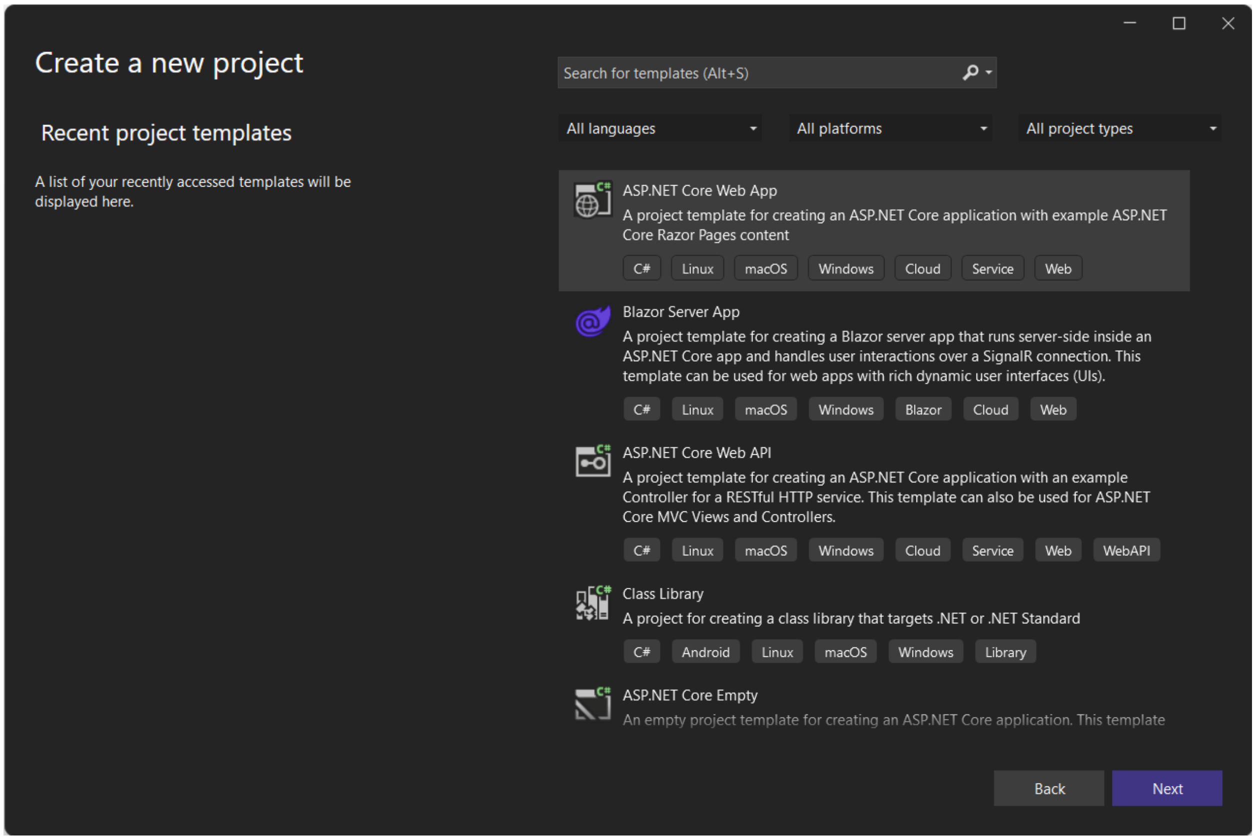
4. Execute the Platformus *database scripts* on your database.

5. Run your web application and navigate to /backend to configure Platformus. Use the default “admin@platformus.net” and “admin” credentials to sign in.

1.1.3 Using as the Source Code

Use Platformus CMS as the source code only if you consider it as a starting point for your own project and do not need any updates in the future.

1. Create an ASP.NET Core host web application (or use an existing one):


Create a new project

Recent project templates

A list of your recently accessed templates will be displayed here.

Search for templates (Alt+S)

All languages | All platforms | All project types

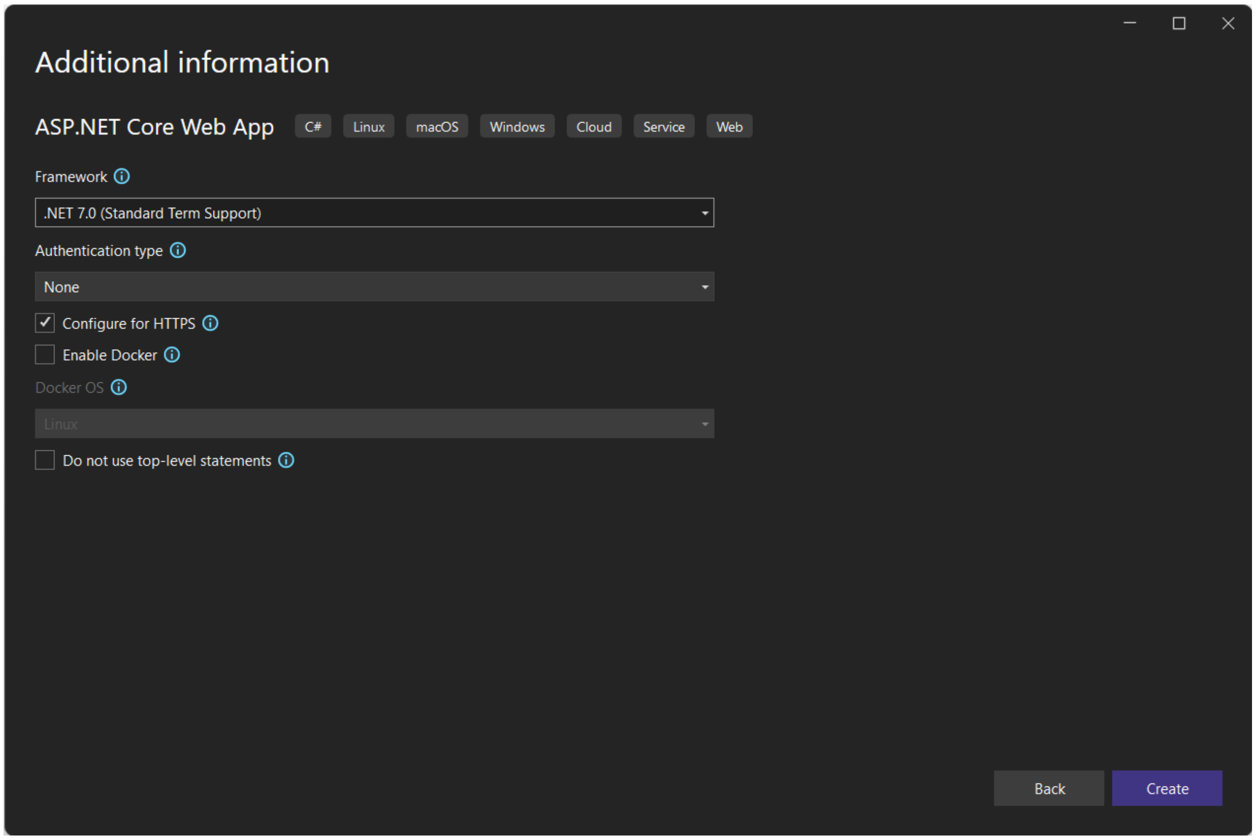
ASP.NET Core Web App
A project template for creating an ASP.NET Core application with example ASP.NET Core Razor Pages content
C# Linux macOS Windows Cloud Service Web

Blazor Server App
A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Blazor Cloud Web

ASP.NET Core Web API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.
C# Linux macOS Windows Cloud Service Web WebAPI

Class Library
A project for creating a class library that targets .NET or .NET Standard
C# Android Linux macOS Windows Library

ASP.NET Core Empty
An empty project template for creating an ASP.NET Core application. This template


Additional information

ASP.NET Core Web App C# Linux macOS Windows Cloud Service Web

Framework: .NET 7.0 (Standard Term Support)

Authentication type: None

[✓] Configure for HTTPS

[ ] Enable Docker

Docker OS: Linux

[ ] Do not use top-level statements

2. Download Platformus sources from the GitHub. Copy them into your solution folder.

3. Add dependencies on the following projects to your web application project:

- Platformus.Core.Data.EntityFramework.Sqlite
- Platformus.Images
- Platformus.Website.Backend
- Platformus.Website.Data.EntityFramework.Sqlite
- Platformus.Website.Frontend
- Platformus.WebApplication

4. Open your `Startup` class.

Add the `services.AddPlatformus()` extension method call inside the `ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddPlatformus();
}
```

Add the `StorageContextOptions` options class configuration inside the `ConfigureServices` method in order to provide the connection string (of course, you should take it from the application settings):

```
public void ConfigureServices(IServiceCollection services)
{
  services.Configure<StorageContextOptions>(options =>
    {
      options.ConnectionString = this.configuration.GetConnectionString("Default");
    }
  );

  services.AddPlatformus(this.extensionsPath);
}
```

Add the `applicationBuilder.UsePlatformus()` extension method call inside the `Configure` method:

```
public void Configure(IApplicationBuilder applicationBuilder, IWebHostEnvironment␣
↪webHostEnvironment)
{
  if (webHostEnvironment.IsDevelopment())
    applicationBuilder.UseDeveloperExceptionPage();

  applicationBuilder.UsePlatformus();
}
```

Don't forget to include the `Platformus.WebApplication.Extensions` namespace in order these extension methods to be resolved.

4. Execute the Platformus *database scripts* on your database.

5. Run your web application and navigate to /backend to configure Platformus. Use the default "admin@platformus.net" and "admin" credentials to sign in.

1.1.4 Database Scripts

Currently Platformus CMS supports following database types:

- PostgreSQL database

- SQLite database
- SQL Server database

GitHub repository contains SQL scripts for each database type, separately with the schema and initial data.

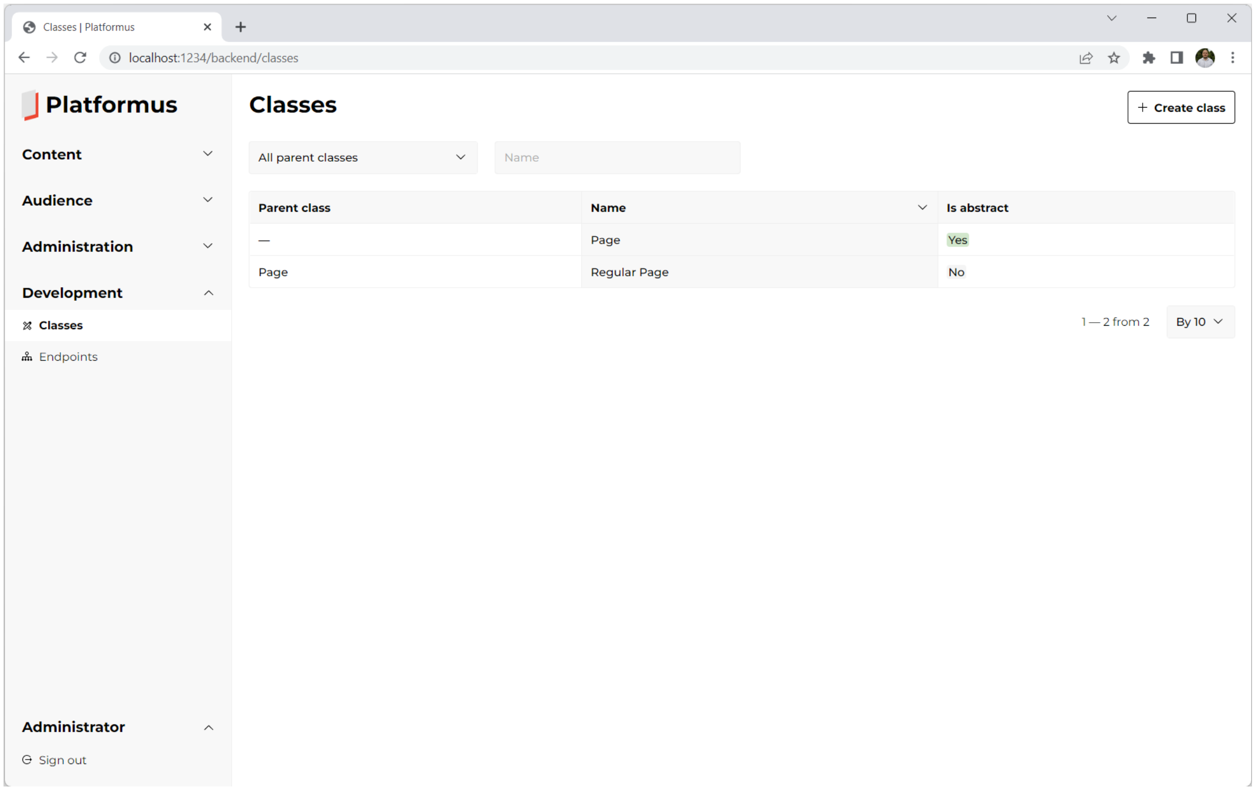
1.1.5 Tutorial: Basic Content Management

To understand better how the Platformus content management works, let's use the default personal website sample and add blog feature to it.

There are two ways to work with content using Platformus CMS: using the built-in Website extension and writing a custom one. The first option is suitable when simplicity and speed of development are more important than application performance. It works good for website pages, blog categories, posts, tags etc. The Website extension implements the EAV pattern using the classes and members to describe content, and objects, properties, and relations to store it. Writing custom extension is more complex task, but in this case performance and flexibility are maximized. In this tutorial we will go the first way.

First of all, we need the blog post pages. Each post page should have the same properties as the regular page, but also it needs preview (a small piece of content), image, and creation date. In the Platformus context, all the pages are objects which are described by the classes. Therefore, to create new type of page (and new type of object) we need to create the corresponding class first.

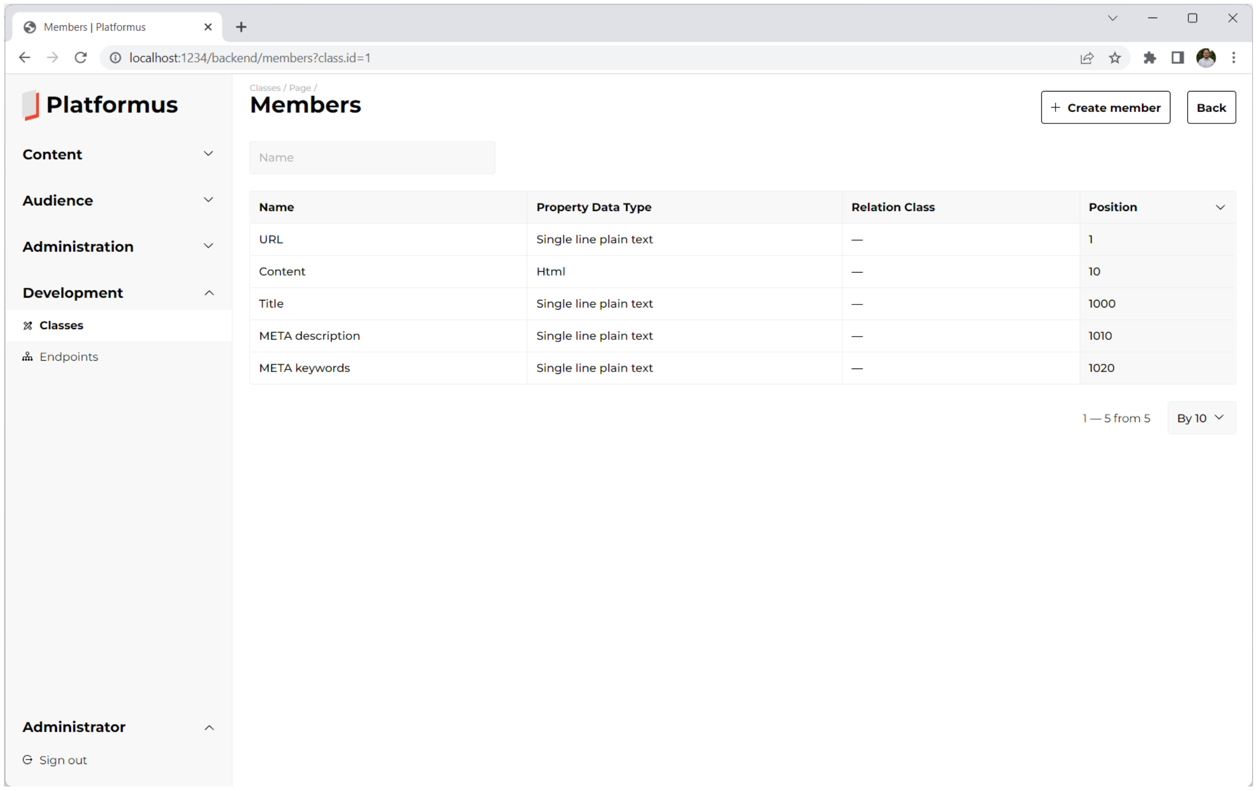
Go to the backend and sign in (navigate to https://localhost:5000/backend/) and then go to the *Development/Classes* section. There are already two classes here: `Page` and `Regular Page`.



Classes

Parent class	Name	Is abstract
—	Page	Yes
Page	Regular Page	No

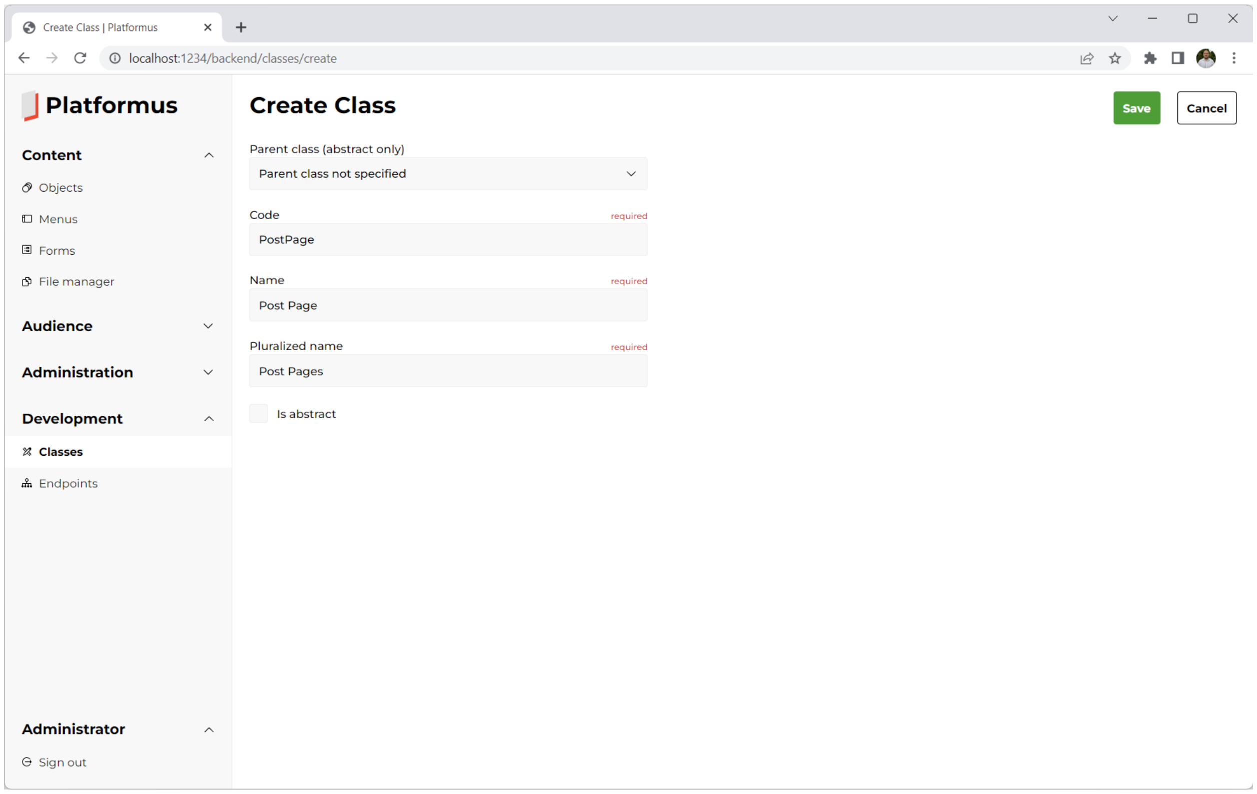
The `Page` class is abstract, it means that it is used as the base class for the other ones (class copies all the members of its parent class). Click the *Members* link of the `Page` class to see the list of its members. As you can see, there are the standard page properties here, like `URL`, `Content`, `Title`, `META description`, and `META keywords`:



Name	Property Data Type	Relation Class	Position
URL	Single line plain text	—	1
Content	Html	—	10
Title	Single line plain text	—	1000
META description	Single line plain text	—	1010
META keywords	Single line plain text	—	1020

1 — 5 from 5 By 10

Now return to the class list and click the *Create class* button. Select the `Page` class as the parent class for our new one. Fill the *Code*, *Name*, and *Pluralized name* fields as shown below:



Parent class (abstract only): Parent class not specified

Code (required): PostPage

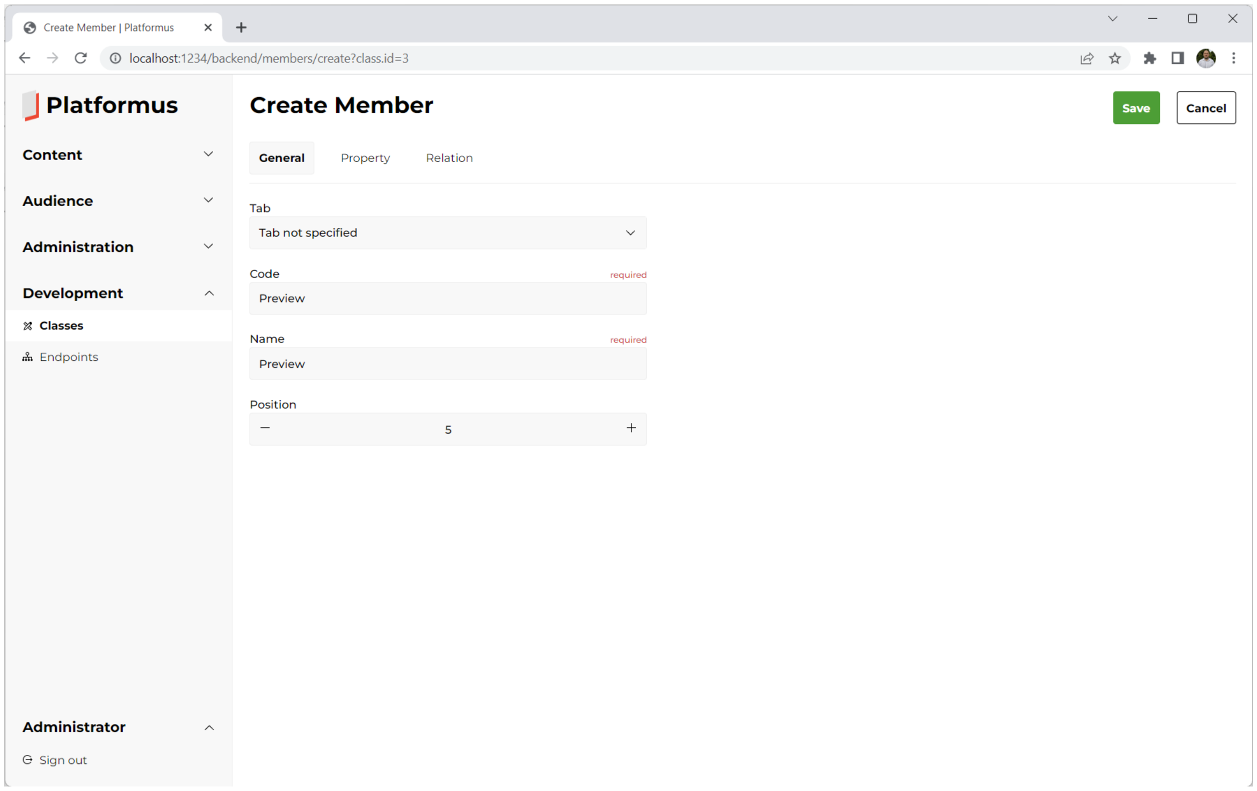
Name (required): Post Page

Pluralized name (required): Post Pages

Is abstract

Click the *Save* button. New class is created. Now go to the list of its members. It is empty for now (but don't forget that our class will have all the members from the `Page` class, because it is selected as the parent one).

Let's add the `Preview` member to our class. Click the *Create member* button and fill the *Code*, *Name*, and *Position* fields as shown below:



Create Member

Tab: Tab not specified

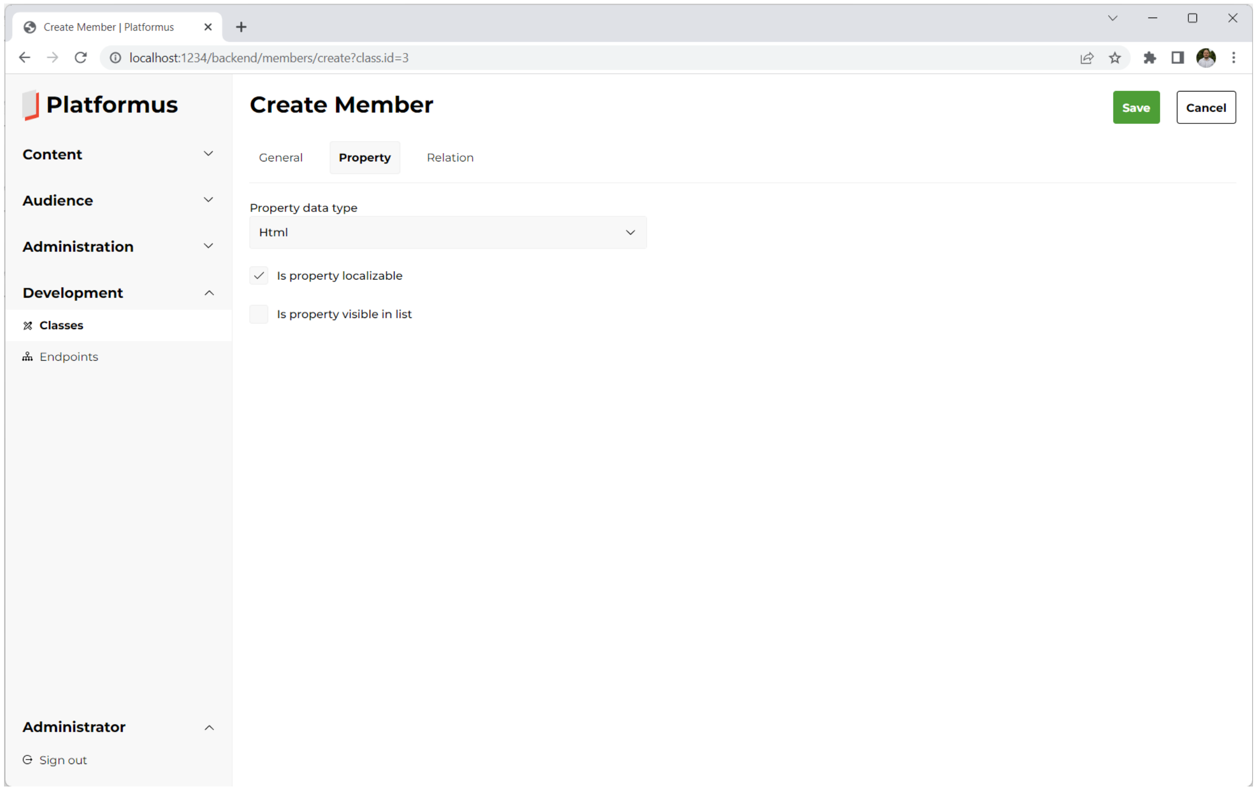
Code (required): Preview

Name (required): Preview

Position: 5

If we had a lot of the members, we could use tabs to group them on the object edit page, but in our case, we only use it to group the properties that are related to SEO in the parent class. Position is set to 5, because we want our property editor to appear before the `Content` property one (`Content` member has position set to 10).

Now click the *Property* tab and fill the fields as shown below:



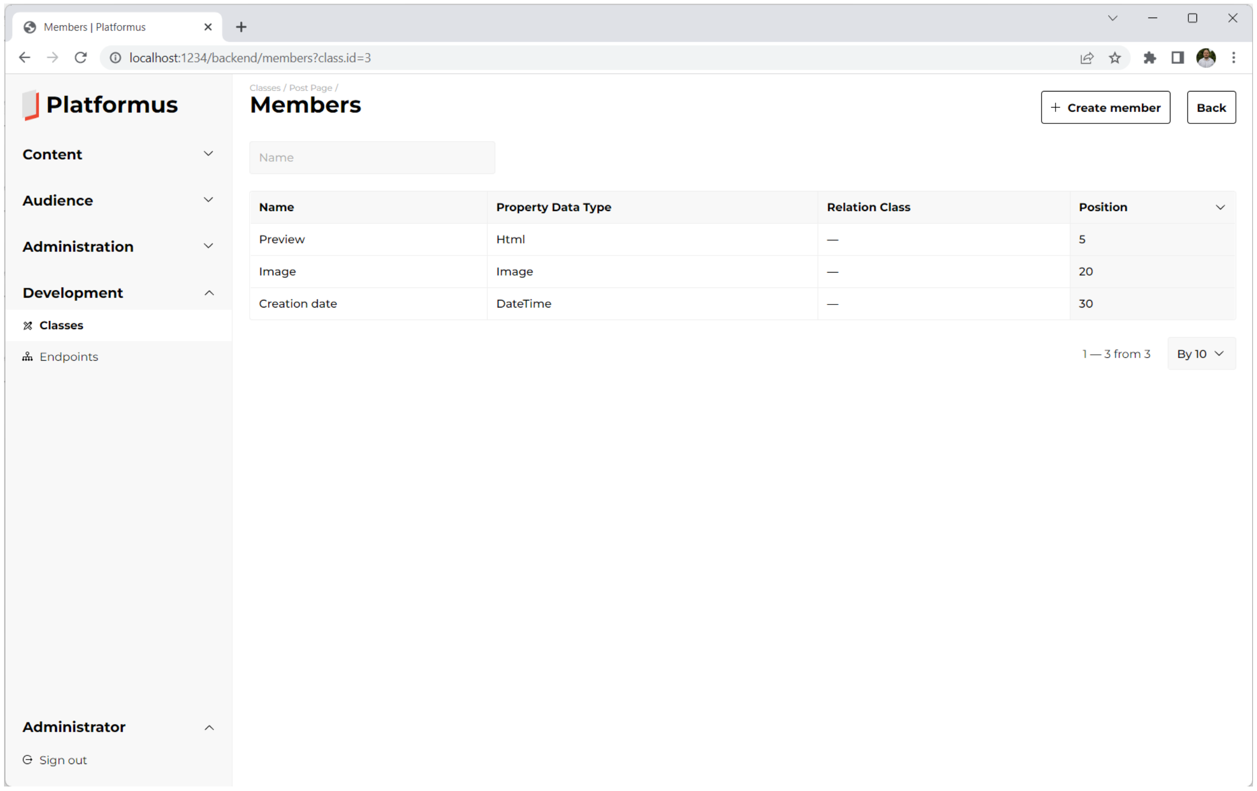
Property data type: Html

✓ Is property localizable

Is property visible in list

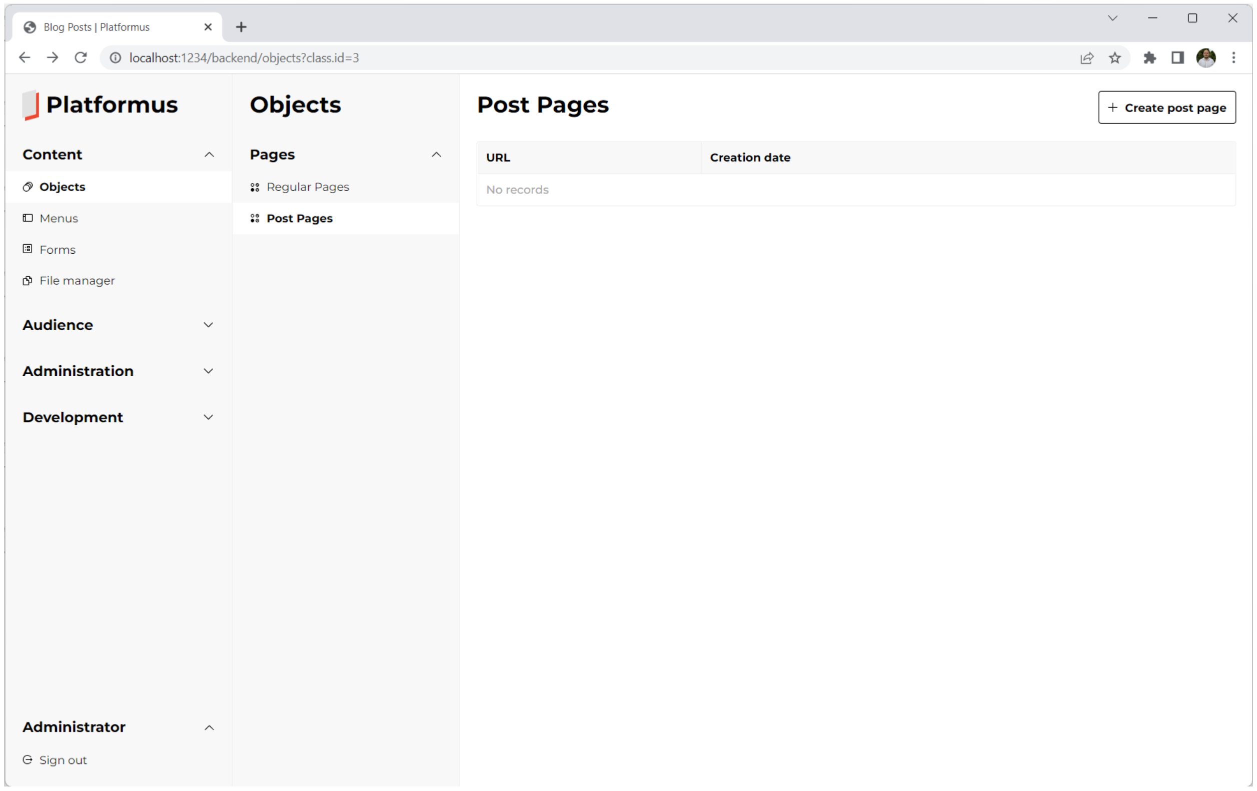
When you change the property data type, the set of the fields on this tab is changed too. You can add your own data types and specify their properties (as well as the client-side editors that are used to edit them). For the properties that have short values we can set the *Is property visible in list* checkbox, so that properties will be displayed in the object list (we will see that later). Now click the *Save* button again, our member is created.

Add the `Image` and `Creation date` members in the same way (but select the `Image` and `Date` property data types for them). Our member list will look like this:



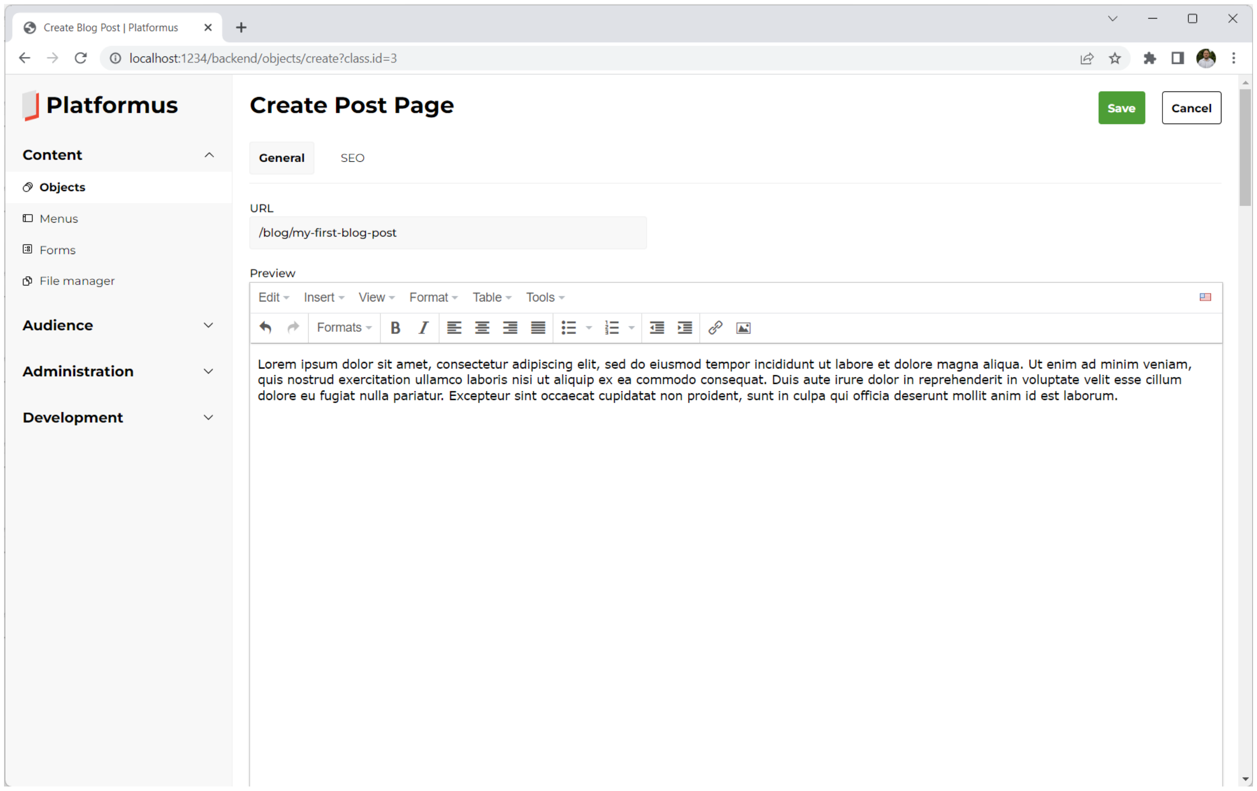
Name	Property Data Type	Relation Class	Position
Preview	Html	—	5
Image	Image	—	20
Creation date	DateTime	—	30

That's it, we are done with our data model for now. Let's add some content. Go to the *Content/Objects* section. Objects (and again, our pages are objects) are grouped by the parent classes (pluralized names are used to name the groups). Objects of the classes that doesn't have parent ones go under the Others group. Our `Post Page` class is already here:



URL	Creation date
No records	

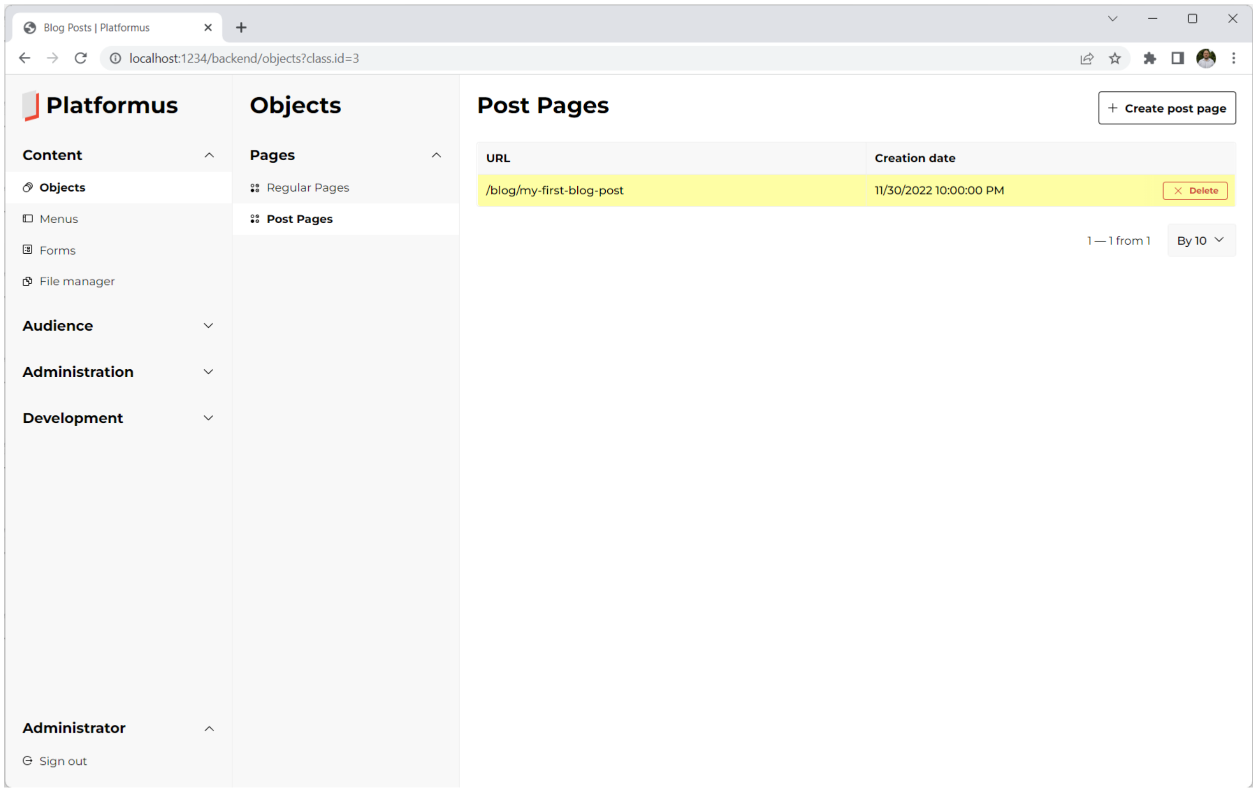
Click the *Create post page* button:



URL
/blog/my-first-blog-post

Preview
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

As you can see, all the properties we have defined in the corresponding class are here. Fill the fields and click the *Save* button. New post is created:



URL	Creation date
/blog/my-first-blog-post	11/30/2022 10:00:00 PM

There are only the properties are displayed whose members have *Is property visible in list* checkbox checked.

Now we have our post page object created. We can use different ways to present it (view, API, plain text and so on), but now let's use old good view for that.

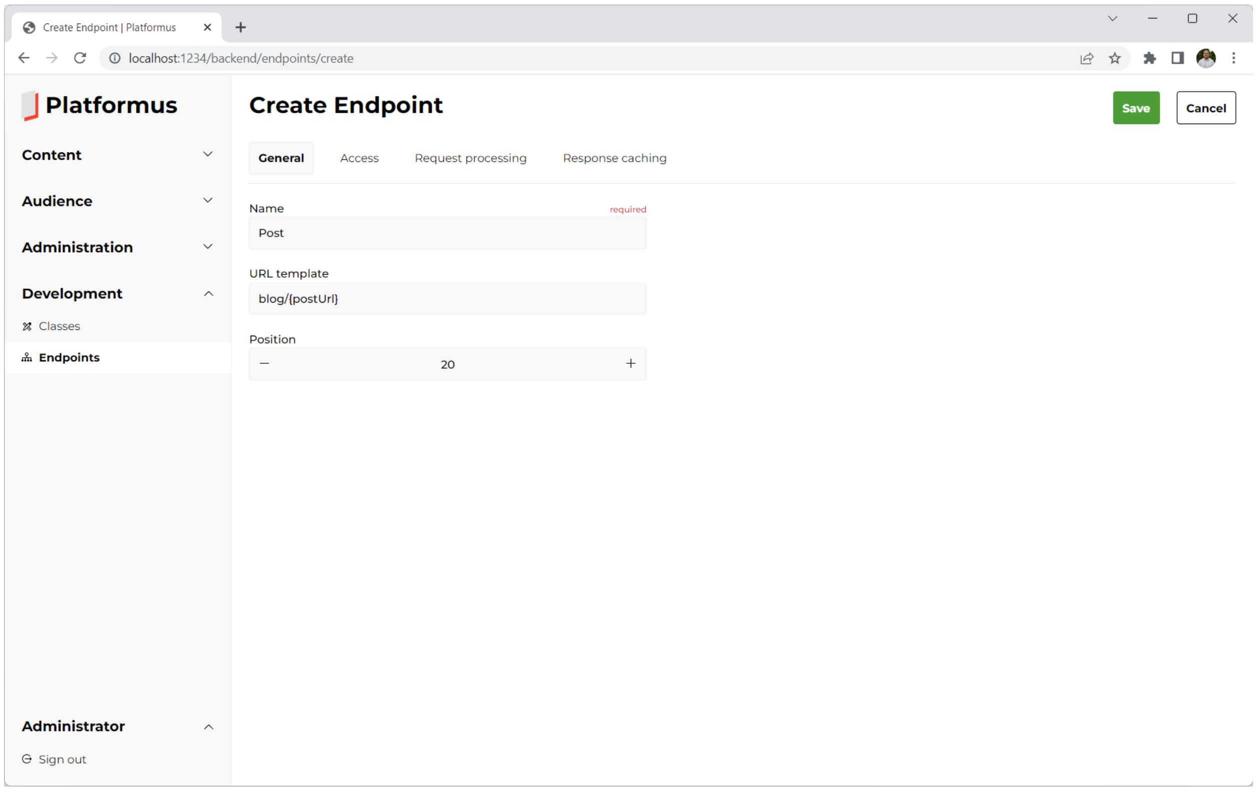
Create PostPage.cshtml file inside the Views folder of the web application project with the following content:

```
@model dynamic
@{
  this.ViewBag.Title = this.Model.Page.Title;
  this.ViewBag.MetaKeywords = this.Model.Page.MetaKeywords;
  this.ViewBag.MetaDescription = this.Model.Page.MetaDescription;
}
<div class="post-page post">
  <h1>
    @Model.Page.Title
  </h1>
  <div class="post__cover">
    <img class="post__cover-image" src="@Model.Page.Image" alt="@Model.Page.Title" />
  </div>
  @Html.Raw(this.Model.Page.Content)
  <div class="post__created">
    @Model.Page.CreationDate
  </div>
</div>
```

The HTML ifself is very simple. You can see that all the data comes from the view model. There is the `Page` property which contains all the properties of our post page object that we have described by the class members (and property names are the same as the member codes). This `Page` property is created for us by the corresponding data source. If your view needs more different data in order to be rendered, just add more data sources that will provide this data to the view model.

Data sources specify the C# classes that implement the IDataProvider interface, you can *create your own ones*. They can provide data in any way you need: to load some objects, to take it from the web services (weather forecast?), or to return some hardcoded values. All the data sources that are used to process the particular request are grouped inside the endpoint. Endpoints process the requests and return response in Platformus-based web applications (as well as data sources, they specify C# classes that implement the IRequestProcessor interface, and you can create your own implementations). We will see how this all works a bit later in this article. You can still use regular C# controllers to process requests which is simple, but it is related to writing custom extensions and is not considered in this tutorial.

We have described and created the content (our post page object), we have also created the presentation for that content (our view). The last thing we must do to make it all work is to create the endpoint and the data source. Go to the *Development/Endpoints* section. Click the *Create endpoint* button and fill the fields as shown below:

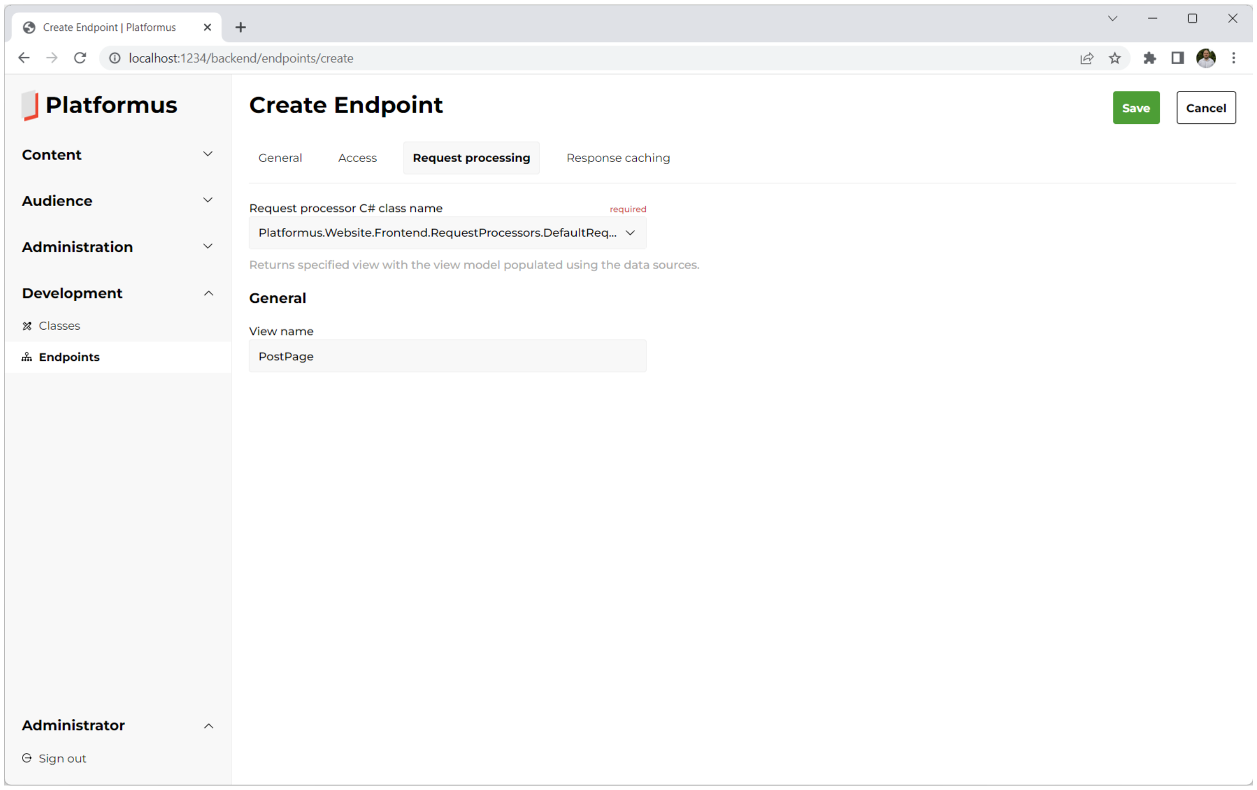


Create Endpoint

Name required: Post

URL template: blog/{postUrl}

Position: 20



Create Endpoint

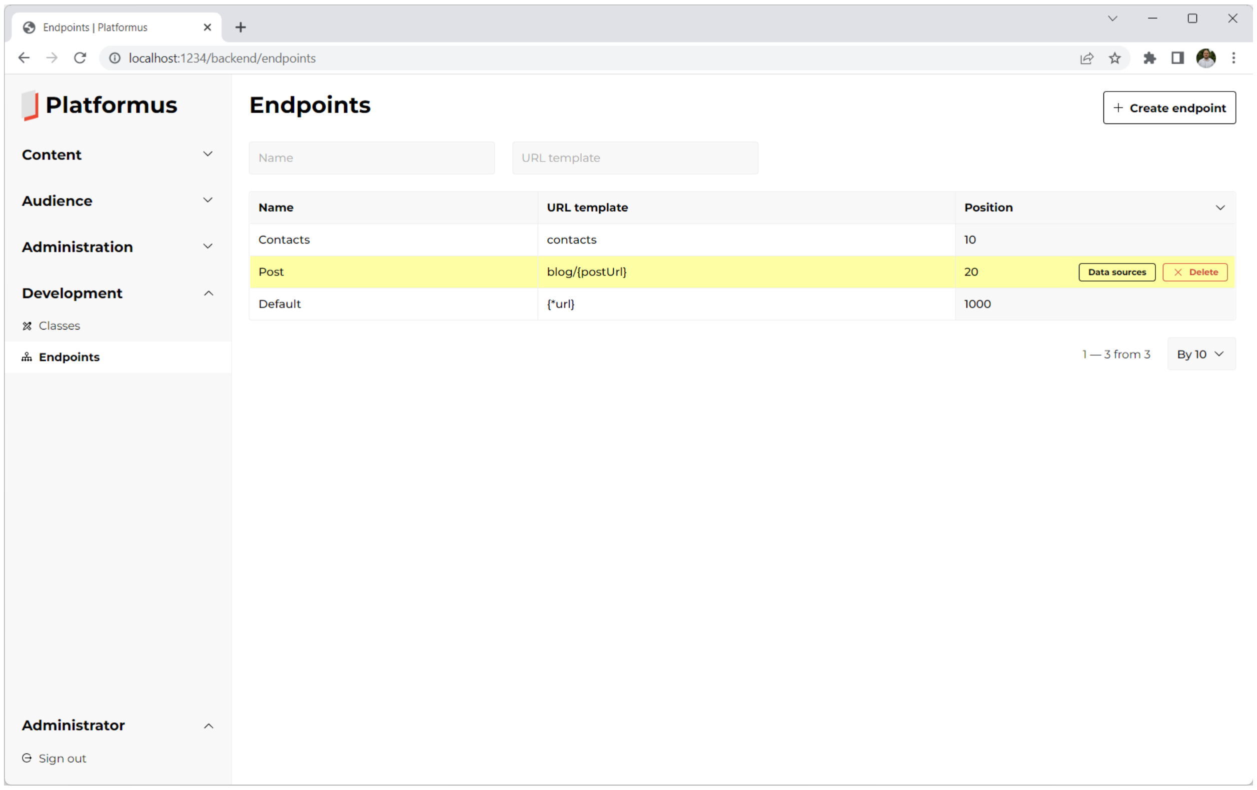
Request processor C# class name required: Platformus.Website.Frontend.RequestProcessors.DefaultReq...

Returns specified view with the view model populated using the data sources.

General

View name: PostPage

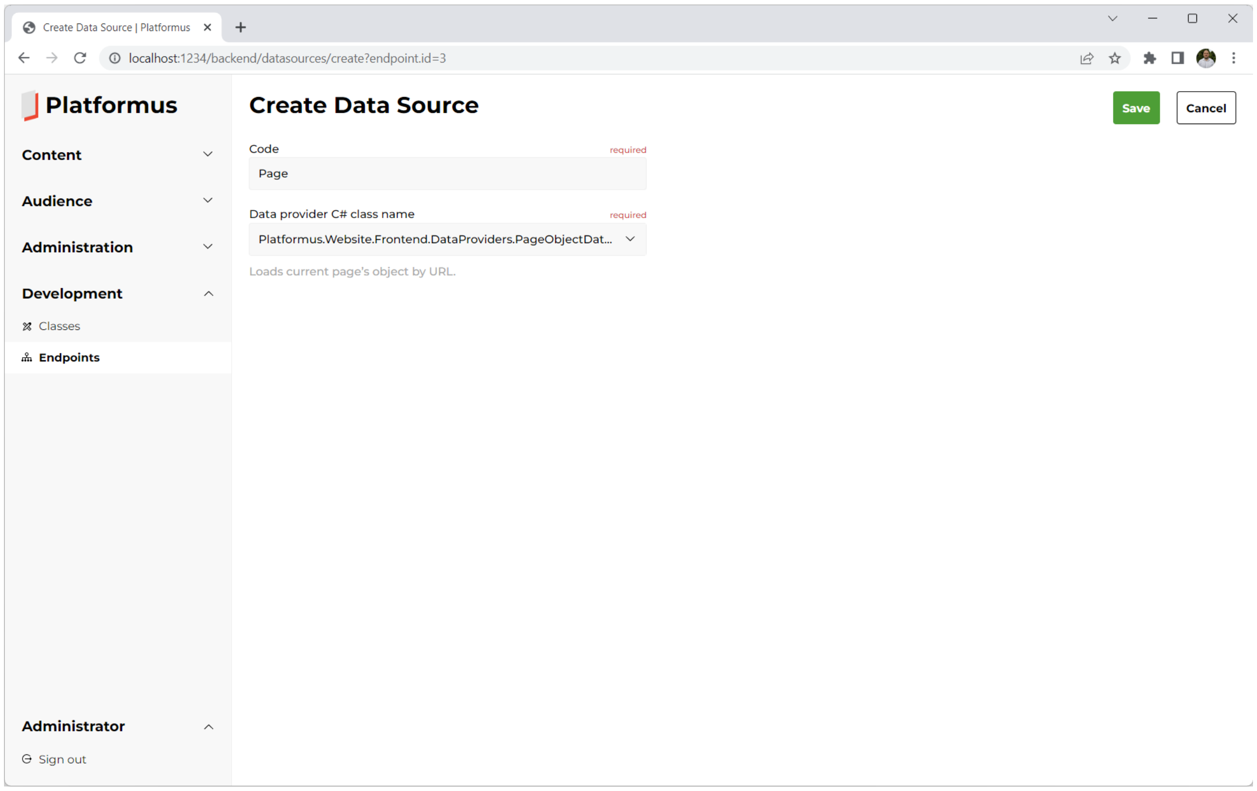
Endpoints define how your Platformus-based web application processes the HTTP requests. By default, if there are no endpoints (and regular C# controllers or pages) configured, you will have 404 response on every request. By specifying the URL template for the endpoint, you tell the instance of the IEndpointResolver interface which endpoint it should use to process the particular request (you can use {*url} one to handle all the requests). It is done the similar way as the MVC routes configuration (endpoint is something like route and controller at once; endpoints support URL parameters too) and it is executed after the default MVC routing. Also, you can specify which C# class (implementation of the `IRequestProcessor` interface) will process the request and return the result. You can write your own implementations of that interface and use them to process the requests. Specify the view name that we have created earlier that will be used by this endpoint to render the response. Click the *Save* button to create our new endpoint:



Name	URL template	Position
Contacts	contacts	10
Post	blog/{postUrl}	20
Default	{*url}	1000

One more thing about the endpoints. Default implementation of the `IEndpointResolver` interface checks endpoints, sorted by the position, one by one (whether the current one's URL template matches the request's URL or not). That's why position field value is important. If you have a few endpoints that match the given URL, the first one will be used.

The last thing we have to do is to add the data source that will load the post page object by the value of the `URL` property and assign it to the view model's `Page` property (that will also be created). Click the *Data sources* link and then the *Create data source* button. Fill all the fields as shown below and click the *Save* button:



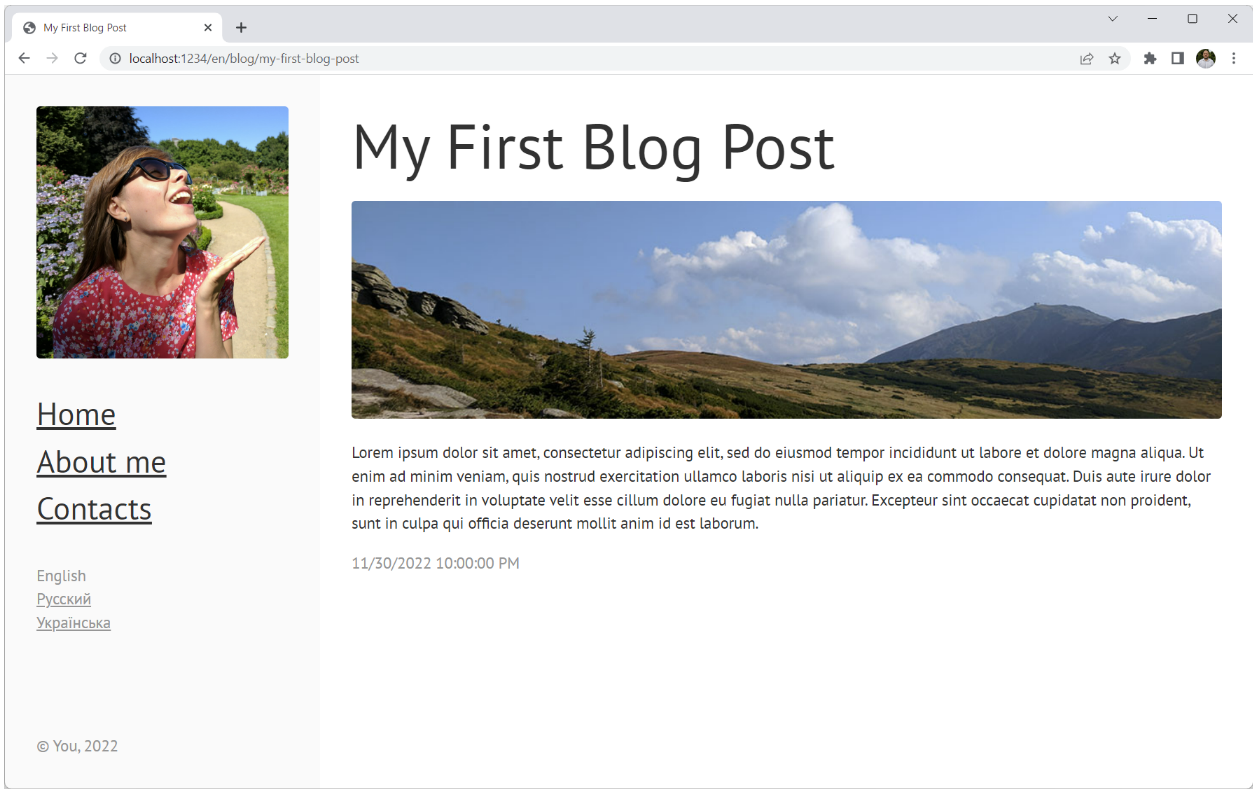
Create Data Source

Code *required*
Page

Data provider C# class name *required*
Platformus.Website.Frontend.DataProviders.PageObjectDat...

Loads current page's object by URL.

That's it. Now we can test how our post page is displayed. Navigate to https://localhost:5000/en/blog/my-first-blog-post:



My First Blog Post

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

11/30/2022 10:00:00 PM

It works! But we also need to have a page with all the posts. We will make it quickly, because now you know enough.

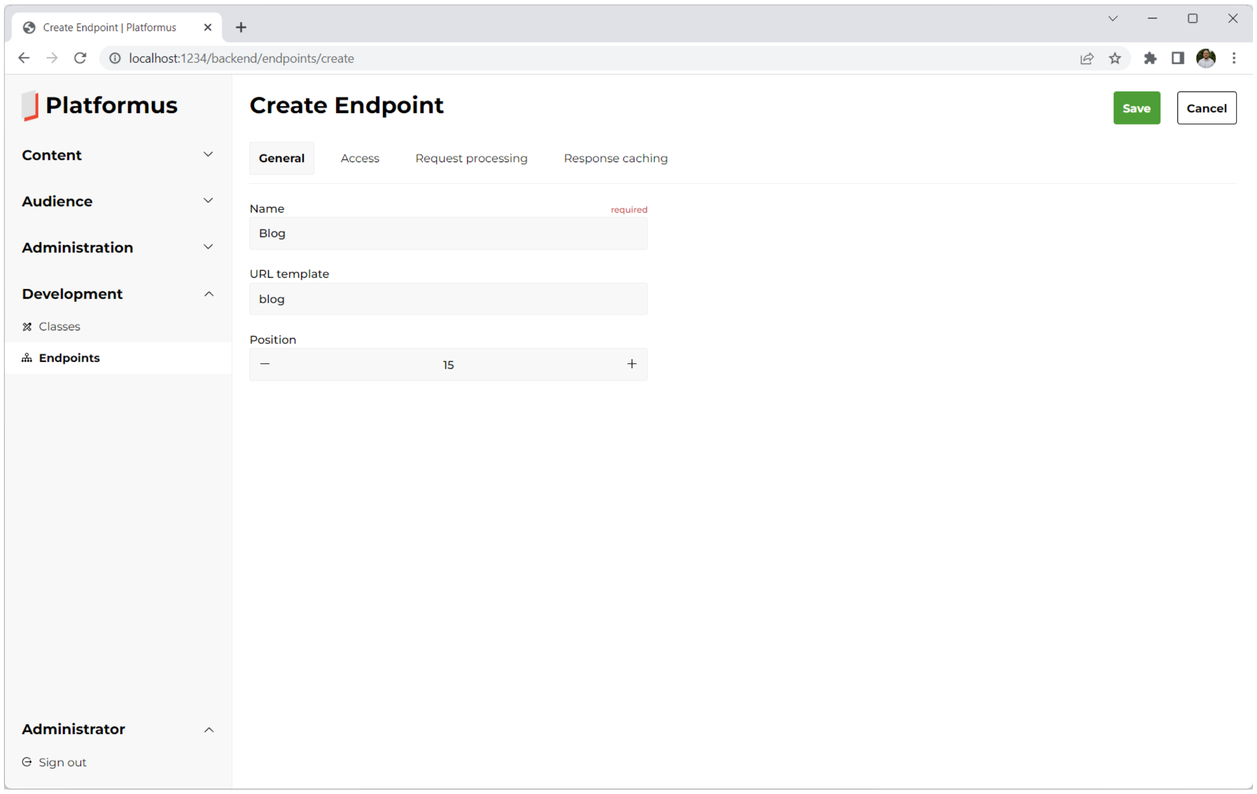
This page will display the posts, so we don't need to create any new class (just create the regular page object with the `URL` property value set to /blog). All we need is to create new view, endpoint and two data sources for it. Let's start from the BlogPage.cshtml view:

```
@model dynamic
@{
  this.ViewBag.Title = this.Model.Page.Title;
  this.ViewBag.MetaKeywords = this.Model.Page.MetaKeywords;
  this.ViewBag.MetaDescription = this.Model.Page.MetaDescription;
}
@Html.Raw(this.Model.Page.Content)
<div class="blog">
  @foreach (var post in this.Model.Posts)
  {
    @Html.Partial("_Post", post as object)
  }
</div>
```

As you can see, we will have a data source that will provide the `Posts` view model property for us. Also we have to create the _Post.cshtml partial view (inside the Shared folder):

```
@model dynamic
<div class="posts__post post">
  <h2>
    <a href="/@System.Globalization.CultureInfo.CurrentUICulture.
↪TwoLetterISOLanguageName@Model.Url">@Model.Title</a>
  </h2>
  <div class="post__cover">
    <a href="/@System.Globalization.CultureInfo.CurrentUICulture.
↪TwoLetterISOLanguageName@Model.Url">
      <img class="post__cover-image" src="@Model.Image" alt="@Model.Title" />
    </a>
  </div>
  @Html.Raw(this.Model.Preview)
  <div class="post__created">
    @Model.CreationDate
  </div>
</div>
```

Now create the new endpoint (you have to have the separated endpoint for each page template (or view)):

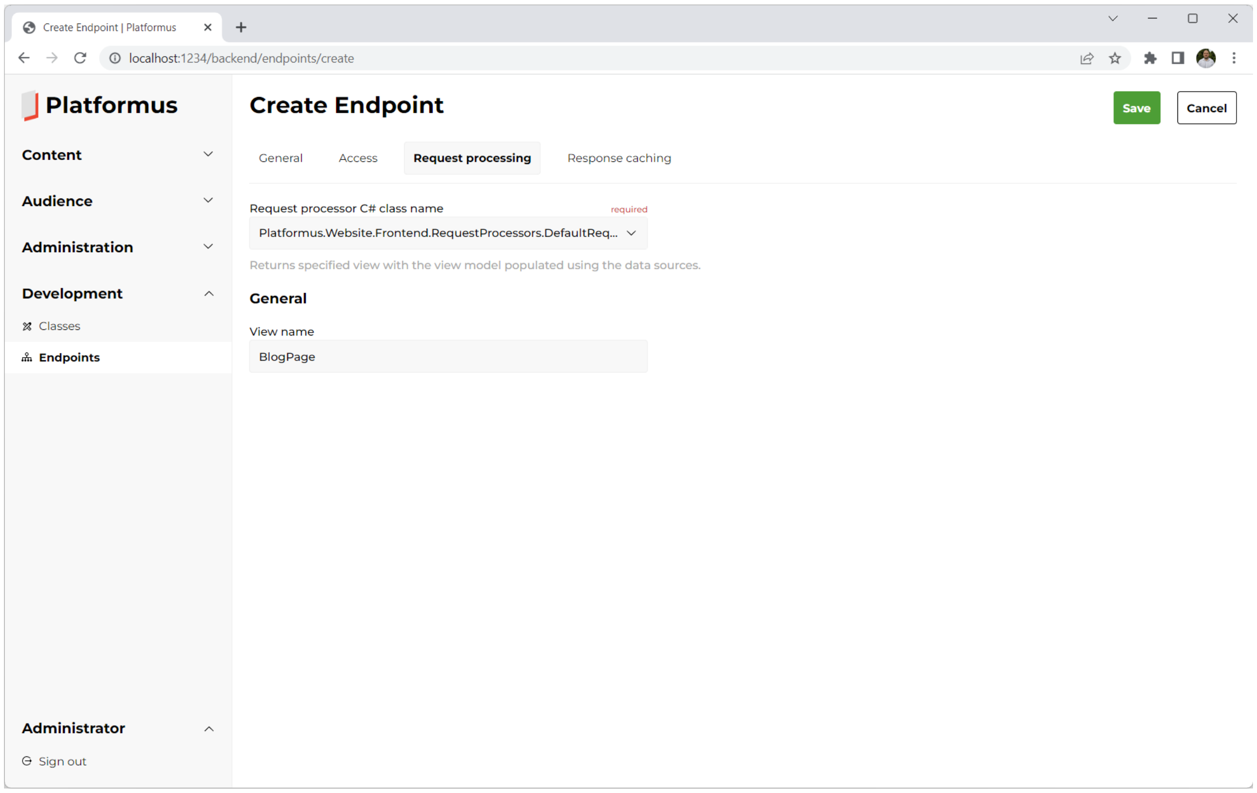


Create Endpoint

Name required: Blog

URL template: blog

Position: 15



Create Endpoint

Request processor C# class name required: Platformus.Website.Frontend.RequestProcessors.DefaultReq...

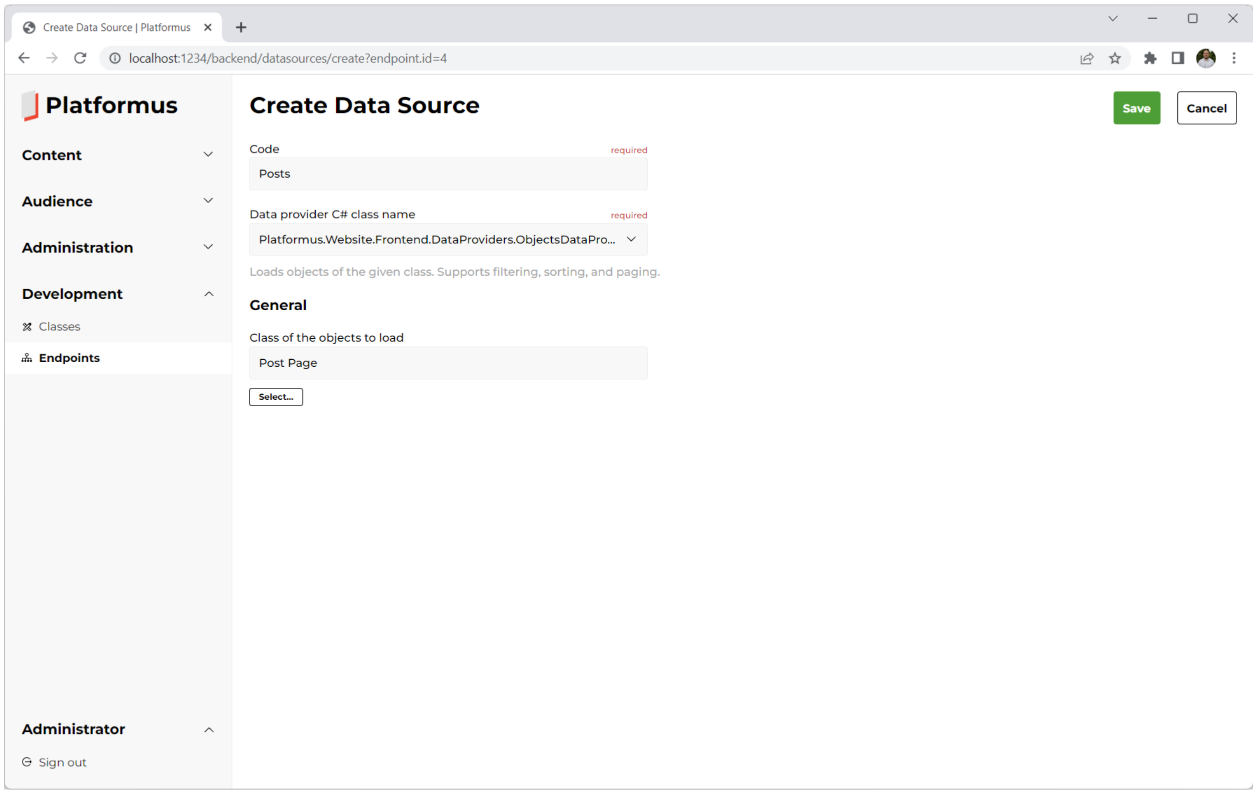
Returns specified view with the view model populated using the data sources.

General

View name: BlogPage

Because the page that will display the list of the posts is the page too, add the Page data source for our new endpoint (the same way we have done that for the previous one). It will load our regular page object that holds `Content` and other properties of this page.

But in order to be able to display the posts on this page, we must add one more data source:



Create Data Source

Code (required): Posts

Data provider C# class name (required): Platformus.Website.Frontend.DataProviders.ObjectsDataPro...

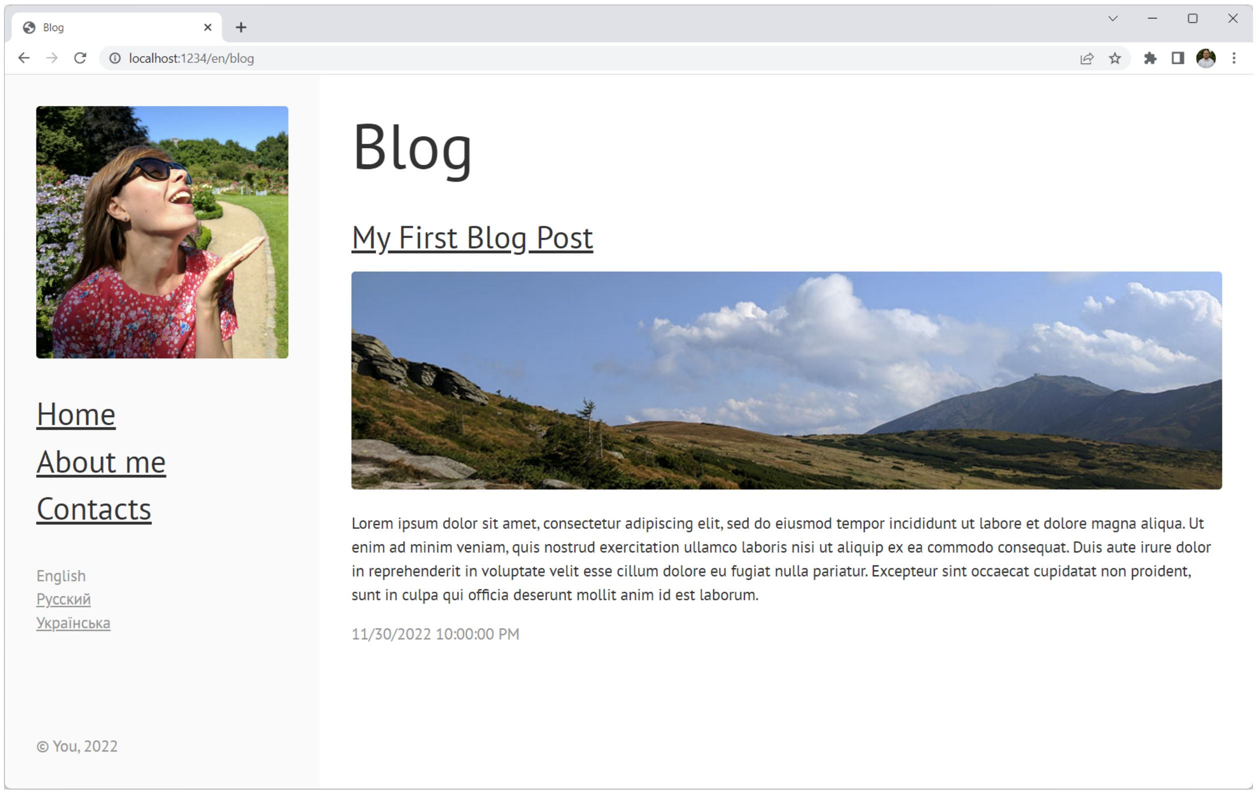
Loads objects of the given class. Supports filtering, sorting, and paging.

General

Class of the objects to load: Post Page

As you can see, another C# class is selected as data provider for this data source. It provides more properties for us. For example, it allows to specify the class of the objects to load etc.

Everything is done. Now you can navigate to https://localhost:5000/en/blog and see the result:



Blog

My First Blog Post

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

11/30/2022 10:00:00 PM

Click the image to go to the post page. You can add the new menu item in the menu to have your blog there.

1.2 Fundamentals

Platformus CMS is modular and extendable, so its behavior is very dependent on the set of the extensions. The main and the only required extension is the *Platformus.Core*. It doesn't provide any content management, instead it implements such basic things like reusable admin panel (or backend) UI, user access control, configuration, localization etc.

Depending on your needs and requirements you can either use other existing extensions to manage your content or write your own one(s). Anyway, the main purpose of the Platformus CMS is to save you from writing routine code, to speed up development, and to provide you with convenient and standardized admin panel UI. Also, you can write and then reuse small typical extensions from project to project.

1.2.1 Standard Extensions

There are 4 standard extensions:

- *Platformus.Core*
- *Platformus.Website*
- *Platformus.Ecommerce*
- *Platformus.Images*

Let's look at a few usage examples to choose the best way to go. All these approaches do not exclude each other and can be combined.

Mobile app API with admin panel

You can create an empty web application, add dependencies on the *Platformus.Core* extension packages, and write a *custom extension* that will contain all the application-specific things: entities, models, DTOs, controllers, admin panel sections etc.

In this case you have the very basic things out of the box and do not need to think about backend UI, combining it using the built-in tag helpers like bricks. At the same time, CMS has no effect on performance, so you can get maximum of what the .NET gives.

Website

Most of the websites' content is changed from time to time, so if you are using a cache, it could be not so important how many milliseconds it takes to retrieve from a database and display your data.

Development time is much more important in such cases, so you can use the *Platformus.Website* extension. It provides features to describe your content with classes and then create and use it as objects. This extension allows to avoid programming and, in many cases, trivial configuration and writing Razor views could be enough.

Ecommerce

Platformus CMS contains the *Platformus.Ecommerce* extension which contains standard ecommerce features such categories and products, filter, cart etc. It can be used as the NuGet packages, or as the source code. The second option could be useful when you need a very specific ecommerce app, so you can just use the source code as a simple barebone and implement features you need.

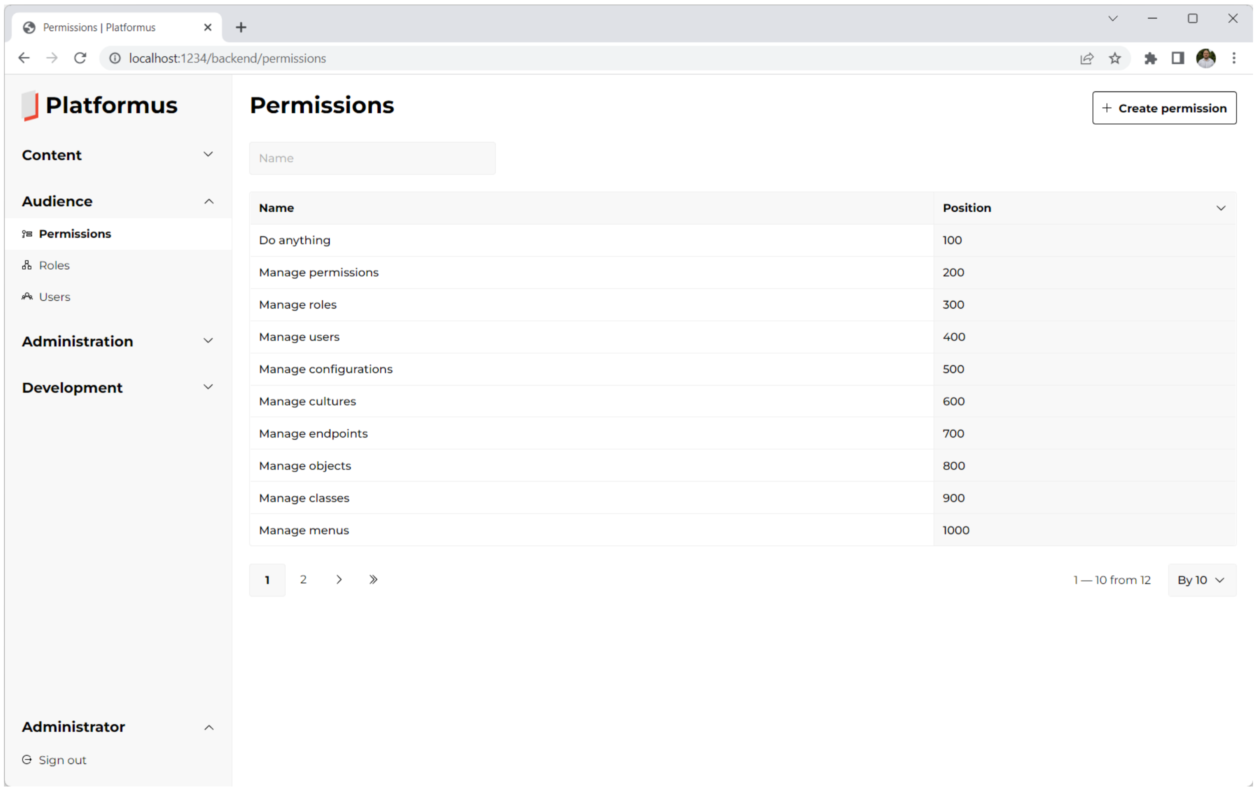
1.3 Platformus.Core

This is the only required extension for any Platformus-based web application. It contains such basic things like reusable admin panel (or backend) UI, user access control based on permissions and roles, configuration, localization etc.

1.3.1 Admin Panel Sections

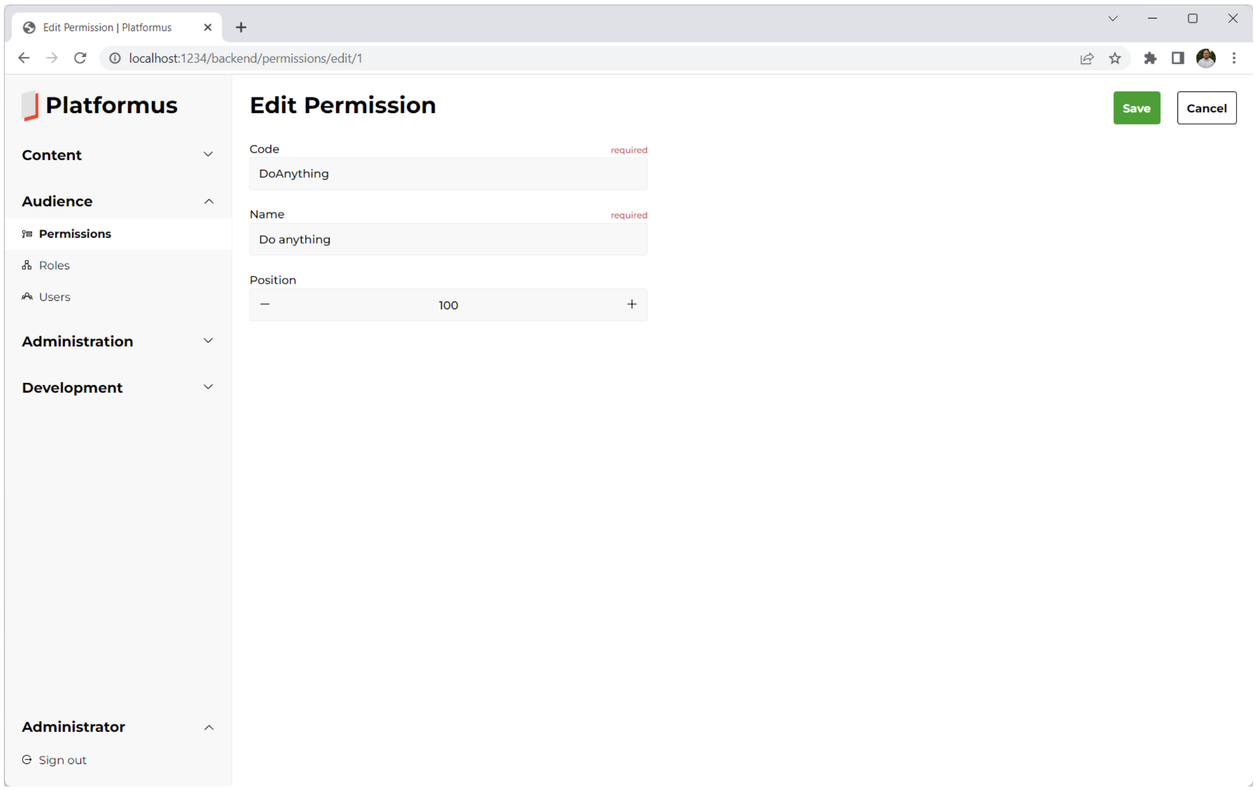
Permissions

Permissions are used to control user access to a web application resources. You can manage them (create, edit, and delete) from the backend using the *Audience/Permissions* section:



Name	Position
Do anything	100
Manage permissions	200
Manage roles	300
Manage users	400
Manage configurations	500
Manage cultures	600
Manage endpoints	700
Manage objects	800
Manage classes	900
Manage menus	1000

Each permission has name, code, and position:



Code required
DoAnything

Name required
Do anything

Position
100

Code

`Code` might be used to check permissions from code (see examples below).

Position

`Position` might be used to sort the permissions in the correct order.

Once you created a permission, you can assign it to a *role* (and then to a *user*). While signing in, all the user roles and all the permissions from that roles are attached to the user as the claims. These claims then can be checked from the code:

```
if (context.User.HasClaim(PlatformusClaimTypes.Permission, Permissions.ManageUsers))
{
}
```

Platformus uses authorization policies to control access to the controllers and actions:

```
[Authorize(Policy = Policies.HasManageUsersPermission)]
public class UsersController : ControllerBase { }
```

In order to be able to use an authorization policy, it should be added to the authorization options inside the `services.AddAuthorization()` extension method:

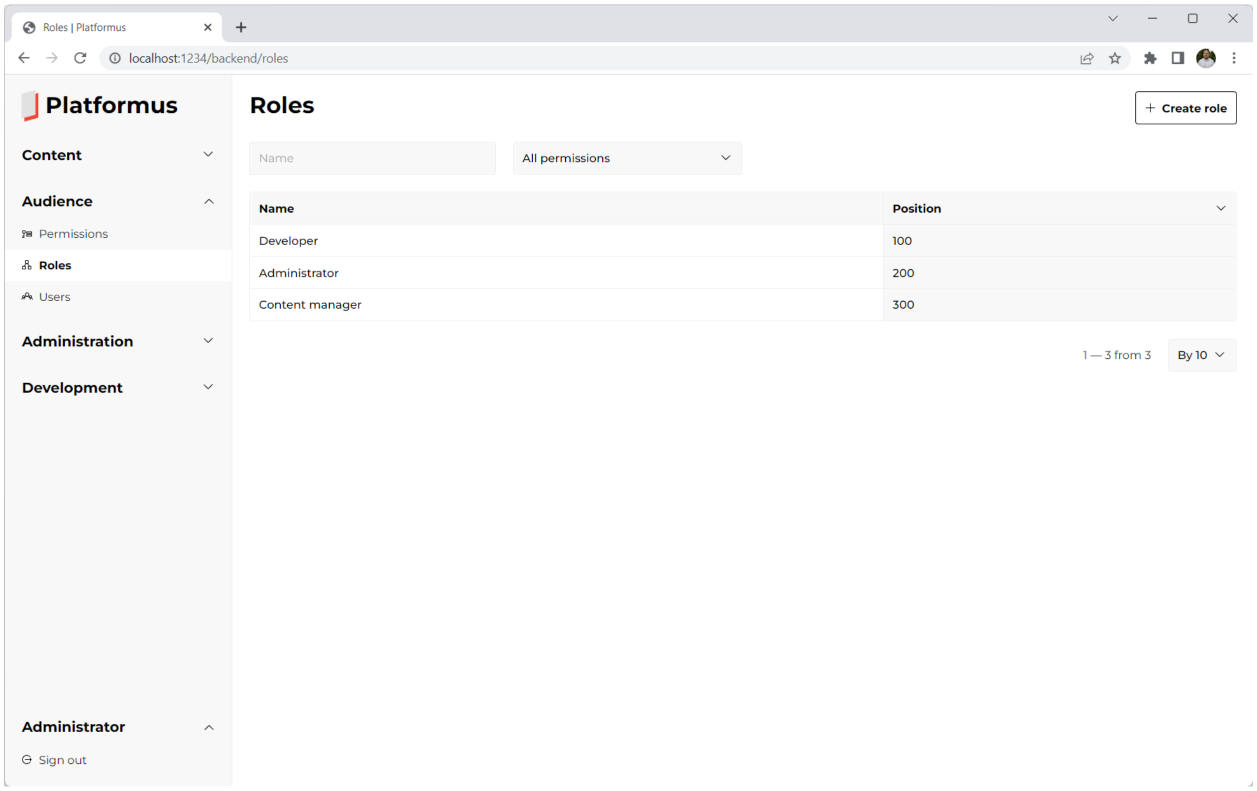
```
services.AddAuthorization(options =>
  {
    foreach (IAuthorizationPolicyProvider authorizationPolicyProvider in␣
↪ExtensionManager.GetInstances<IAuthorizationPolicyProvider>())
      options.AddPolicy(authorizationPolicyProvider.Name, authorizationPolicyProvider.
↪GetAuthorizationPolicy());
    }
  );
```

As you can see, the ExtCore framework's ExtensionManager class is used to get all the instances of the IAuthorizationPolicyProvider interface implementations. Then method `IAuthorizationPolicyProvider.GetAuthorizationPolicy()` is used to get the authorization policies.

So, if the permission is used to control access to a controller or action via policy, you need to implement the `IAuthorizationPolicyProvider` interface and then add corresponding attribute to the controller or action. If you only want to check the permission from code, you don't have to implement that interface.

Roles

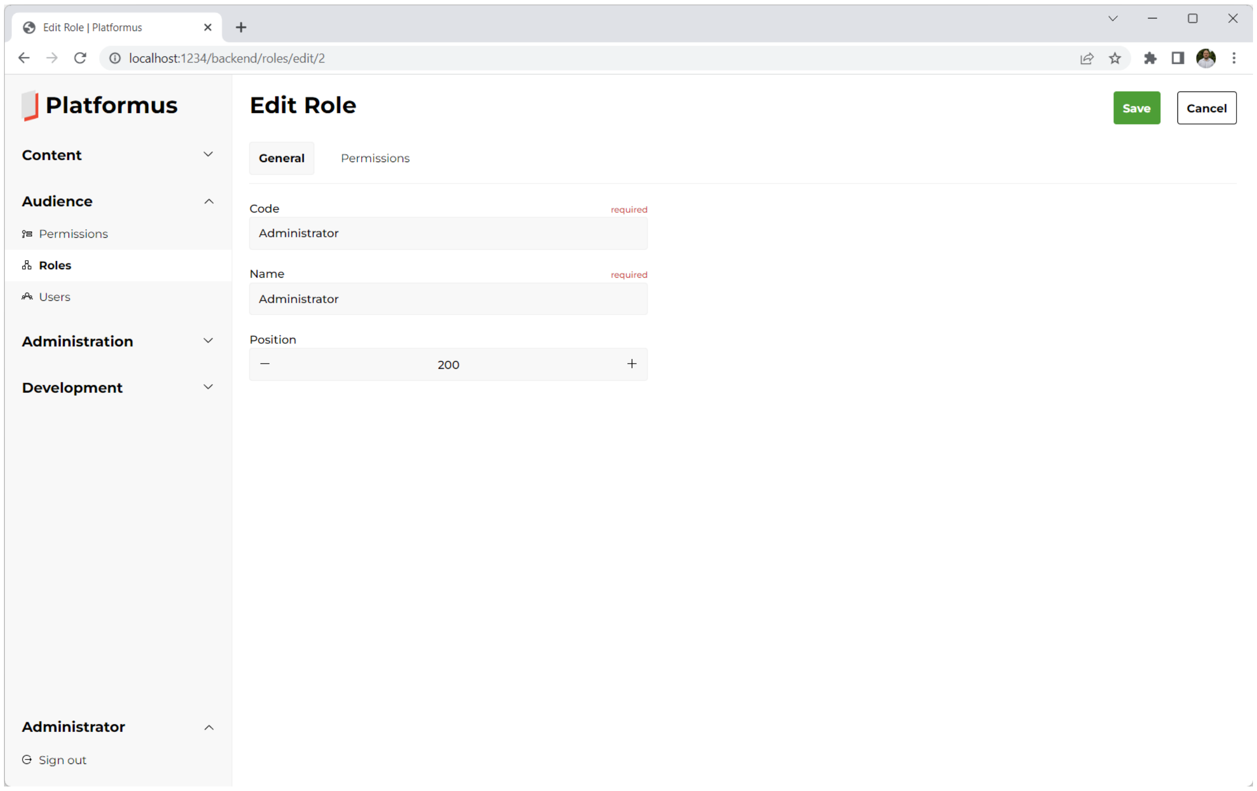
Roles are used to group the permissions. You can't assign a permission to a user directly, it is only possible to assign a role. You can manage them (create, edit, and delete) from the backend using the *Audience/Roles* section:



Name	Position
Developer	100
Administrator	200
Content manager	300

1 — 3 from 3 By 10

Each role has name, code, position, and other fields:



Code (required)	Administrator
Name (required)	Administrator
Position	200

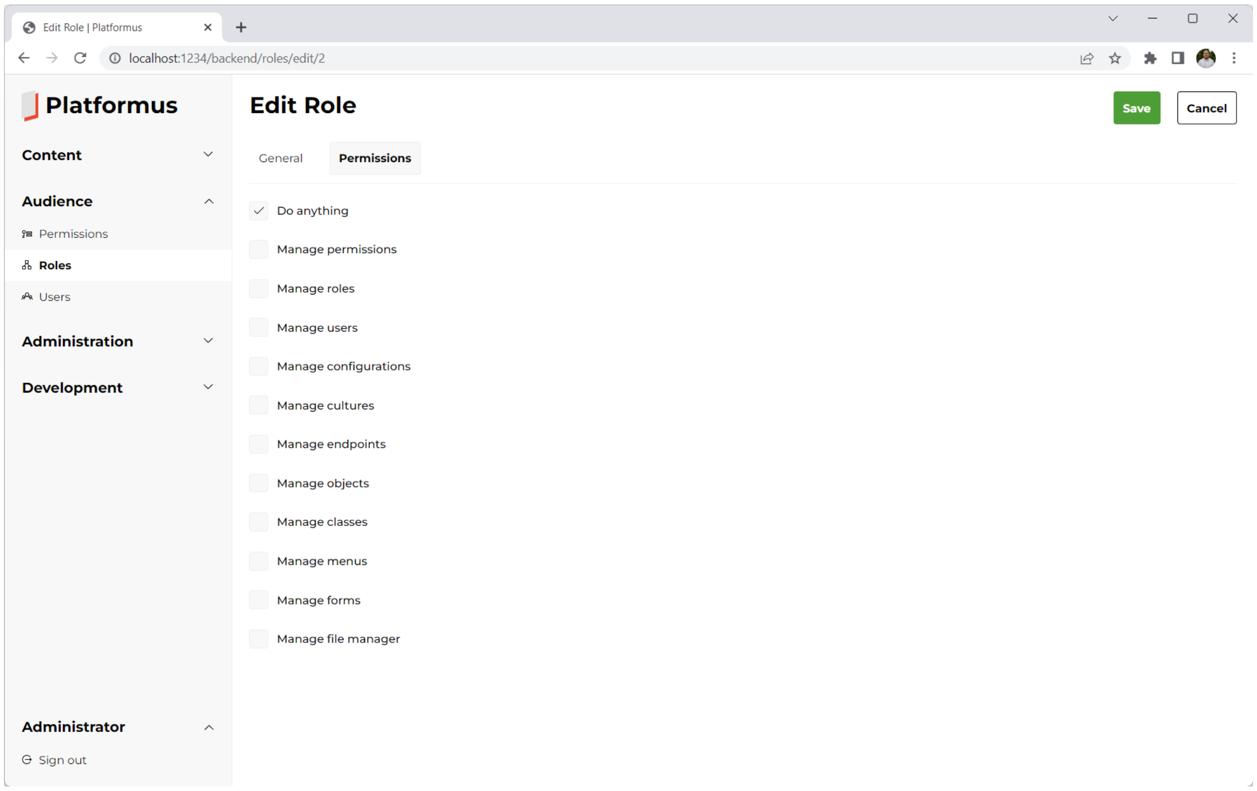
General/Code

`Code` might be used to check roles from code.

General/Position

`Position` might be used to sort the roles in the correct order.

Also, on the *Permissions* tab you can assign the permissions to a role:

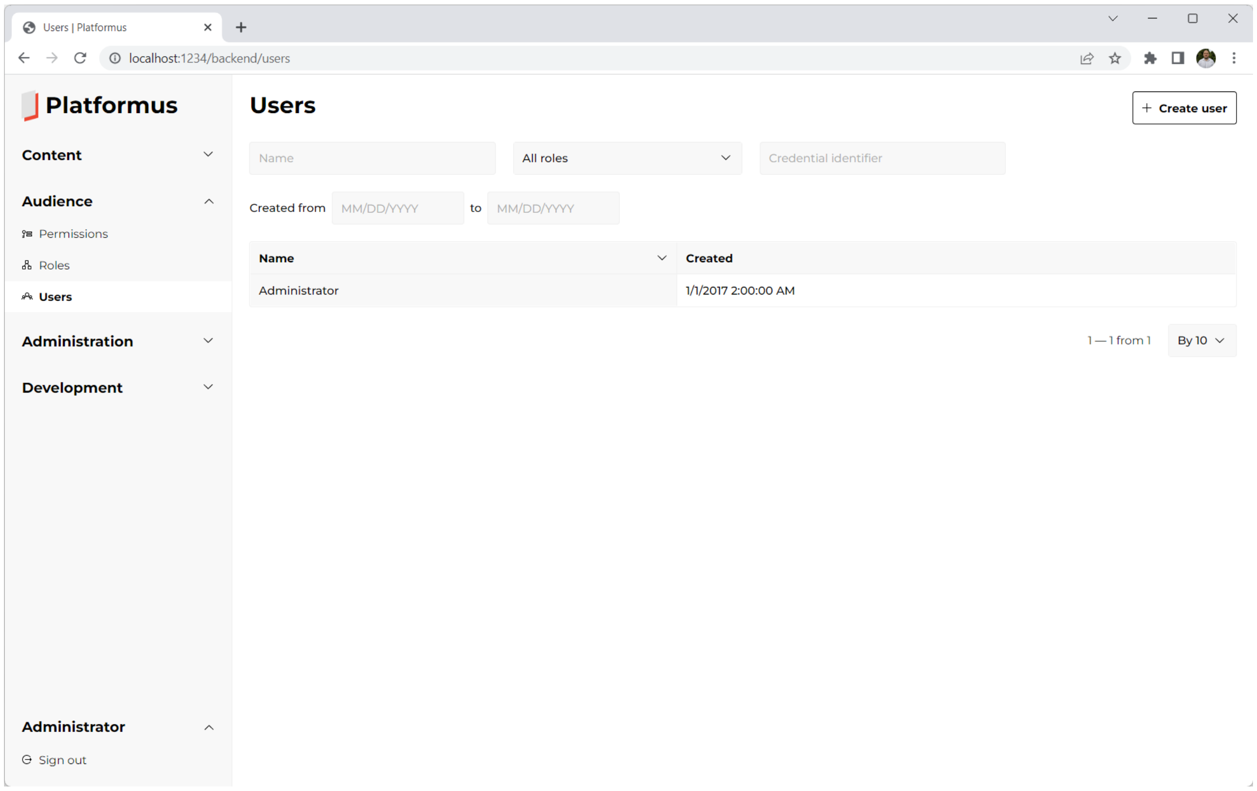


✓ Do anything
Manage permissions
Manage roles
Manage users
Manage configurations
Manage cultures
Manage endpoints
Manage objects
Manage classes
Manage menus
Manage forms
Manage file manager

The same as *permissions*, roles are attached to a user as the claims while signing in. You can check them from the code to control rights and restrict access, but permissions checking is preferred.

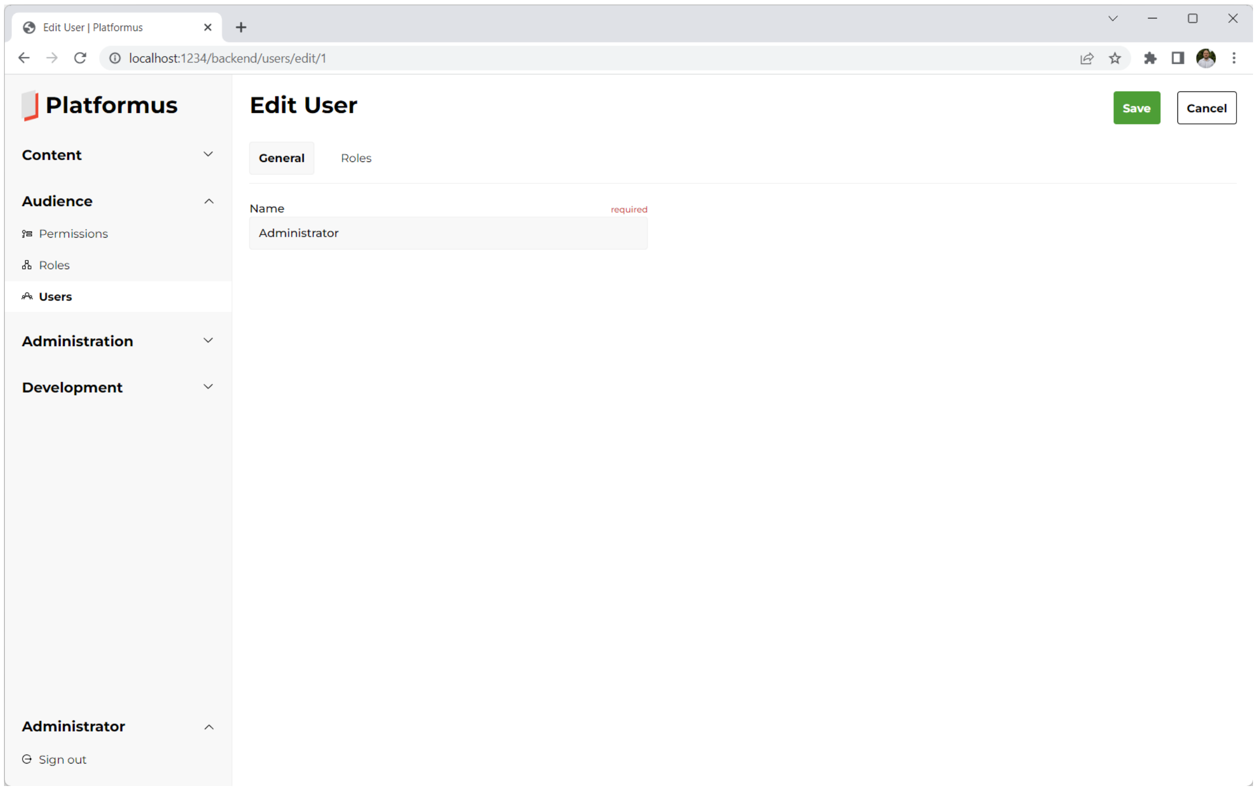
Users

Users (as well as *permissions* and *roles*) are used to control access to a web application resources. You can manage them (create, edit, and delete) from the backend using the *Audience/Users* section:



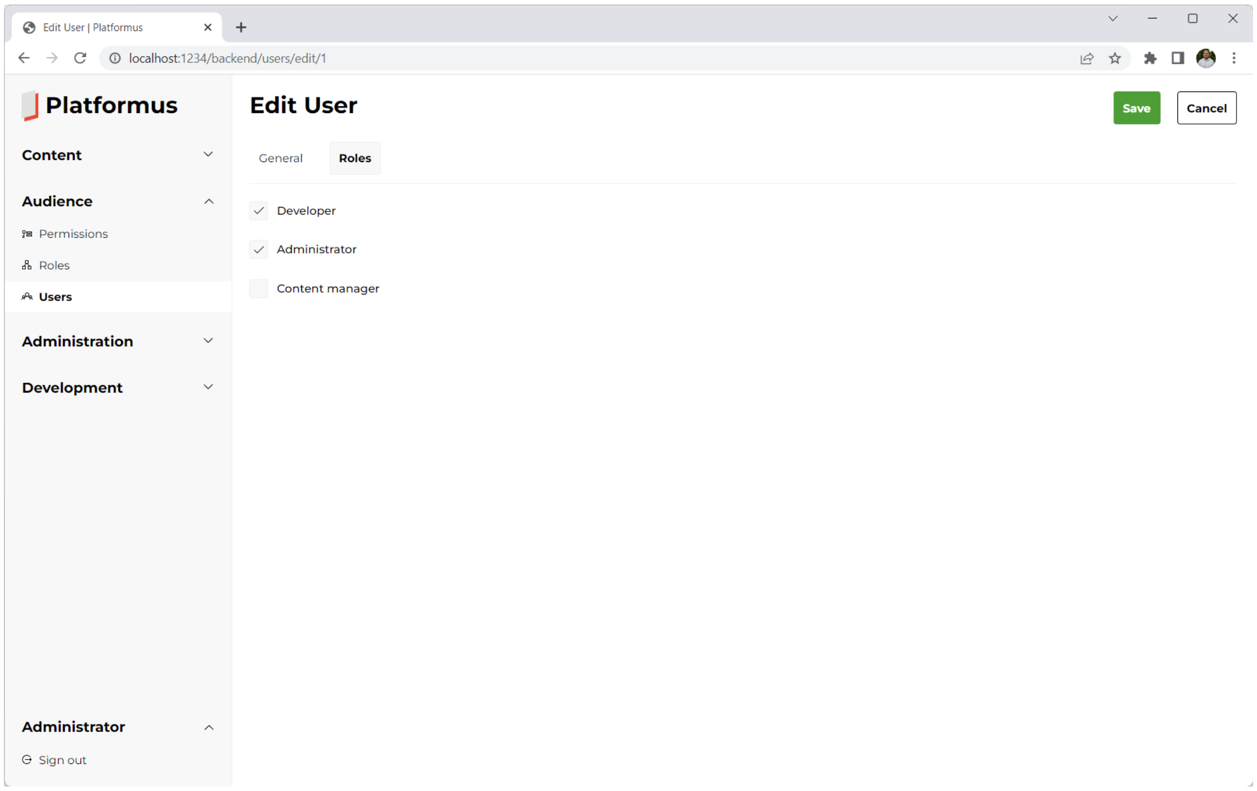
Name	Created
Administrator	1/1/2017 2:00:00 AM

Each user has name:



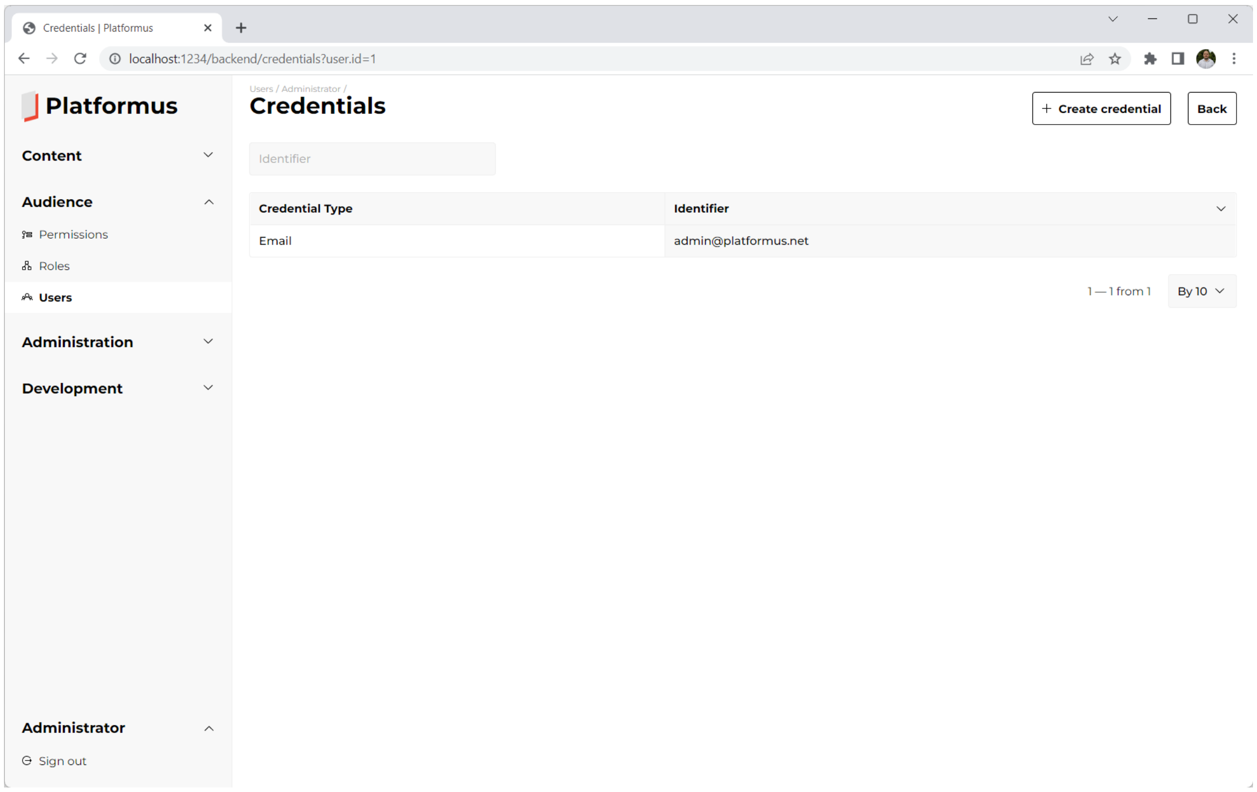
Name: Administrator

Also, on the *Roles* tab you can assign the roles to a user:



As user itself only has a name, he doesn't store any information about how he signs in. This information is stored using the Credential objects. Each user can have different credentials, and each credential has its type (it can be email and password, Facebook account, Microsoft account and so on). When user signs in, Platformus checks if there is a credential with the given type, identifier, and secret exists. If the credential is found, corresponding user is signed in.

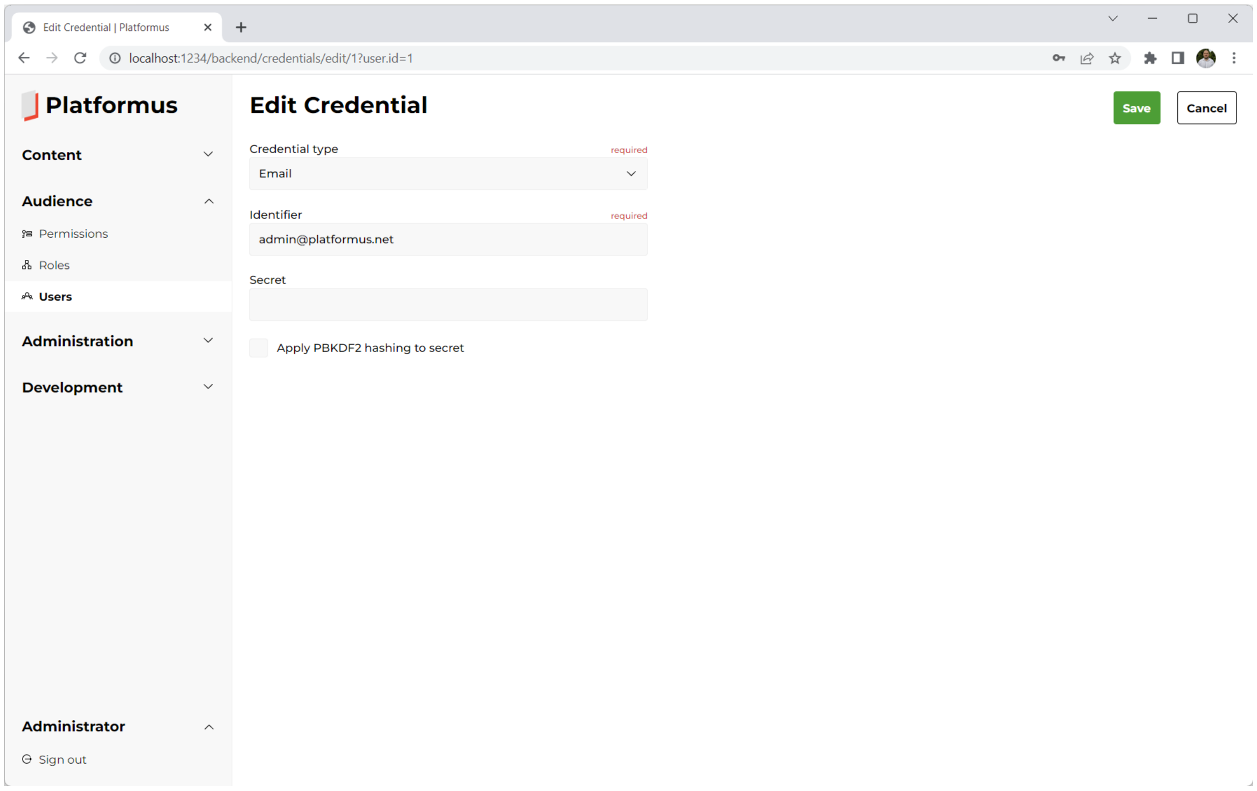
The credential list looks like this:



Credentials

Credential Type	Identifier
Email	admin@platformus.net

Each credential has type, identifier, and secret:



Edit Credential

Credential type (required)	Email
Identifier (required)	admin@platformus.net
Secret	

Apply PBKDF2 hashing to secret

Credential type

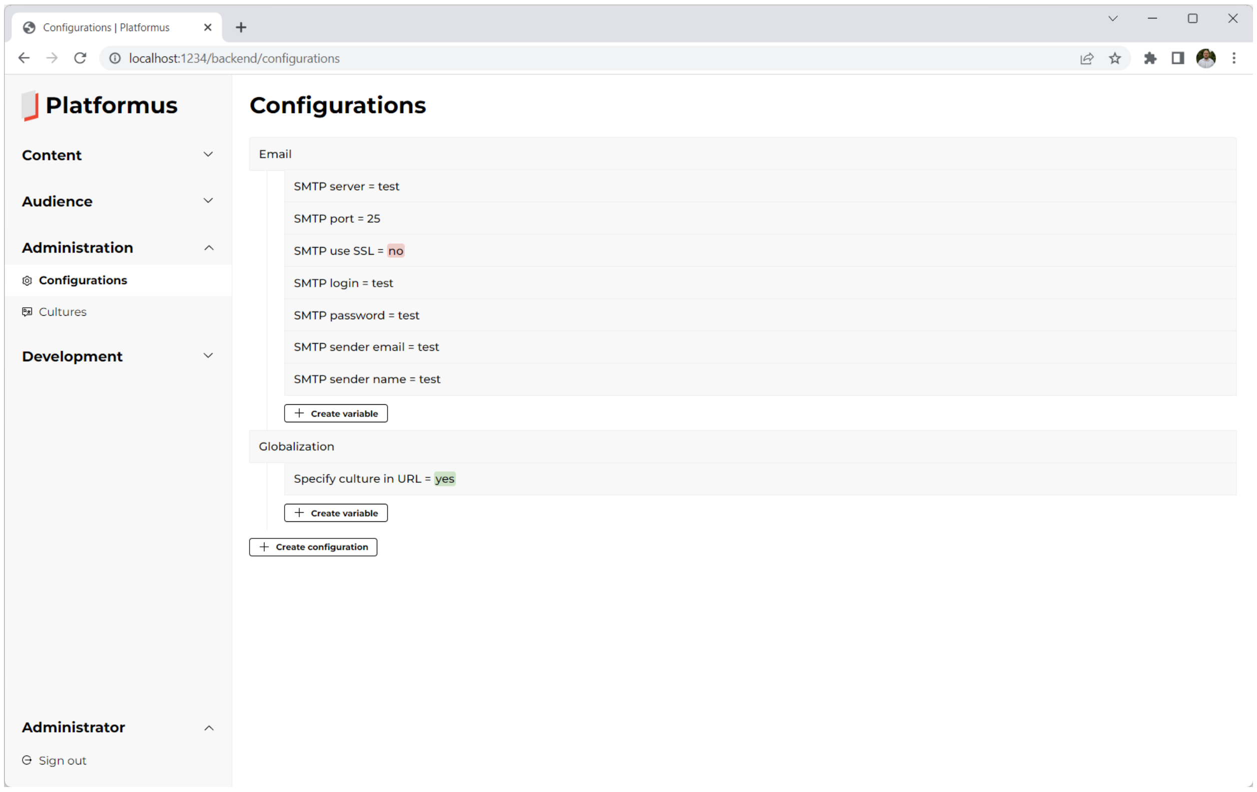
As a developer, you can create your own credential types which depend on the sign in methods your application supports.

Secret

`Secret` is optional and can be used to store any additional information. For example, it stores passwords (as hashes) for the email and password credential type. If you need to change the password, just type it into this field. Don't forget to set the *Apply PBKDF2 hashing to secret* checkbox to apply hashing, otherwise your password will be saved as plain text and signing in won't work (as it compares hashes).

Configurations

Configurations and variables are used to provide user-defined configuration parameters to the web application (while developer-defined ones can be provided using the appsettings.json file). You can manage them (create, edit, and delete) from the backend using the *Administration/Configurations* section:



Configurations

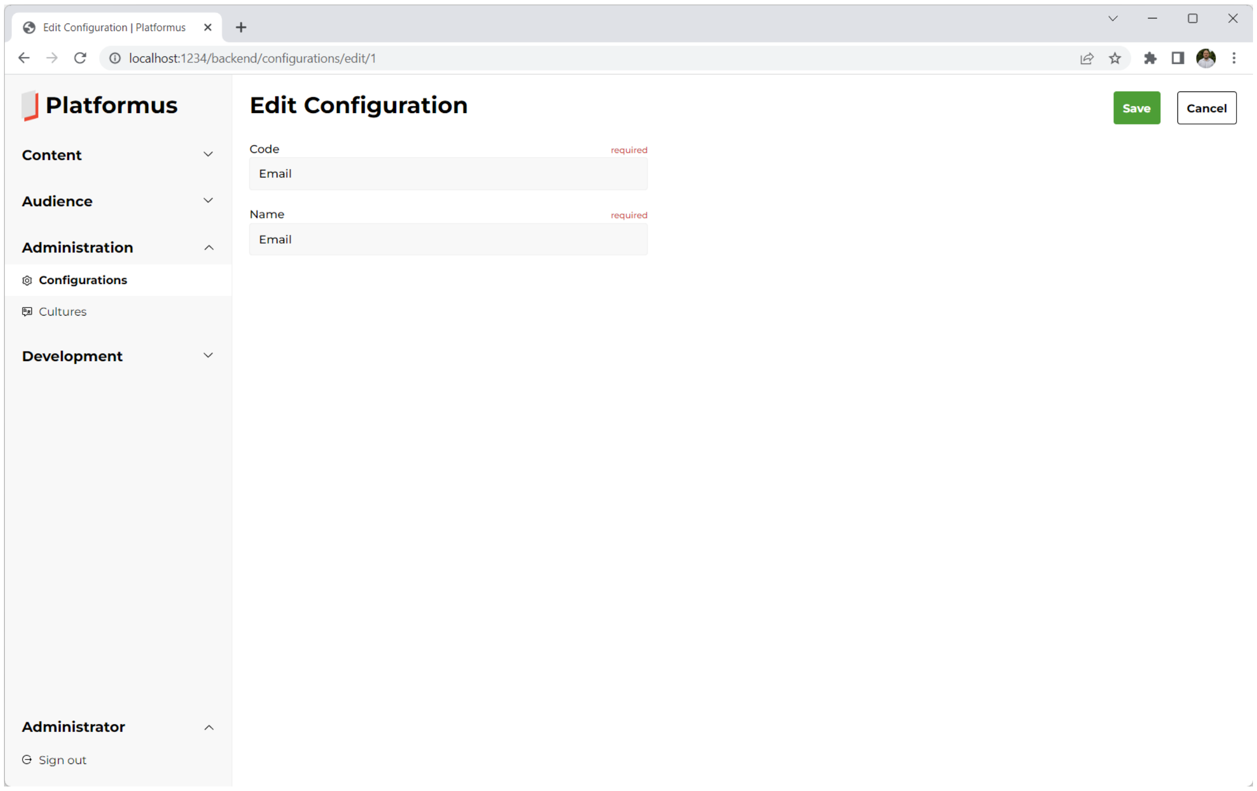
Email

SMTP server = test
SMTP port = 25
SMTP use SSL = no
SMTP login = test
SMTP password = test
SMTP sender email = test
SMTP sender name = test

Globalization

Specify culture in URL = yes

Each configuration has code and name:

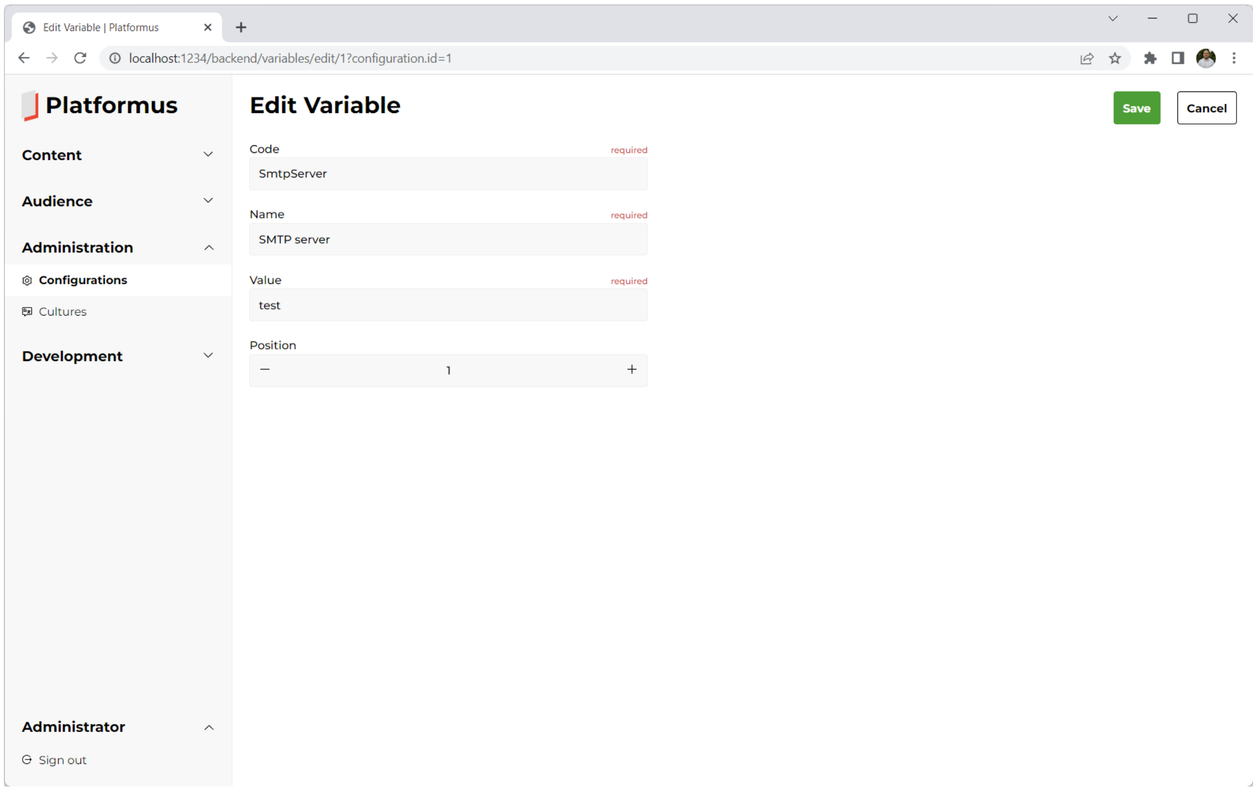


Code (required)	Email
Name (required)	Email

Code

`Code` might be used to get configuration from code (see examples below).

Configurations consist of variables. Each variable has code, name, value, and position:



Code required
SmtpServer

Name required
SMTP server

Value required
test

Position
1

Code

The same as for configurations, `Code` is used to get variable from code.

Value

`Value` contains the variable's value.

Position

`Position` might be used to sort the variables in the correct order.

There is the special DefaultConfigurationManager class that you can use to access the configurations. It implements the IConfigurationManager interface and it is registered as a service inside the DI, so you can replace it with your own implementation.

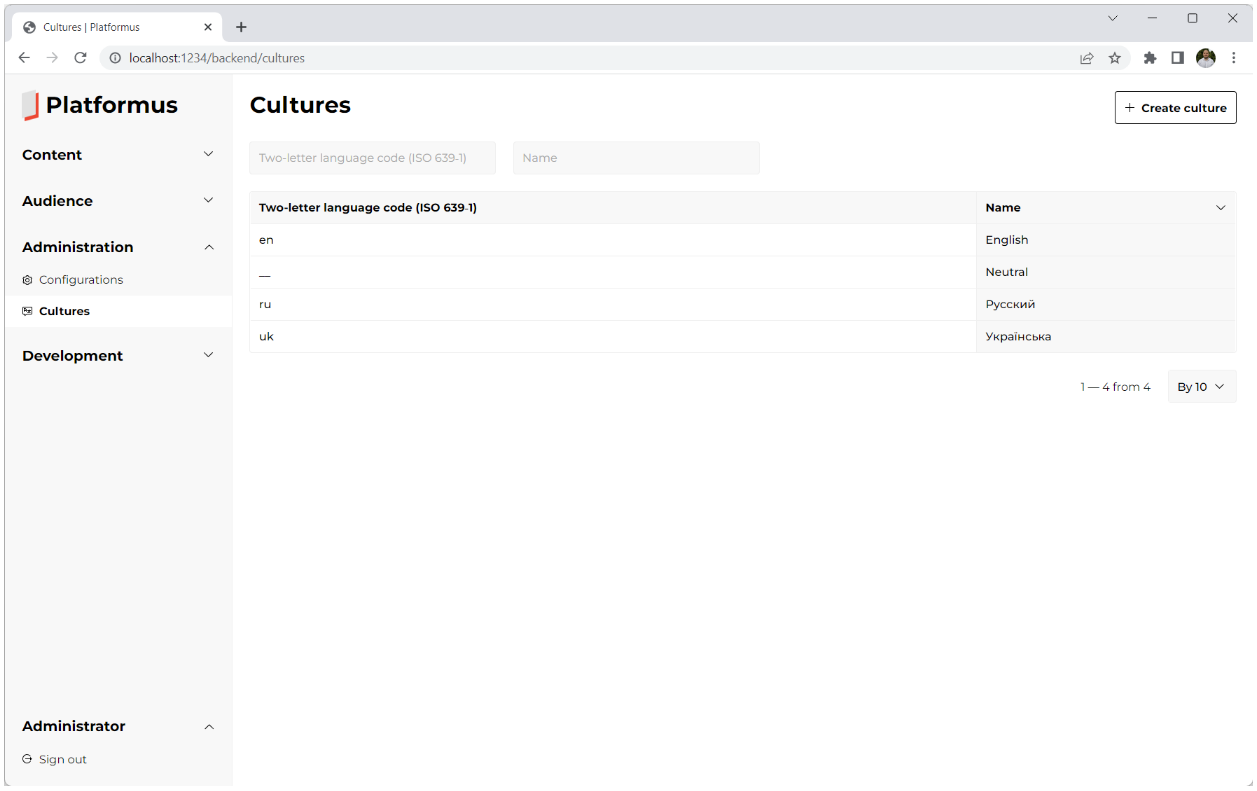
This is the usage example:

```
public class DefaultController : Controller
{
  public DefaultController(IConfigurationManager configurationManager)
  {
    string emailSmtpServer = configurationManager["Email", "SmtpServer"];
  }
}
```

This will get value of the variable with code `SmtpServer` from the configuration with code `Email`.

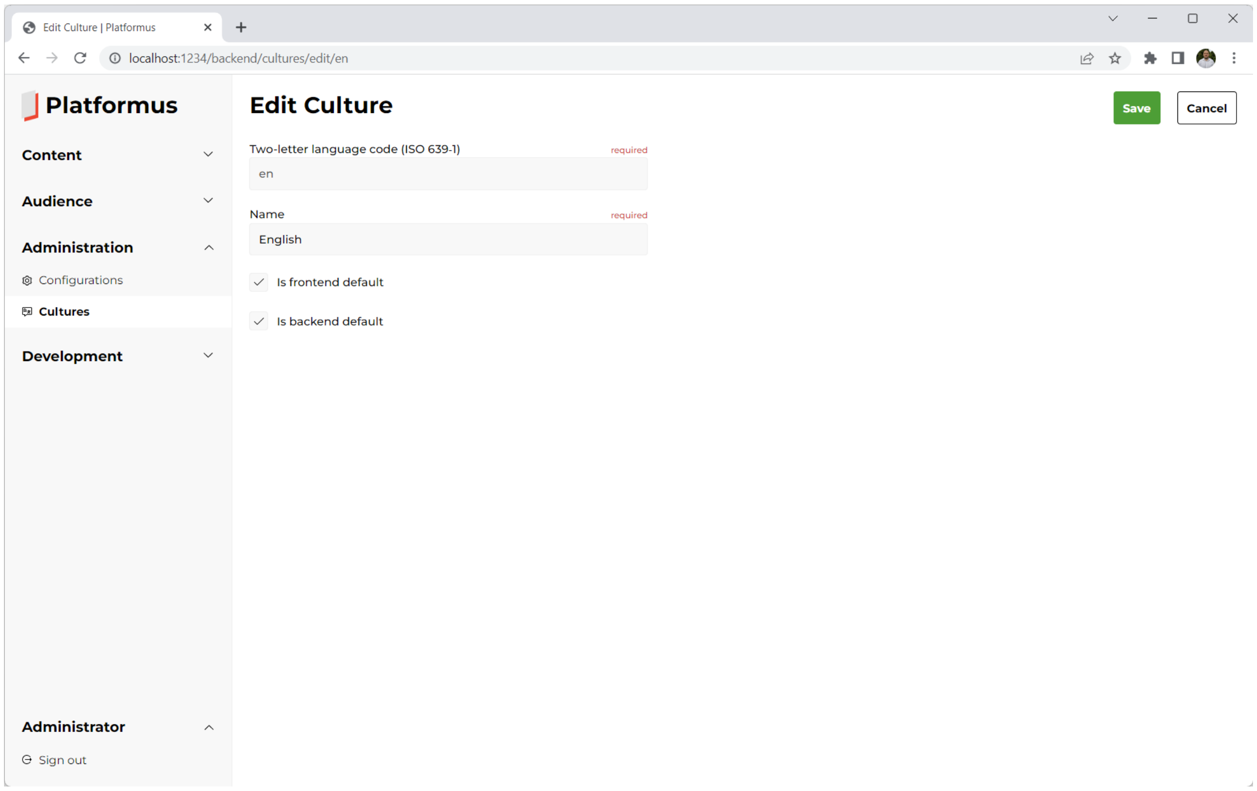
Cultures

Cultures are used to specify which languages and data formats your web application supports. You can manage them (create, edit, and delete) from the backend using the *Administration/Cultures* section:



Two-letter language code (ISO 639-1)	Name
en	English
—	Neutral
ru	Русский
uk	Українська

Each culture has two-letter language code, name, and other fields:



Two-letter language code (ISO 639-1) required
en
Name required
English
✓ Is frontend default
✓ Is backend default

Two-letter language code

This code complies with ISO 639-1. It is used, for example, as URL segment to determine request's language (see below).

Is frontend default

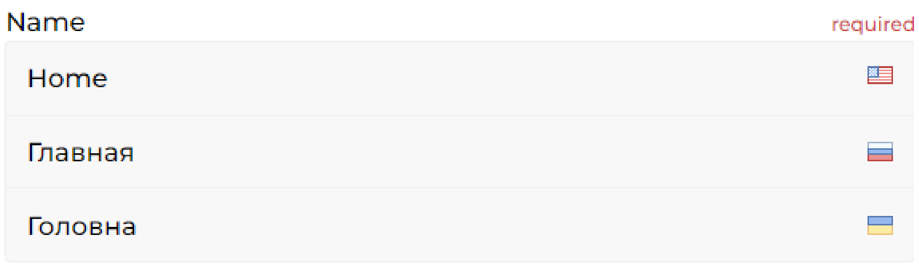
Specifies if this culture should be used as the default one on the frontend (it means that this culture will be used when the culture is not explicitly provided).

Is backend default

Specifies if this cultured should be used as the backend (admin panel) one.

You can see that there is the Neutral culture exists in the list. This culture is used by the *Platformus.Website* extension to store the culture-neutral string values using the localizations.

When you create your own extension or describe your data model with classes and members using the *Platformus.Website* one, you can specify whether the particular string property is localizable or not. If it is localizable, N editors will be displayed, one for each of the cultures. It looks like this:



Home
Главная
Головна

When your string property is not localizable, the only one editor will be displayed, and the property value will be saved either using the localization with neutral culture (in case the *Platformus.Website* extension is used) or in the way you want it to be saved.

When using the *Platformus.Website* extension, by default a short two-letter language code segment is used in the URL on the frontend to specify which culture should be used for the request. For example: /en/some-page. It is done in this way to make it possible for the pages to be indexed by the search engines with the different languages. But if you are sure that your web application will always support the only one language, you can turn off this behavior using the *configurations* and have shorter URLs. In this case, the default culture will be used to display the content (but you can change the way culture is selected for the requests; for example, the `Accept-Language` header can be used).

There is the special DefaultCultureManager class that you can use to work with the cultures. It implements the ICultureManager interface and it is registered as a service inside the DI, so you can replace it with your own implementation.

1.3.2 Packages

- Platformus.Core
- Platformus.Core.Backend
- Platformus.Core.Data.Entities
- Platformus.Core.Data.EntityFramework.PostgreSql
- Platformus.Core.Data.EntityFramework.Sqlite
- Platformus.Core.Data.EntityFramework.SqlServer
- Platformus.Core.Frontend

1.4 Platformus.Website

This extension provides everything for building simple websites without (or almost without) programming. You can describe your custom content types and manage your data using the automatically generated admin panel UI. For example, you can have blog posts, tags, and comments with all the properties you need.

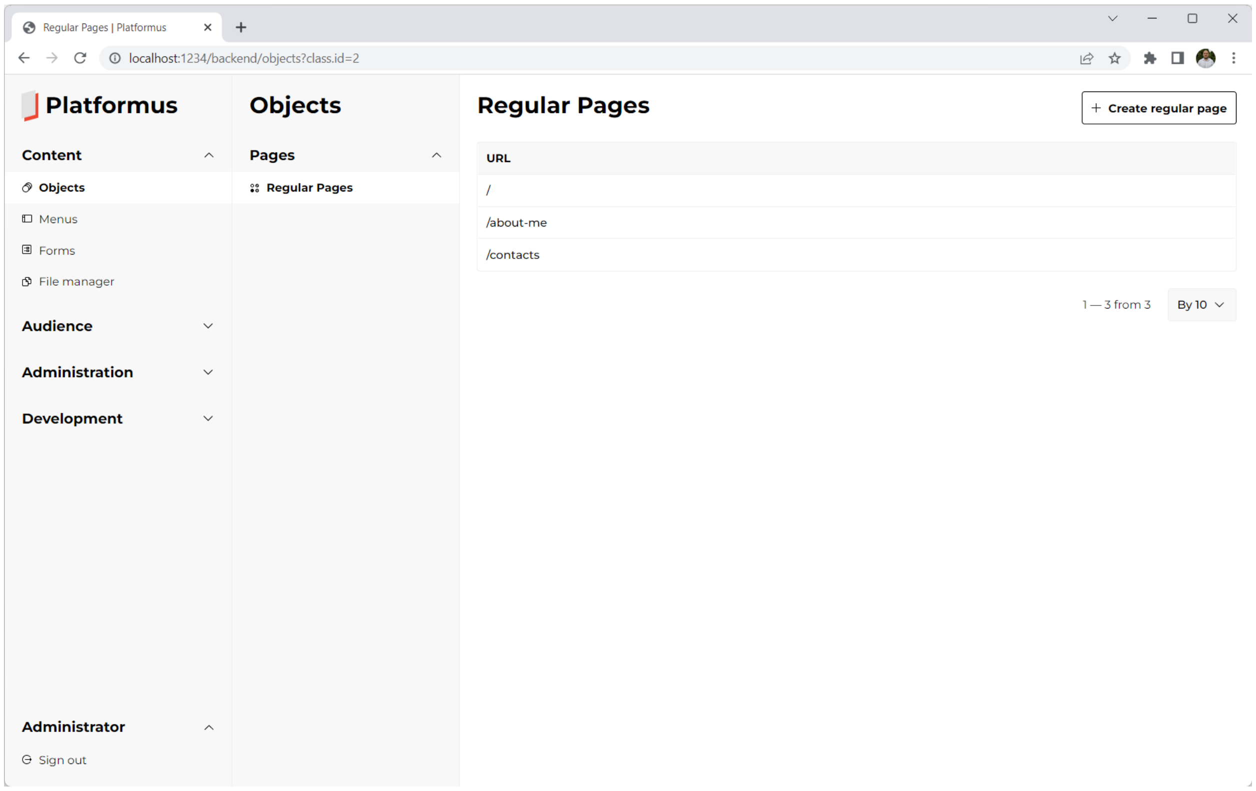
Also, there are built-in features to work with *menus* and *forms*. You can create them in the backend and then display on frontend using the tag helpers.

A basic *file manager* is also provided by Platformus.Website extension.

1.4.1 Admin Panel Sections

Objects

Object is the central part of the default Platformus CMS data model. It is an elementary piece of the information. It can be your blog post, comment, or tag. Objects are described by *classes* and *members* and consist of properties and relations. You can manage them (create, edit, and delete) from the backend using the *Content/Objects* section:

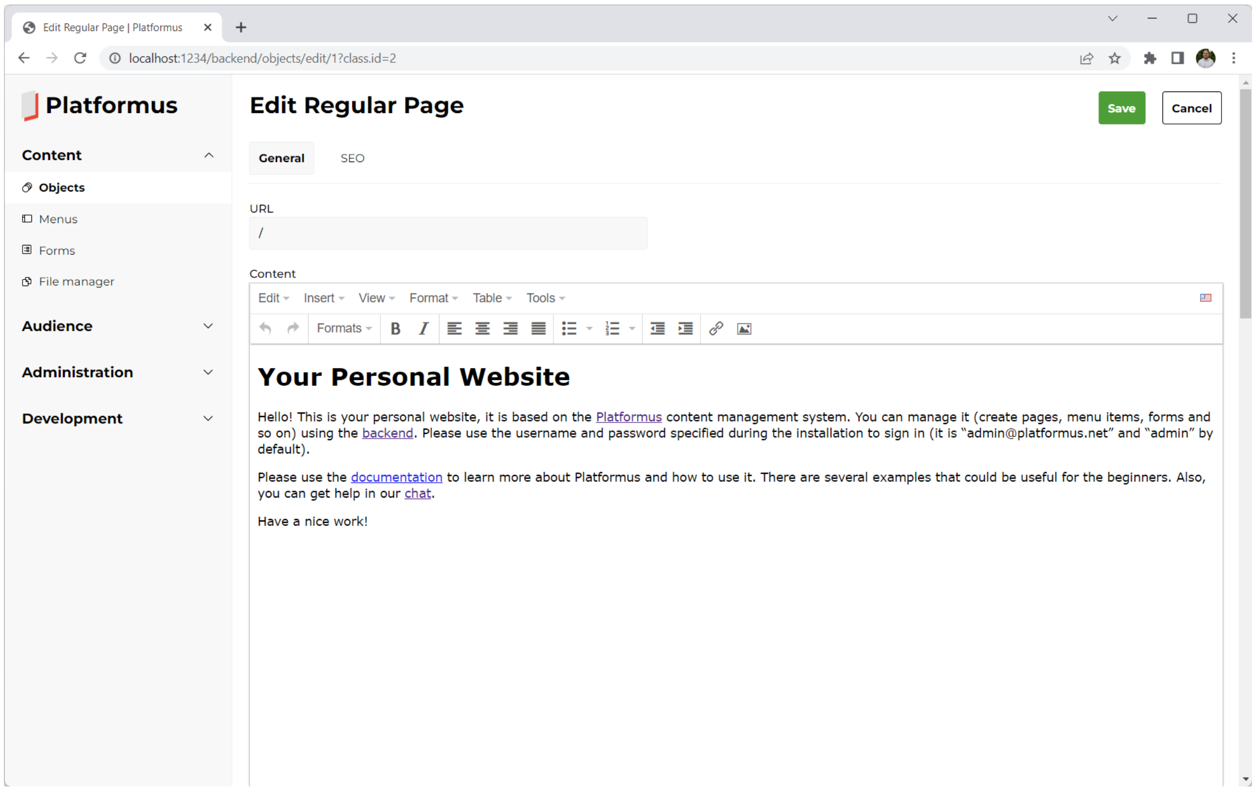


URL
/
/about-me
/contacts

The *Content/Objects* section has subsections that correspond to classes. These subsections are grouped using the parent classes. Classes without parents go under the Other group.

As we said, objects consist of properties and relations and are described by *classes* and *members*. Each property has its own client-side editor, which is specified by the data type of the member. As a developer, you can create your own data types and client-side editors. Also, data types specify which primitive storage data type should be used to store the particular property value (integer, decimal, string and datetime are supported). String properties can be localizable and non-localizable.

The object edit page consist of client-side editors grouped by *tabs*. A typical page can look like this:



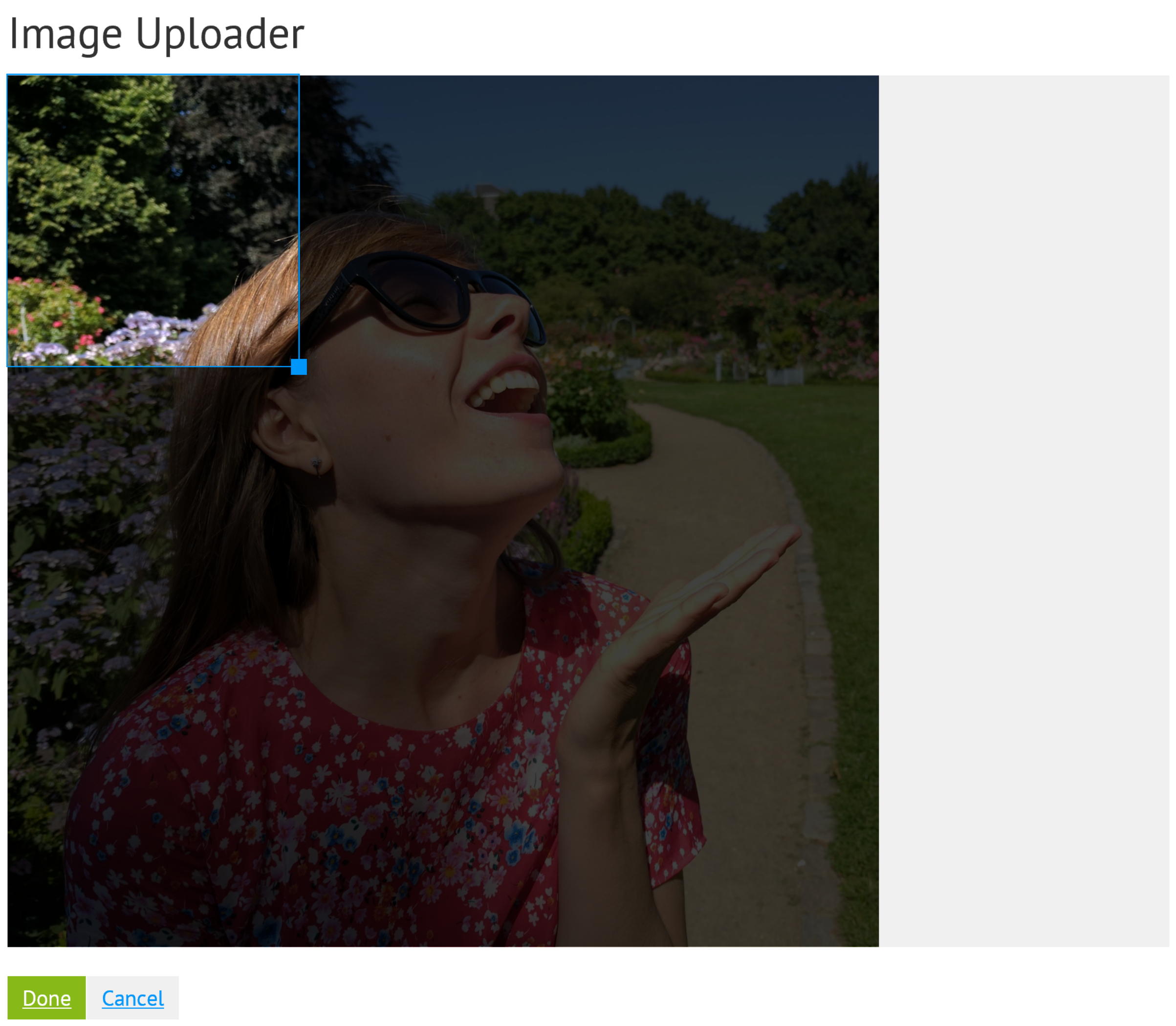
Your Personal Website

Hello! This is your personal website, it is based on the Platformus content management system. You can manage it (create pages, menu items, forms and so on) using the backend. Please use the username and password specified during the installation to sign in (it is "admin@platformus.net" and "admin" by default).

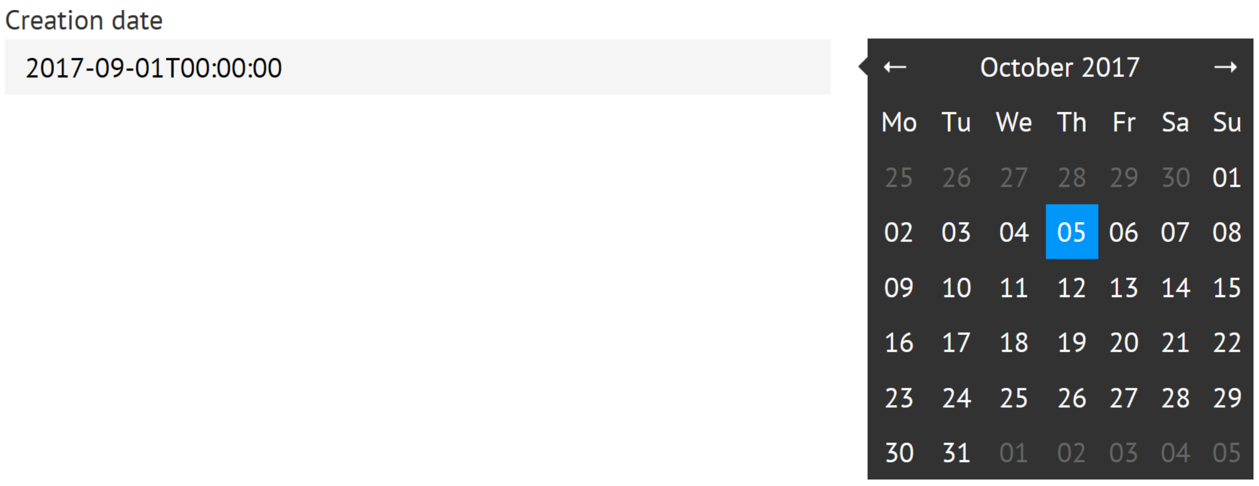
Please use the documentation to learn more about Platformus and how to use it. There are several examples that could be useful for the beginners. Also, you can get help in our chat.

Have a nice work!

The property editors may look and behave absolutely different. This is the property editor for the `Image` data type looks like:



And this is the one for the `Date` data type:

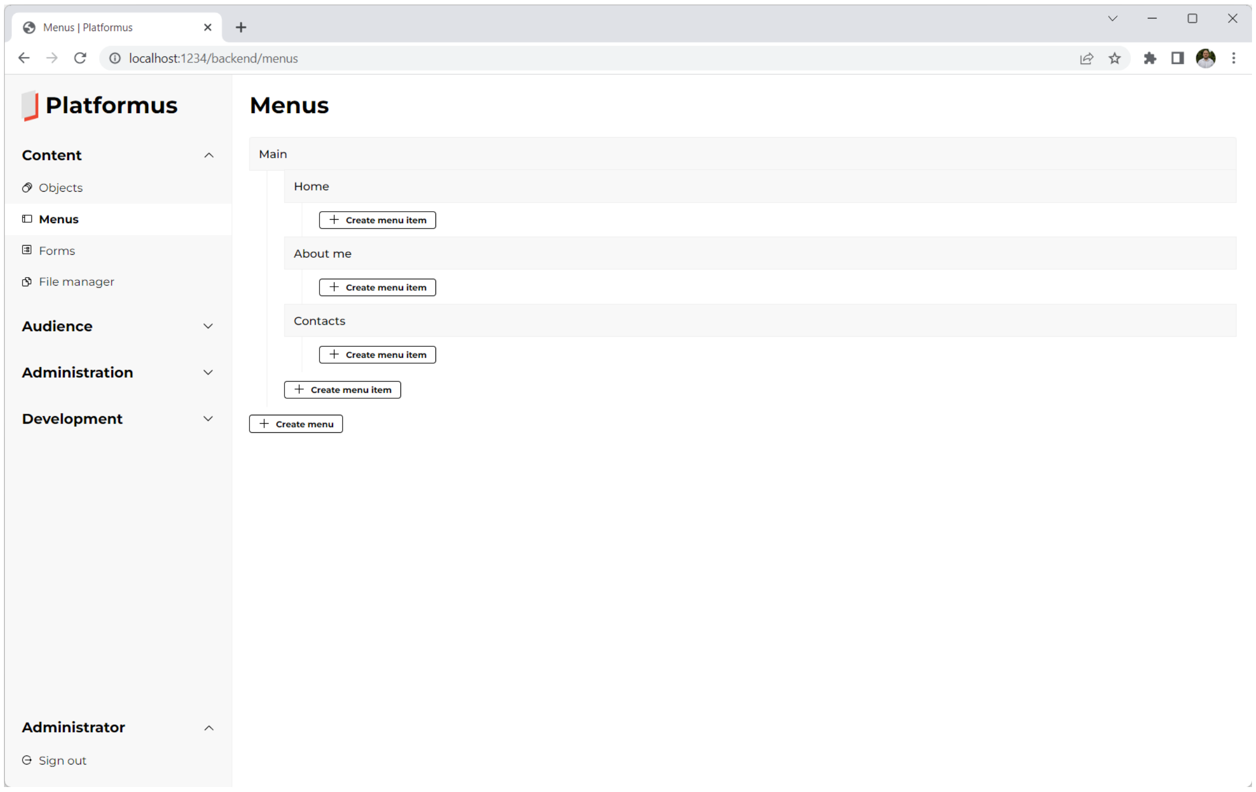


2017-09-01T00:00:00

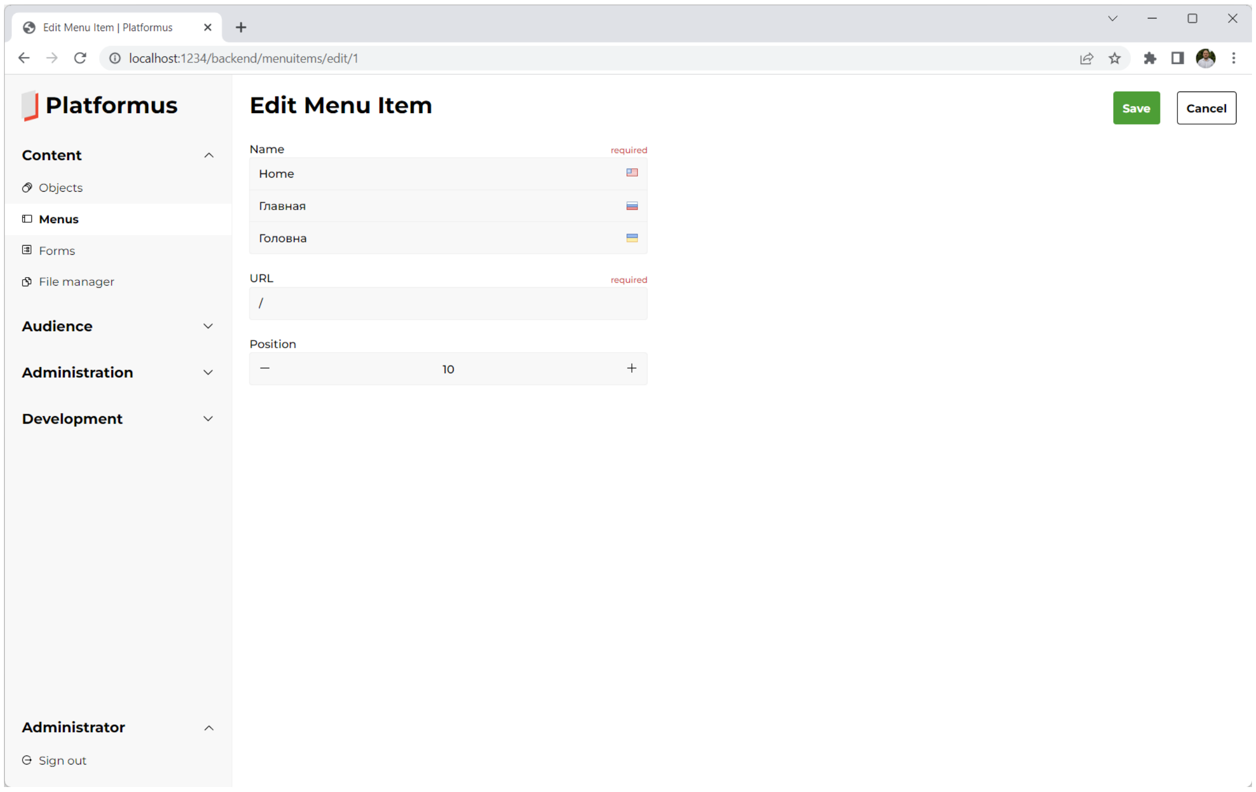
Mo	Tu	We	Th	Fr	Sa	Su
25	26	27	28	29	30	01
02	03	04	05	06	07	08
09	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	01	02	03	04	05

Menus

Menus are used for navigation on the frontend. You can manage them (create, edit, and delete) from the backend using the *Content/Menus* section:



Each menu item has localized name, URL, and position:



Name required
Home
Главная
Головна
URL required
/
Position
10

URL

`URL` is where user is redirected when clicks the menu item.

Position

`Position` might be used to sort the menu items in the correct order within the menu.

Once menu is created, you can display it on the frontend using the built-in `MenuViewComponent` view component like this (the menu code is passed as the parameter to identify the menu we want to display):

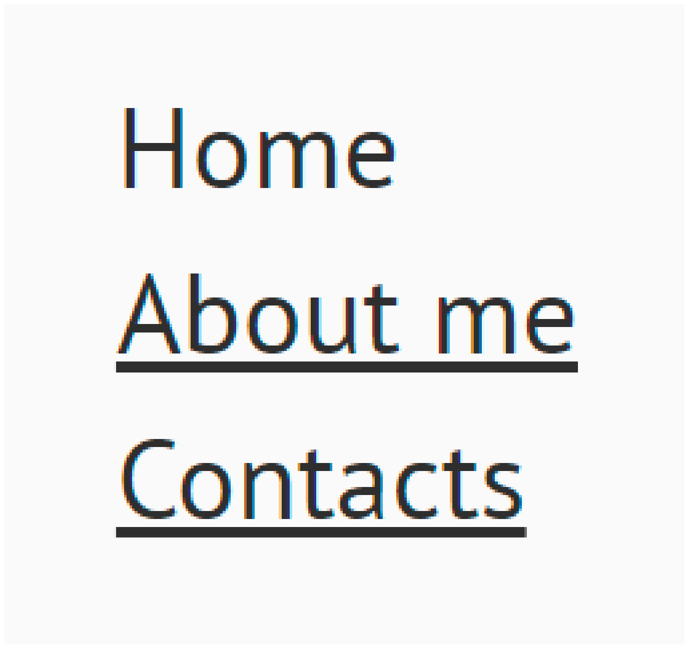
```
@await Component.InvokeAsync("Menu", new { code = "Main", additionalCssClass =
↪"master-detail__menu" })
```

Or using the view component tag helper:

```
<vc:menu code="Main" additional-css-class="master-detail__menu" />
```

As you can see, an additional CSS class might be applied using the corresponding optional parameter.

The result can look something like this (note that the current menu item is highlighted):



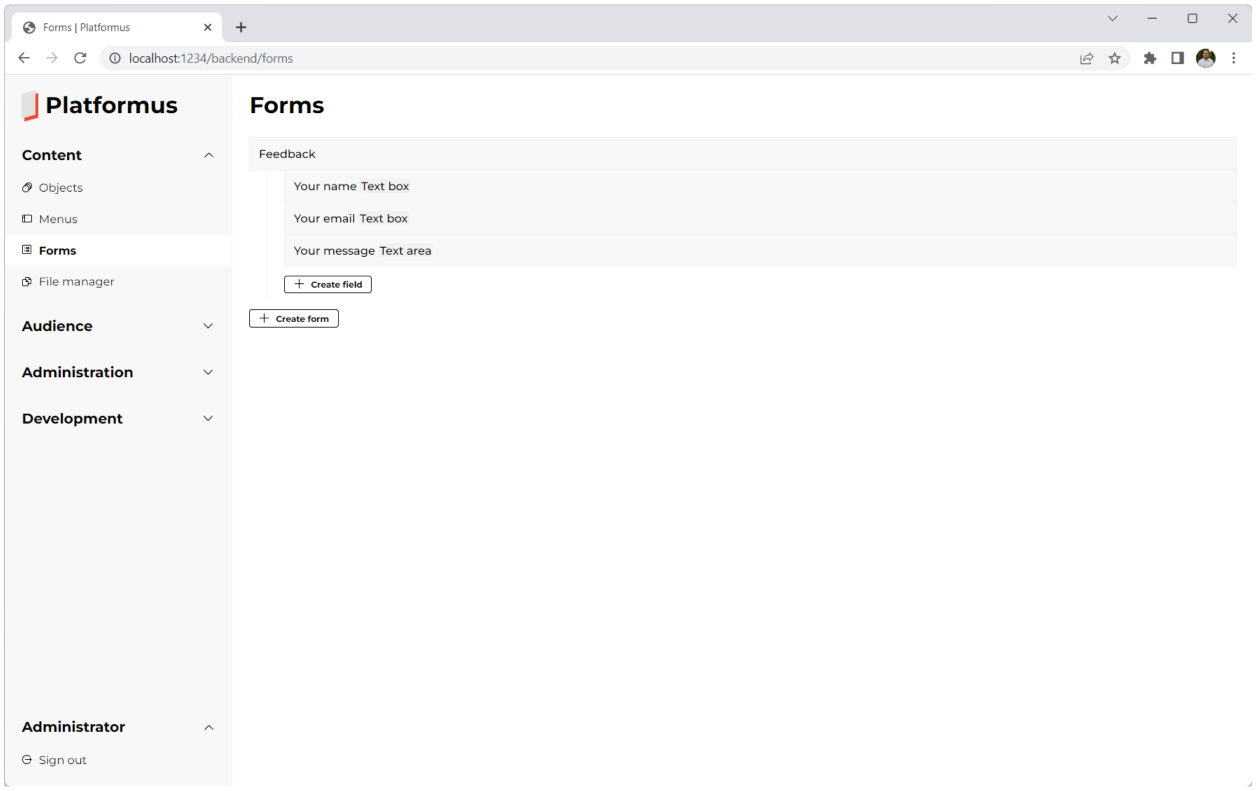
Menus are displayed using the built-in views (_Menu and _MenuItem). The HTML elements have unique CSS classes (the BEM methodology is used), so it is easy to apply styles to them:

```
<div class="menu master-detail__menu">
  <div class="menu__items">
    <div class="menu__item menu__item--active">
      <a class="menu__item-name menu__item-name--active" href="/en/">Home</a>
    </div>
    <div class="menu__item">
      <a class="menu__item-name" href="/en/about-me">About me</a>
    </div>
    <div class="menu__item">
      <a class="menu__item-name" href="/en/contacts">Contacts</a>
    </div>
  </div>
</div>
```

If you want to change the HTML, just copy these default views into your project and they will be used instead of the built-in ones, so you will be able to modify them as you want.

Forms

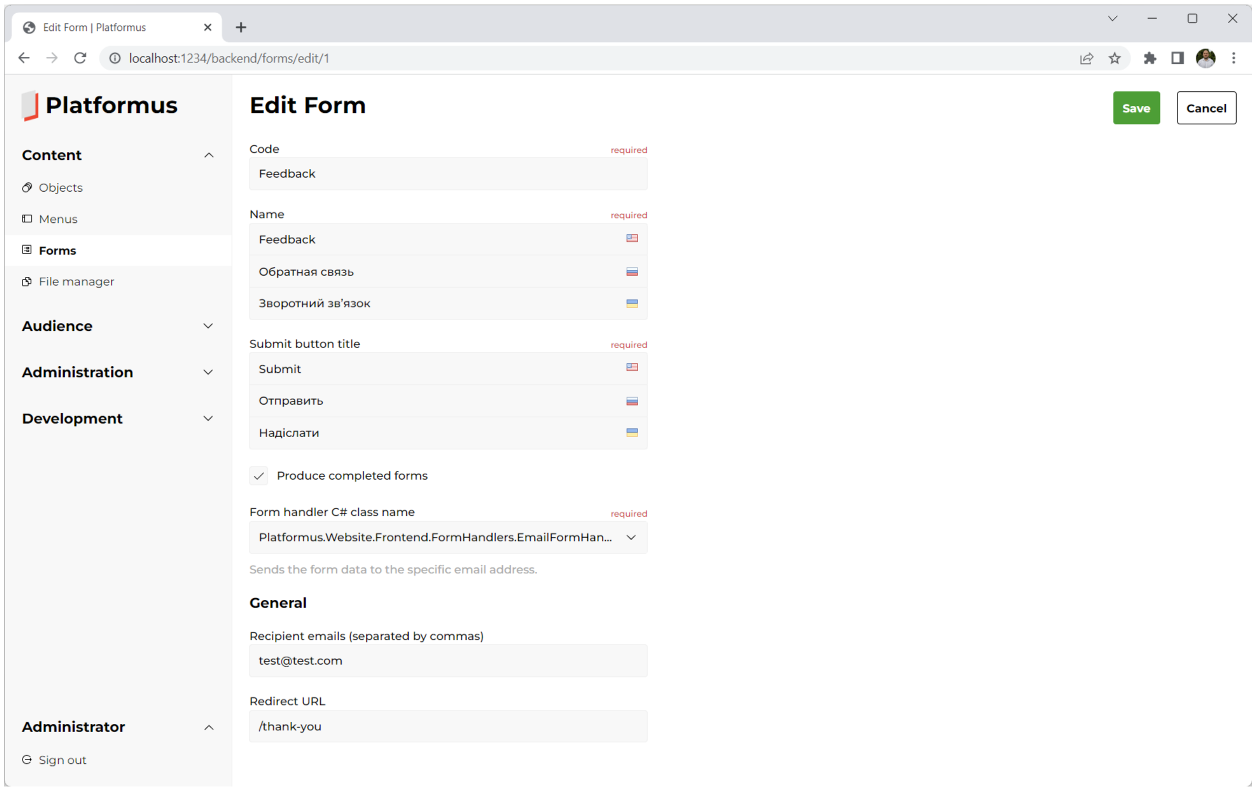
Forms are used to get and process user input on the frontend. You can manage them (create, edit, and delete) from the backend using the *Content/Forms* section:



Forms

Feedback
- Your name Text box
- Your email Text box
- Your message Text area

Each form has code, localized name, submit button title, and other fields:



Edit Form

Code (required): Feedback

Name (required): Feedback / Обратная связь / Зворотний зв'язок

Submit button title (required): Submit / Отправить / Надіслати

✓ Produce completed forms

Form handler C# class name (required): Platformus.Website.Frontend.FormHandlers.EmailFormHan...

Sends the form data to the specific email address.

General

Recipient emails (separated by commas): test@test.com

Redirect URL: /thank-you

Code

`Code` might be used to get forms from code (see examples below).

Produce completed forms

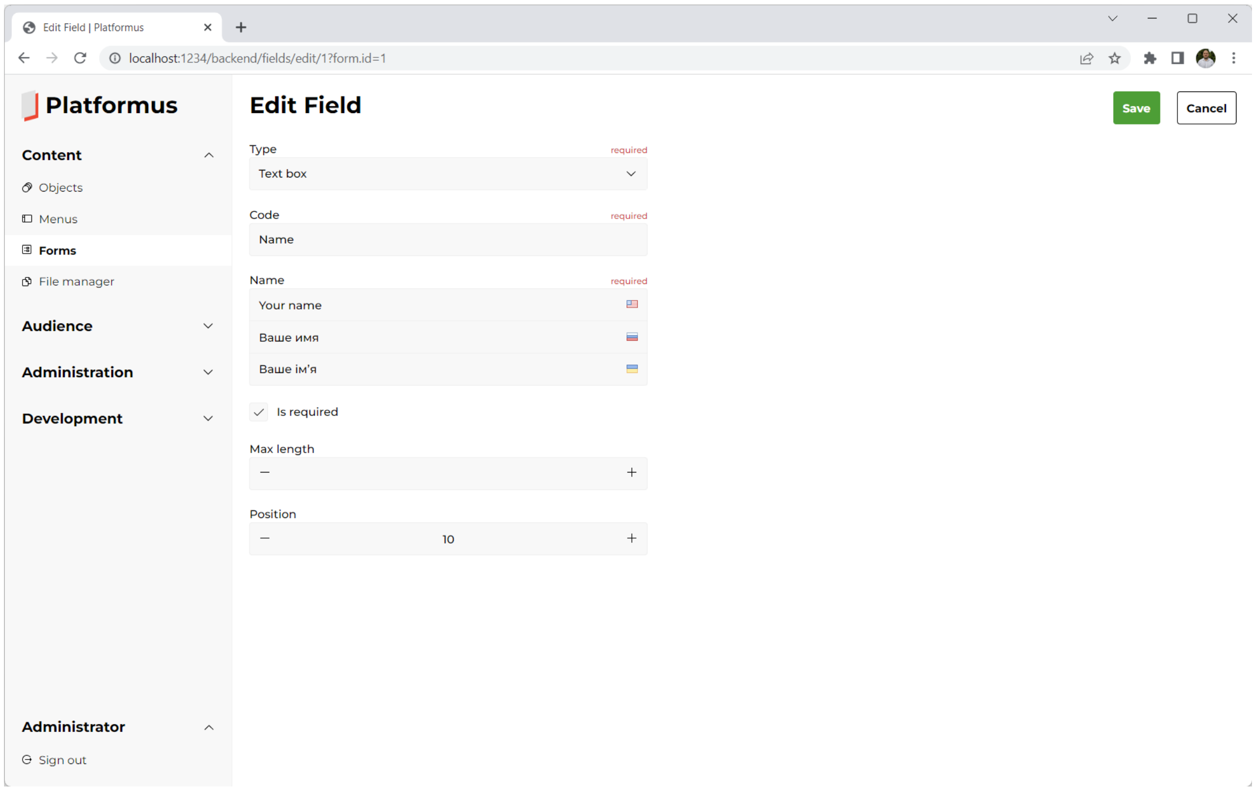
The *Produce completed forms* checkbox allows to specify if you want completed forms to be created each time user fills the form. You can review completed forms (user input) from the backend any time if they are created.

Form handler C# class name

Defines the implementation of the IFormHandler interface that will handle the user input for this form. There is the only one built-in implementation of this interface: the EmailFormHandler class. It sends the user input to the specified recipients by the email. You can write your own implementations of this interface. For example, you can have form handler that creates comments using the user input on a blog post page.

Each form handler can have different (specified by the developer) parameters, which use different parameter editors. Parameter editors might be created by the developer too. (The built-in form handler has two parameters: *Recipient emails* and *Redirect URL*.)

Forms consist of fields. There are different types of fields (and you can add your own ones). Each field has type, localized name, position, and other fields:



Edit Field

Type (required): Text box

Code (required): Name

Name (required):
Your name
Ваше имя
Ваше ім'я

✓ Is required

Max length:

Position: 10

Is required

Prevents user from submitting the form until this field contains value.

Max length

Prevents user from specifying a longer text value than allowed.

Position

`Position` might be used to sort the fields in the correct order within the form.

The fields of the type Drop down list also have user-defined options.

Once form is created, you can display it on the frontend using the built-in `FormViewComponent` view component like this (the form code is passed as the parameter to identify the form we want to display):

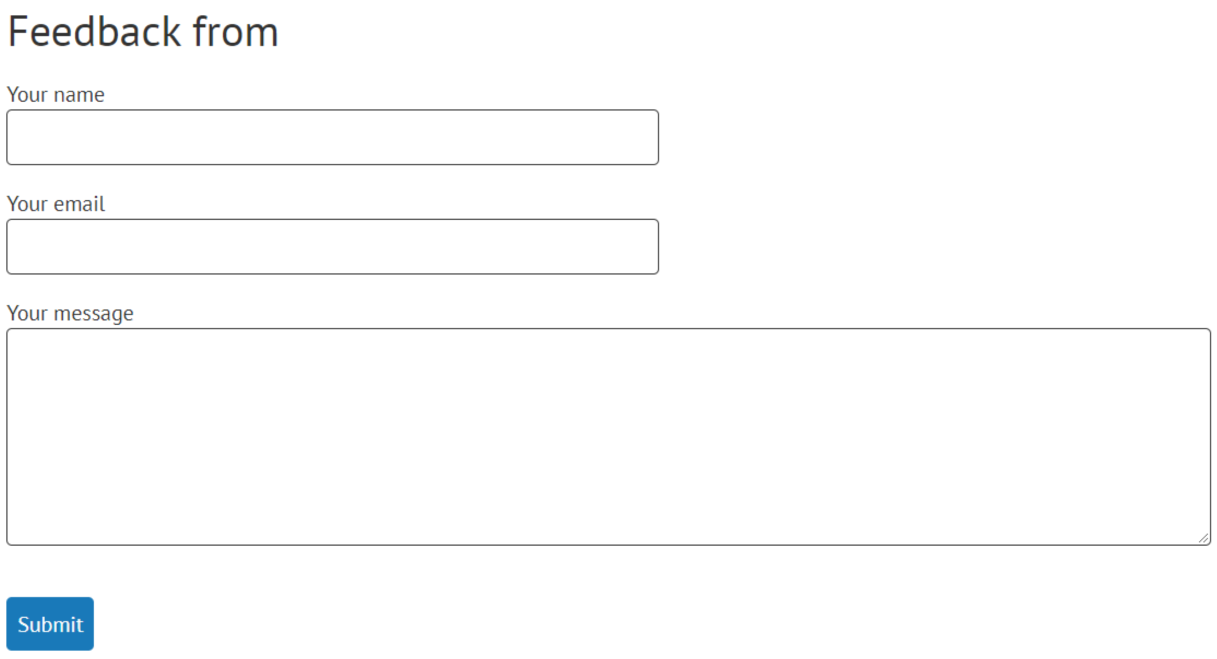
```
@await Component.InvokeAsync("Form", new { code = "Feedback", additionalCssClass =
↪"master-detail__form" })
```

Or using the view component tag helper:

```
<vc:form code="Feedback" additional-css-class="master-detail__form" />
```

As you can see, an additional CSS class might be applied using the corresponding optional parameter.

The result can look something like this:



Forms are displayed using the built-in views (_Form and _Field). The HTML elements have unique CSS classes (the BEM methodology is used), so it is easy to apply styles to them:

```
<form class="form" action="/en/forms/send" enctype="multipart/form-data" method="post
↪" novalidate="novalidate">
  <input name="formId" id="formId" type="hidden" value="1">
    <div class="form__field field">
      <label class="field__label label" for="field1">Your name</label>
      <input name="field1" class="field__text-box text-box" id="field1" type="text"␣
↪maxlength="64" data-val-required="" data-val-maxlength-max="64" data-val="true">
```

(continues on next page)

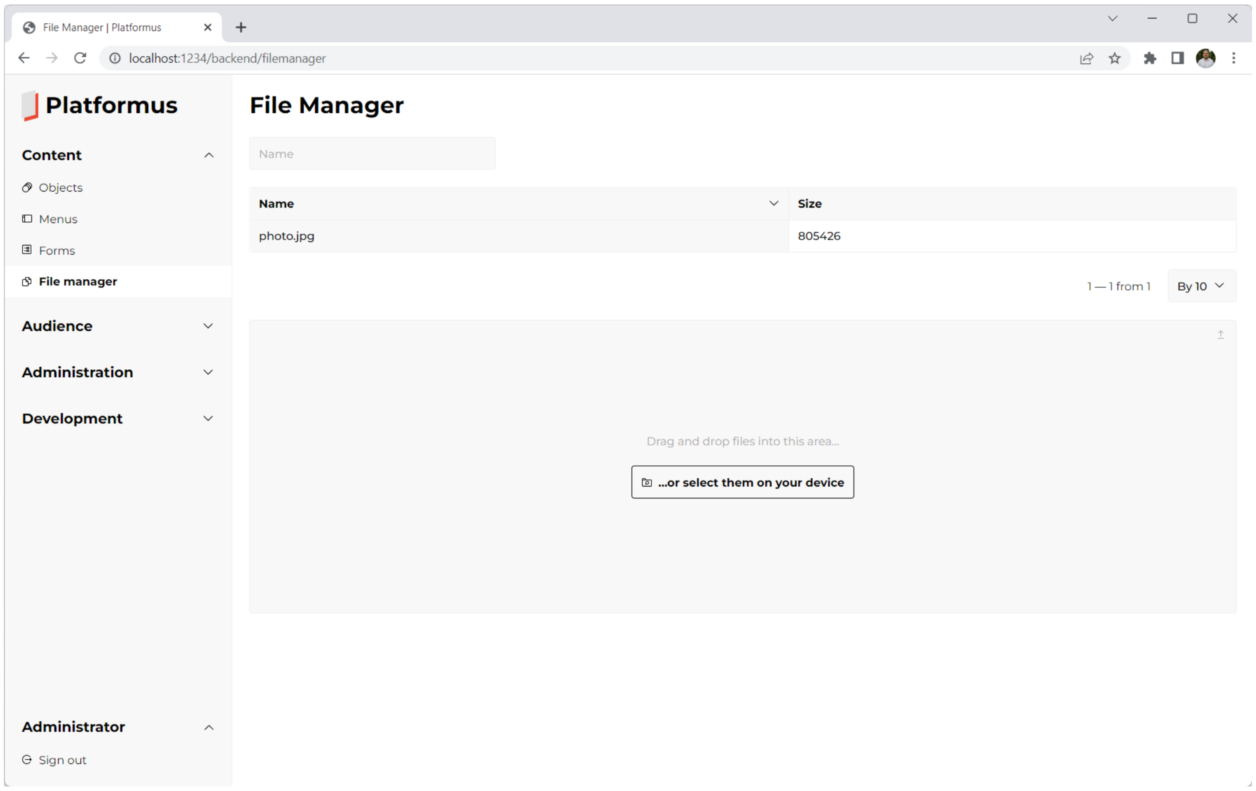
(continued from previous page)

```
    </div>
    <div class="form__field field">
      <label class="field__label label" for="field2">Your email</label>
      <input name="field2" class="field__text-box text-box" id="field2" type="text"␣
↪maxlength="64" data-val-required="" data-val-maxlength-max="64" data-val="true">
    </div>
    <div class="form__field field">
      <label class="field__label label" for="field3">Your message</label>
      <textarea name="field3" class="field__text-area text-area" id="field3"␣
↪maxlength="1024" data-val-required="" data-val-maxlength-max="1024" data-val="true">
↪</textarea>
    </div>
  <div class="form__buttons buttons">
    <button class="buttons__button button" type="submit">Send</button>
  </div>
</form>
```

If you want to change the HTML, just copy the views into your project and they will be used instead of the built-in ones, so you will be able to modify them as you want.

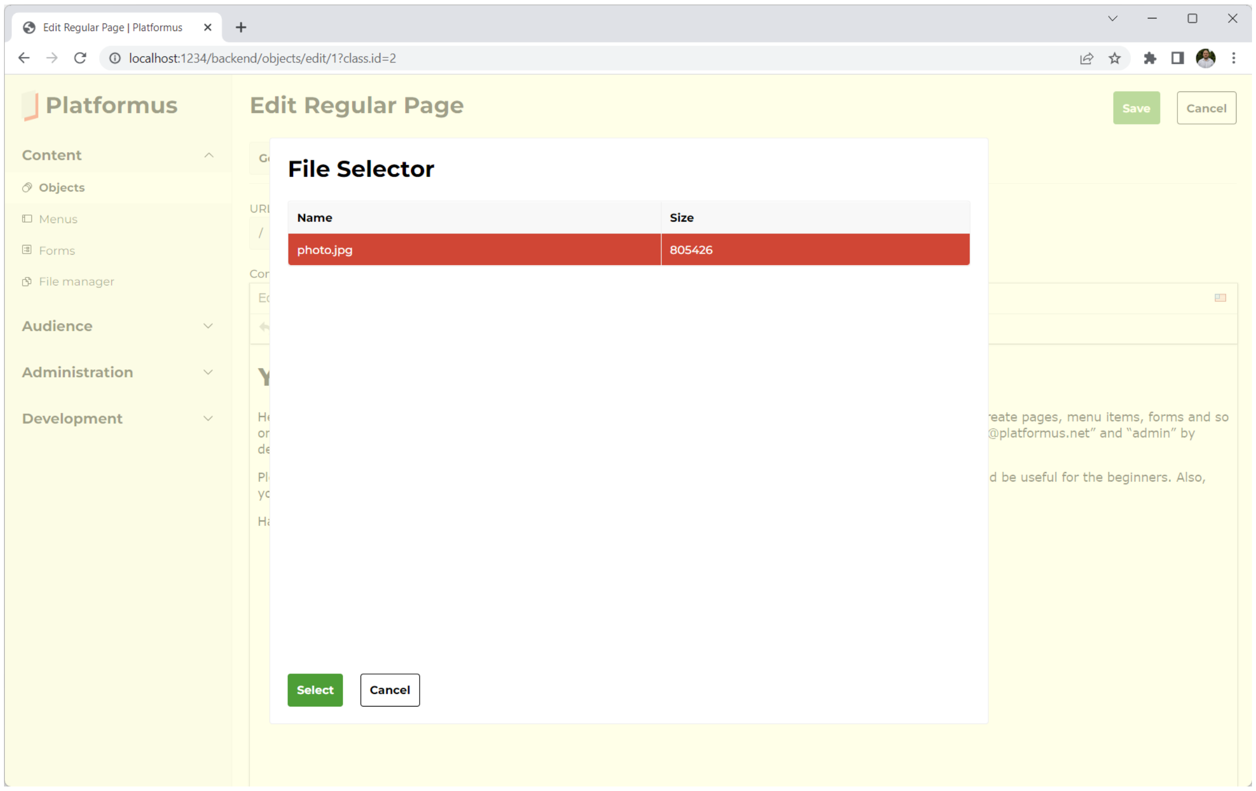
File Manager

File manager allows you to manage your files (upload and delete them) from the backend using the *Content/File manager* section:



Name	Size
photo.jpg	805426

Once a file is uploaded you can use it in different ways. You can copy a link to the file and paste it somewhere. If it is an image, you can paste it in the HTML editor using the file selector (click the *Insert/edit image* button and then the *Browse* one):

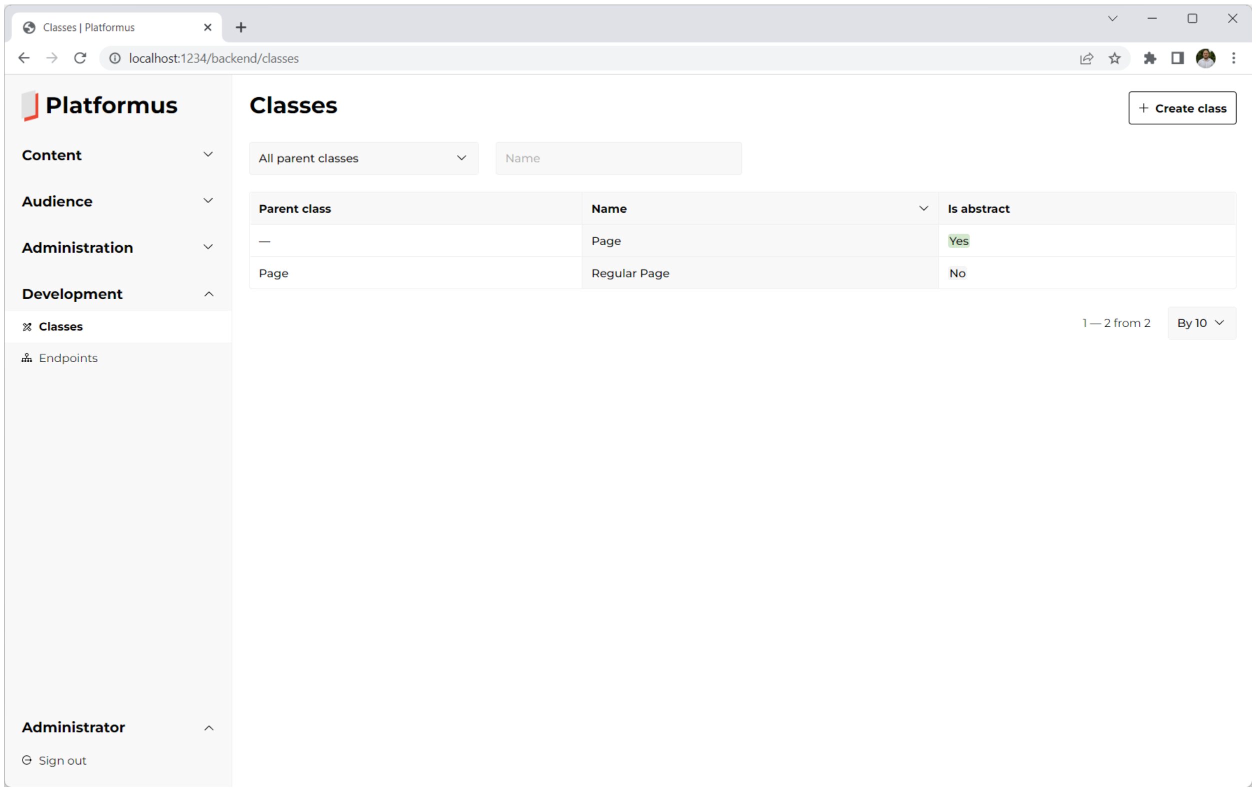


Name	Size
photo.jpg	805426

Please note that if your class contains a member with the `Image` as the data type, you don't need to upload an image using the file manager manually. The image editor will upload, crop, and save the image for the object property automatically (and the image path will be different: each object has its own images path that includes its identifier).

Classes

Classes are used to describe the *objects*. Using the *members*, they describe which properties should the objects have, how these property values should be stored and edited in the backend, and how the objects are related to each other. You can manage them (create, edit, and delete) from the backend using the *Development/Classes* section:

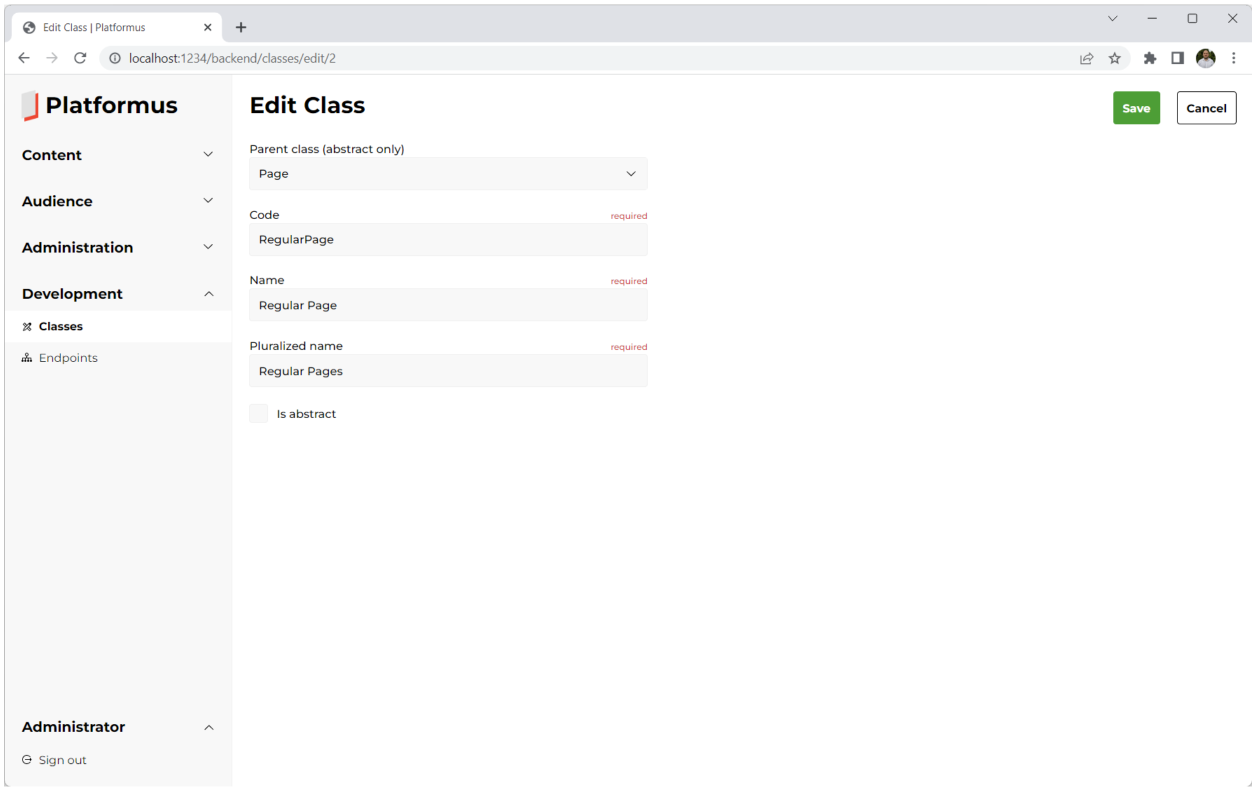


Classes

Parent class	Name	Is abstract
—	Page	Yes
Page	Regular Page	No

1 — 2 from 2 By 10

Each class has optional parent class, code, name, and other fields:



Edit Class

Parent class (abstract only)		Page
Code	required	RegularPage
Name	required	Regular Page
Pluralized name	required	Regular Pages
Is abstract		☐

Parent class (abstract only)

If you select a parent class, your class will have all the tabs and members of that abstract class. This feature helps to avoid copying the same members again and again. For example, it is good idea to have the `Page` abstract class with such members, like `URL`, `Content`, `Title`, and `META` tags. Also, abstract classes are used to group the objects in the *Content/Objects* section.

Code

It is the unique text identifier of the class.

Name

Name is used to identify the classes in the backend.

Pluralized name

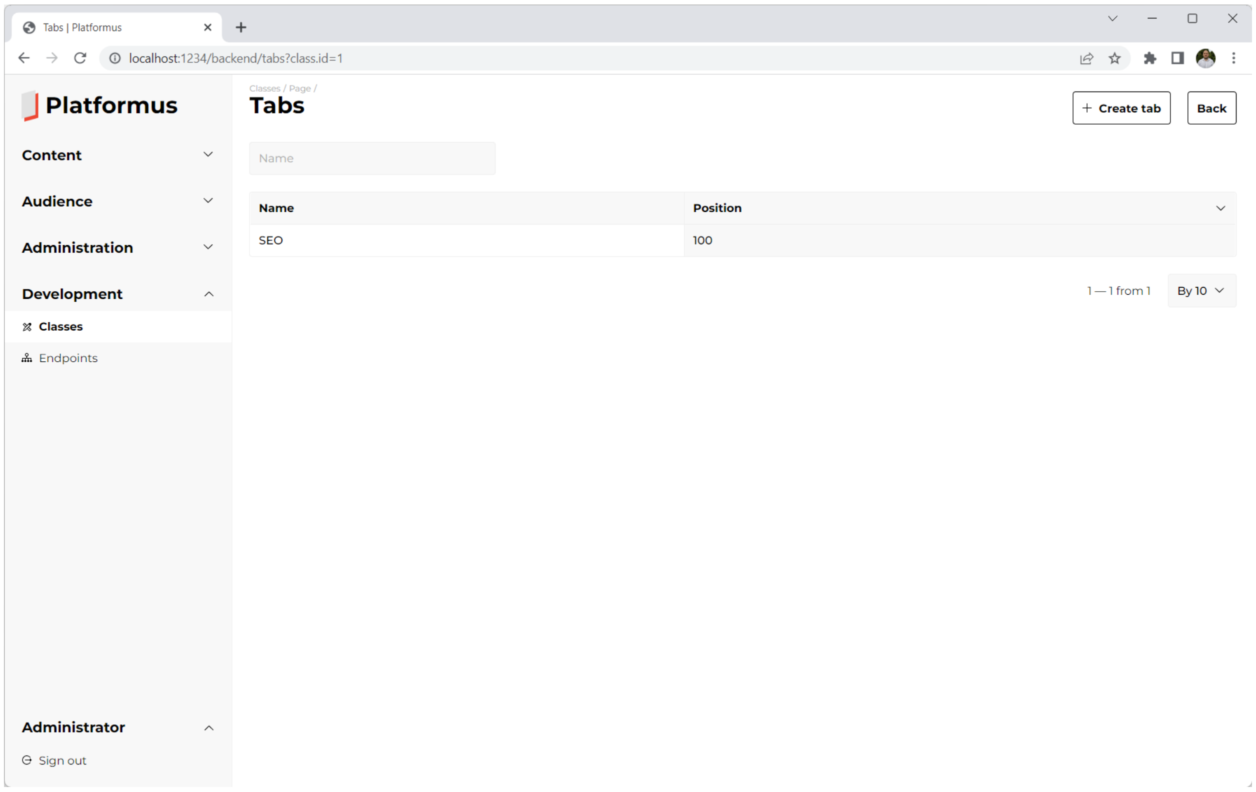
Pluralized name is used in the *Content/Objects* section.

Is abstract

Specifies whether the class is abstract or not.

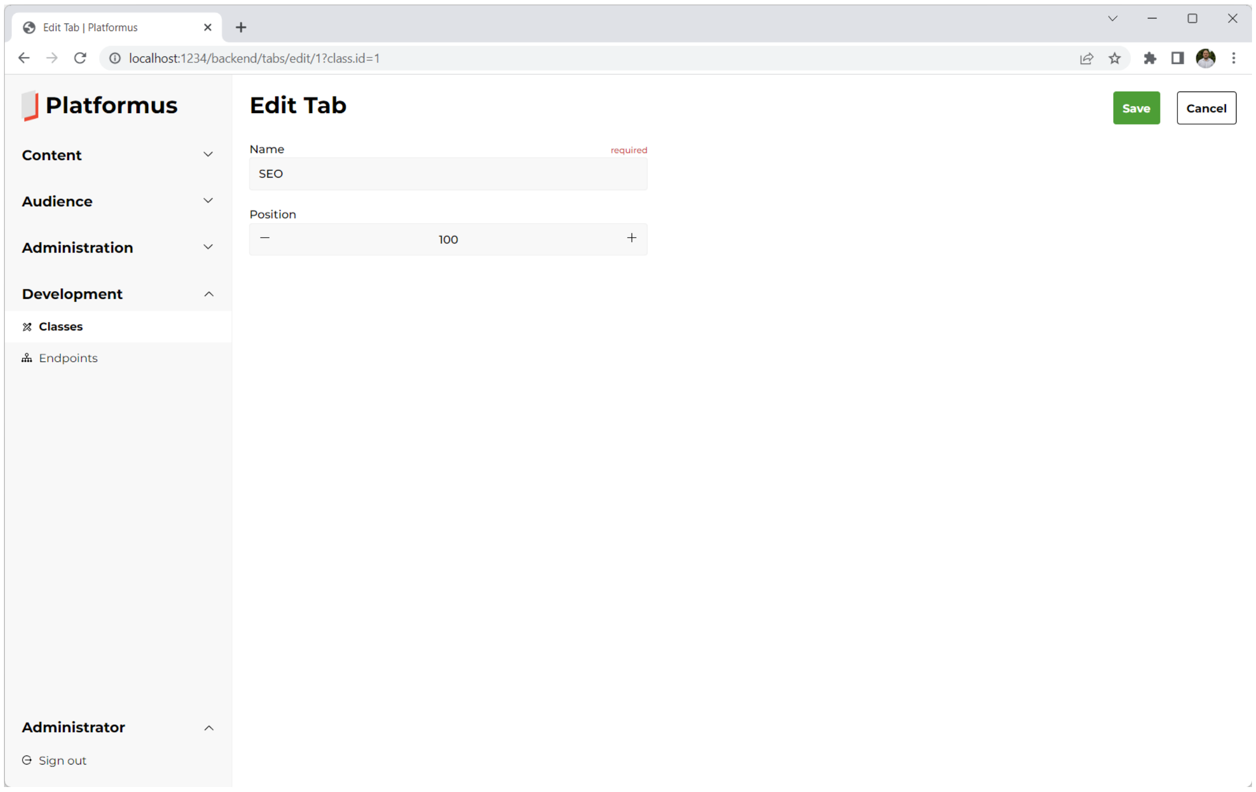
Tabs

Each class can have tabs. Tabs are used to group the object property editors on the create or edit object pages. You can manage them (create, edit, and delete) from the backend using the *Tabs* section of the *classes*:



Name	Position
SEO	100

Each tab has name and position:



Name: SEO

Position: 100

Name

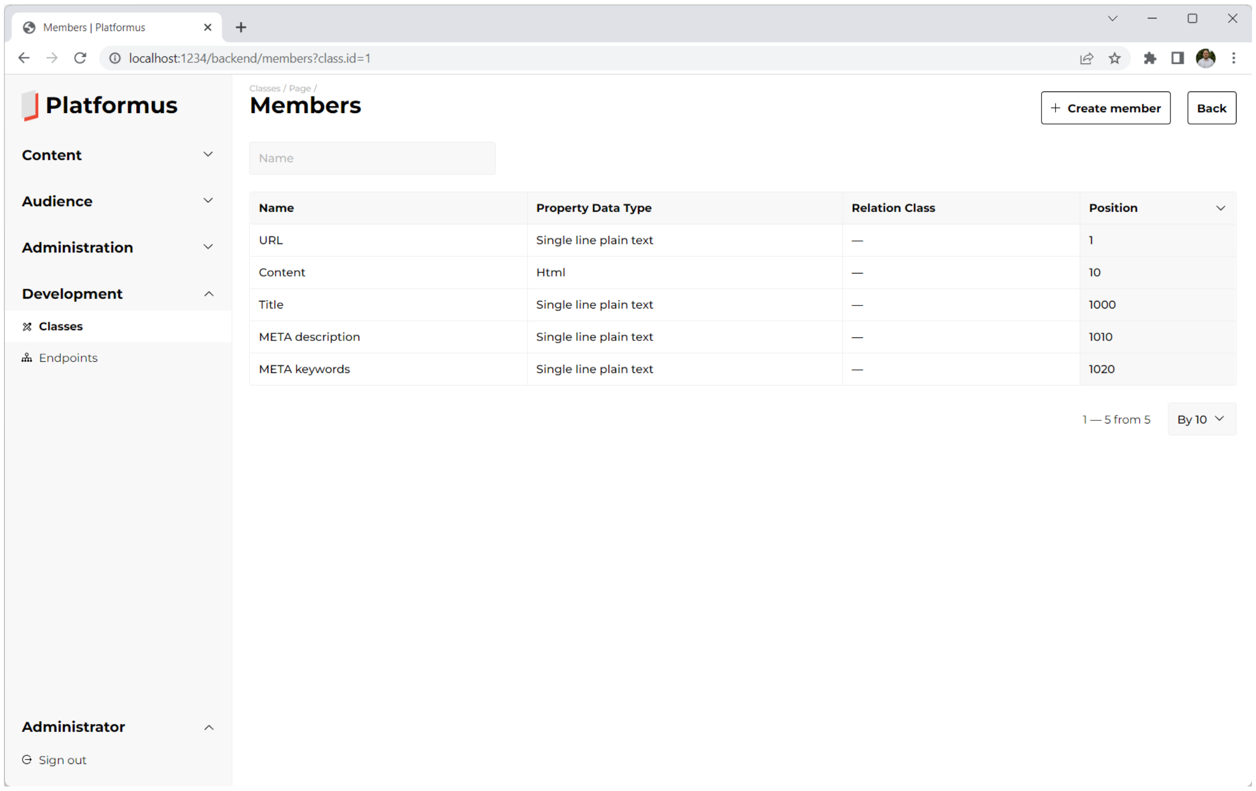
Name is used to identify the tabs in the backend.

Position

Position is used to sort the tabs in the lists.

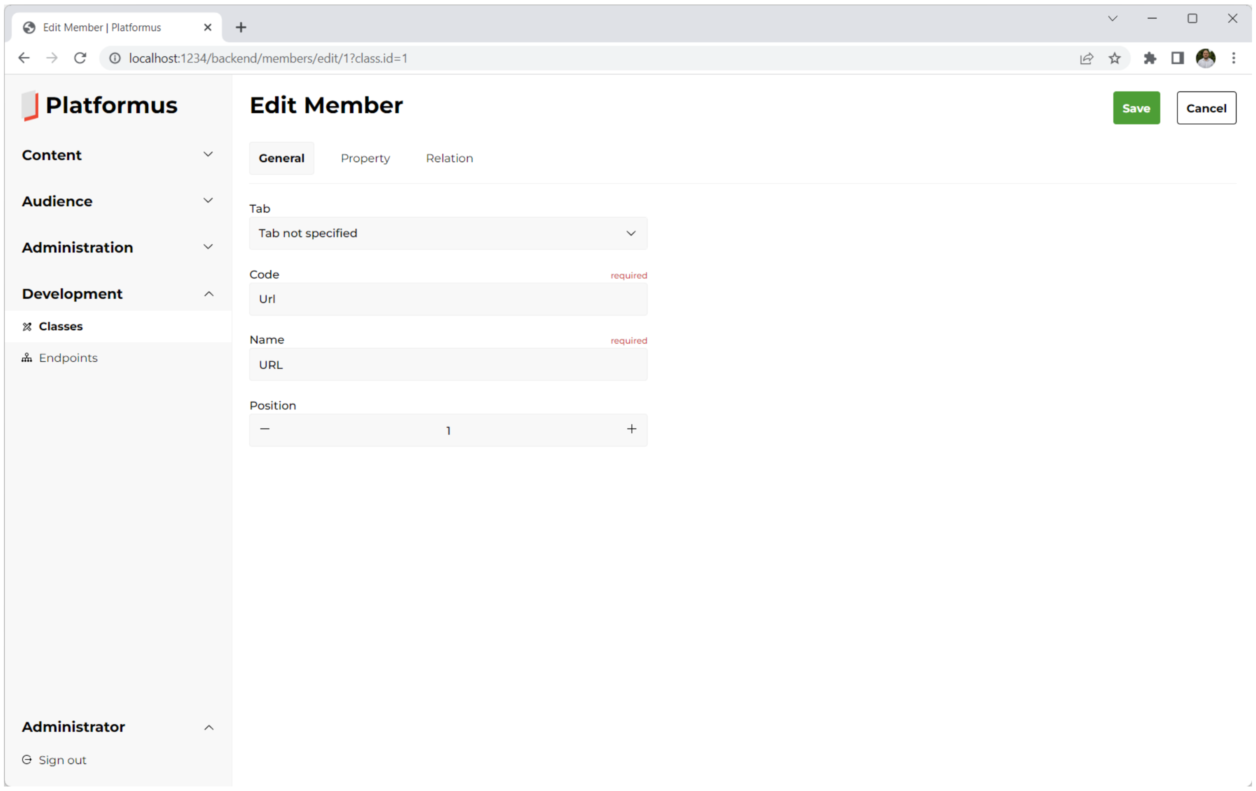
Members

Members are used to describe which properties and relations should the objects of a given class have. You can manage them (create, edit, and delete) from the backend using the *Members* section of the *classes*:



Name	Property Data Type	Relation Class	Position
URL	Single line plain text	—	1
Content	Html	—	10
Title	Single line plain text	—	1000
META description	Single line plain text	—	1010
META keywords	Single line plain text	—	1020

Each member has tab, code, name, and other fields:



Tab: Tab not specified

Code (required): Url

Name (required): URL

Position: 1

General/Tab

You can select a tab this member properties should belong to. All the properties of the members without a tab selected go under the *General* one.

General/Code

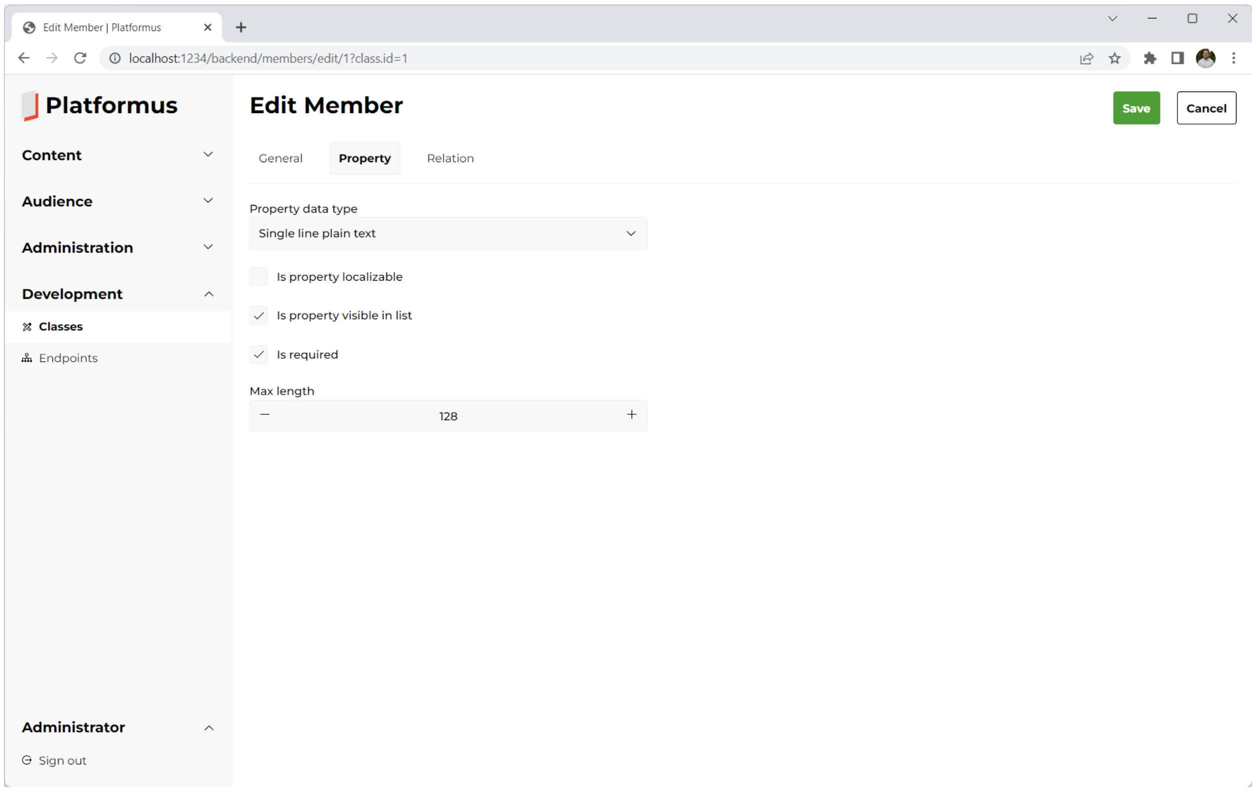
It is the unique text identifier of the member. This code is used in the different places, like sorting, mapping etc.

General/Name

Name is used to identify the members in the backend.

General/Position

Position is used to sort the members in the lists.



Property data type: Single line plain text

Is property localizable

✓ Is property visible in list

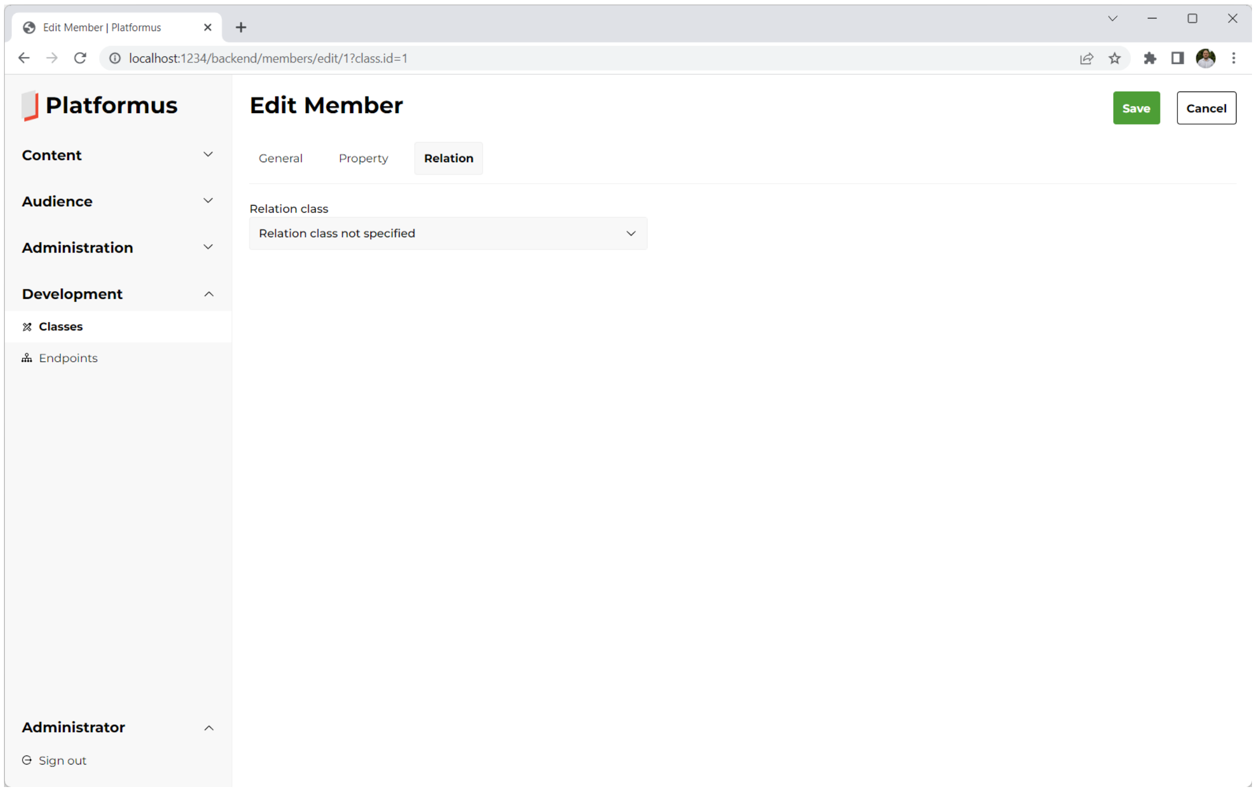
✓ Is required

Max length: 128

Property/Property data type

Member can be a property or a relation. If you specify the property data type, member will be considered as a property. Property data type allows to specify how to store the object property value (and which raw storage data type is used for that), how to display and edit it in the backend.

If a property data type is selected, data type parameters will be also displayed. Each data type can have unique parameters. For example, it could be maximum text length for the text or width and height for the image.



Relation class

Relation class not specified

Relation/Relation class

If you specify the relation *class*, member will be considered as a relation. Relation selector will be displayed as the editor. For example, user will be able to select a category for a blog post or assign tags to it. Also, if any relation class is selected, additional fields will be displayed.

Is relation single parent

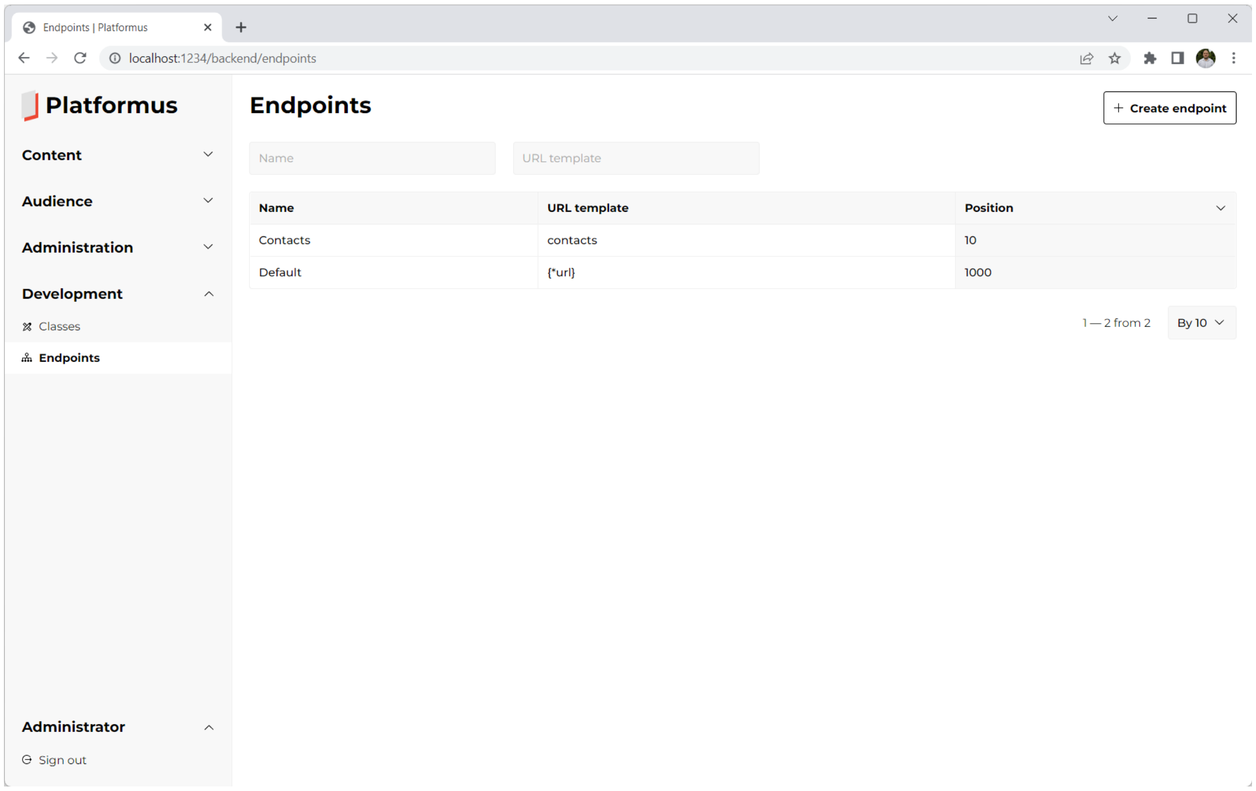
Specifies whether objects of this class can have the only one relation to the specified class (and this specified class will be considered as the parent one for the current class). For example, if blog post page can have the only one category page, you can use this option. In the object list, link to the blog post pages will appear in the category page rows, so all the created blog post pages will be automatically related to the parent category page objects. If the checkbox is not set, there will be two separated lists of the objects: `Category pages` and `Post pages`, and you will have to select a category page in every post page from the relation selector manually.

Relation/Min related objects number and *Relation/Max related objects number*

These fields allow to limit the number of the related objects. For example, you can specify that there should be 3-5 tags on every blog post page, so user will not be able to create a blog post page without the tags, or to specify more than 5.

Endpoints

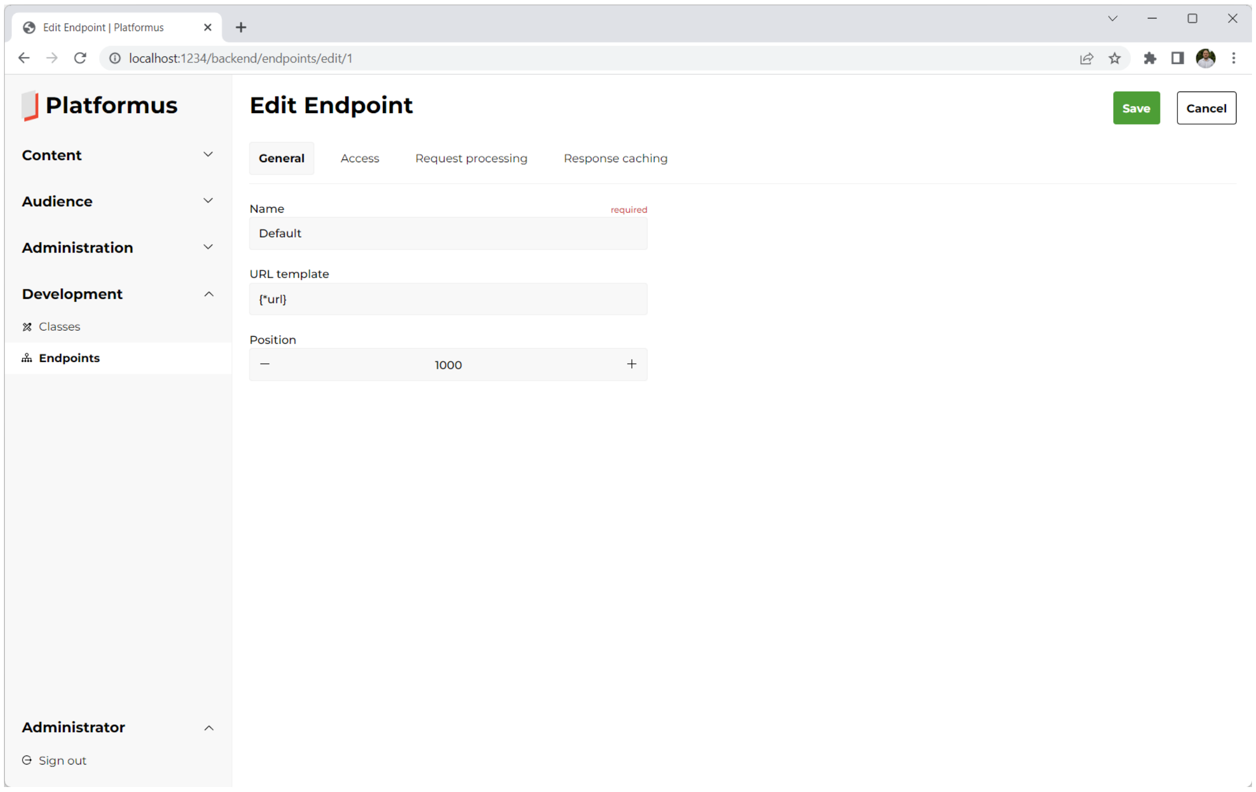
The purpose of the endpoints is to take some data provided by the *data sources*, represent it in some way (HTML, JSON, XML etc.), and return as the response. They also handle access control and caching. You can manage them (create, edit, and delete) from the backend using the *Development/Endpoints* section:



Name	URL template	Position
Contacts	contacts	10
Default	{*url}	1000

Please note, that the default ASP.NET routing still works and it is executed before any endpoint.

Each endpoint has name, URL template, position, and other fields:



Name required

Default

URL template

{*url}

Position

1000

General/Name

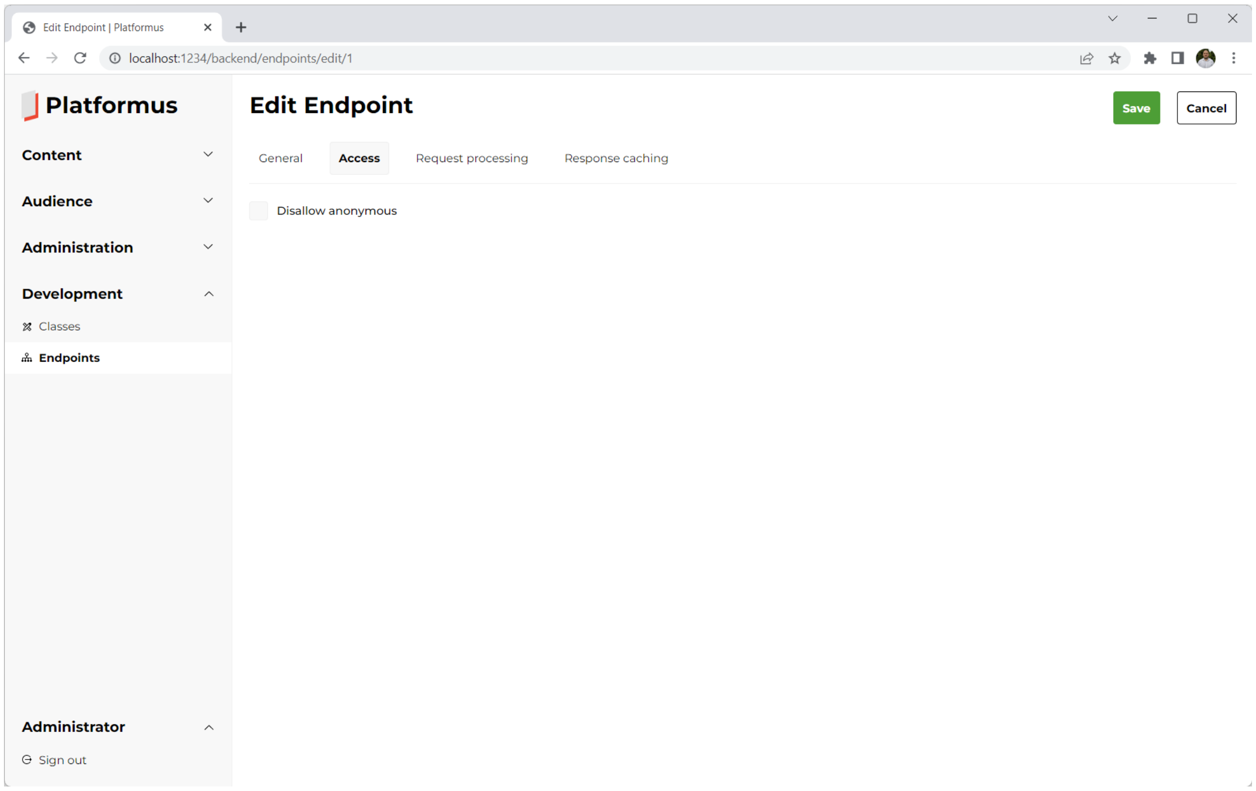
Name is used only in the endpoint list to identify endpoints.

General/URL template

URL template is used by the implementation of the IEndpointResolver interface to select the endpoint which should process current request (see below).

General/Position

Position is important, because the endpoint resolver checks the endpoints one by one, and it will return the first matching endpoint from the list, sorted by position.

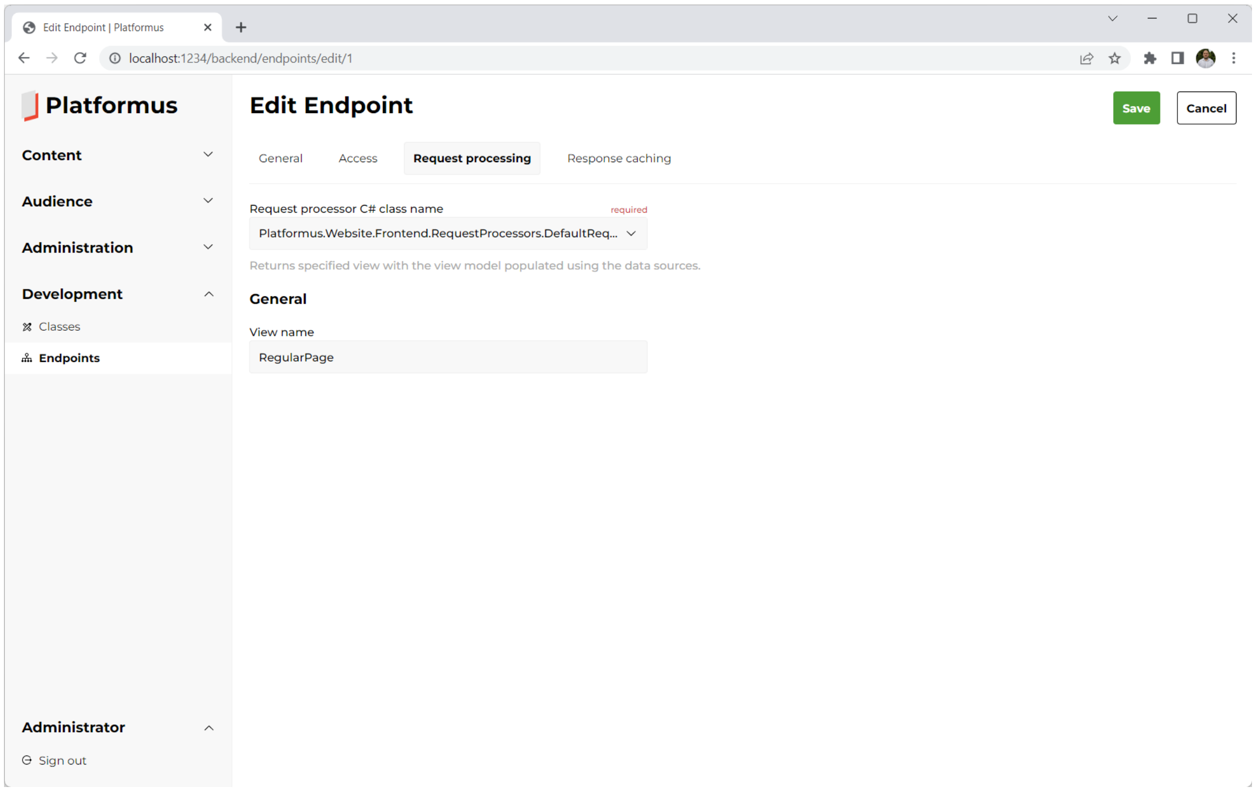


Access/Disallow anonymous

This checkbox allows you to specify whether a user must be authenticated in order to be able to get the response from this endpoint. Once the checkbox is checked, the *Required permissions* list will appear so you can specify which permissions the user must have.

Access/Sign in URL

This URL will be used to redirect user if he must be authenticated or if a required permission is missing.

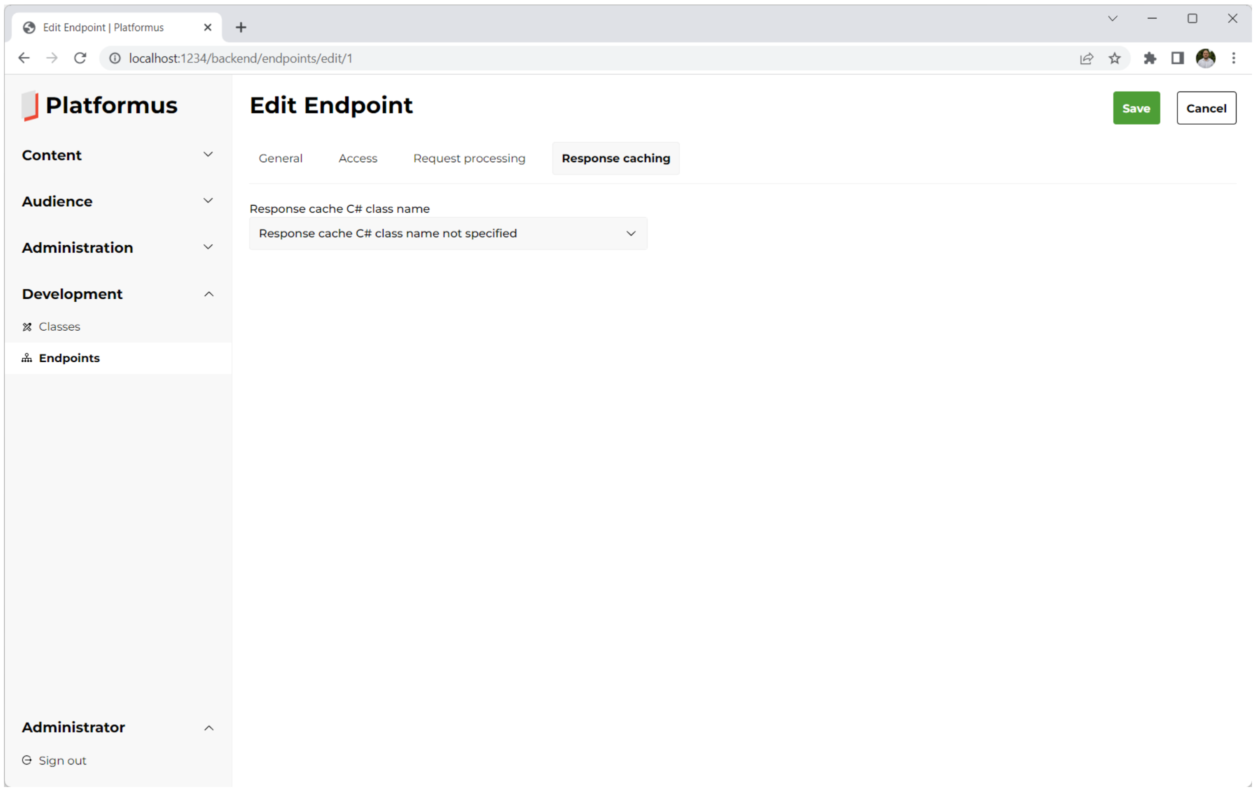


Request processor C# class name required
Platformus.Website.Frontend.RequestProcessors.DefaultReq...
Returns specified view with the view model populated using the data sources.
General
View name
RegularPage

Request processing/Request processor C# class name

It allows you to specify, which C# class (implementation of the IRequestProcessor interface) will process the requests (convert request data and data provided by the data sources to the response). It is very important, because you can write your own implementations of this interface. You can return HTML (using or not using views), JSON, files, plain text, redirects, or any other content. There is the only one built-in request processor: the DefaultRequestProcessor one. It returns views the same way as ASP.NET controllers do (you can specify the view name using the parameter).

Please note that the endpoints process requests using the request processors, they do not (and should not) provide data for the responses. In other words, they take prepared data and represent it in some way (HTML, JSON etc.). Data is provided to the endpoints by *data sources*. Each endpoint can have different data sources at the same time.



Response cache C# class name

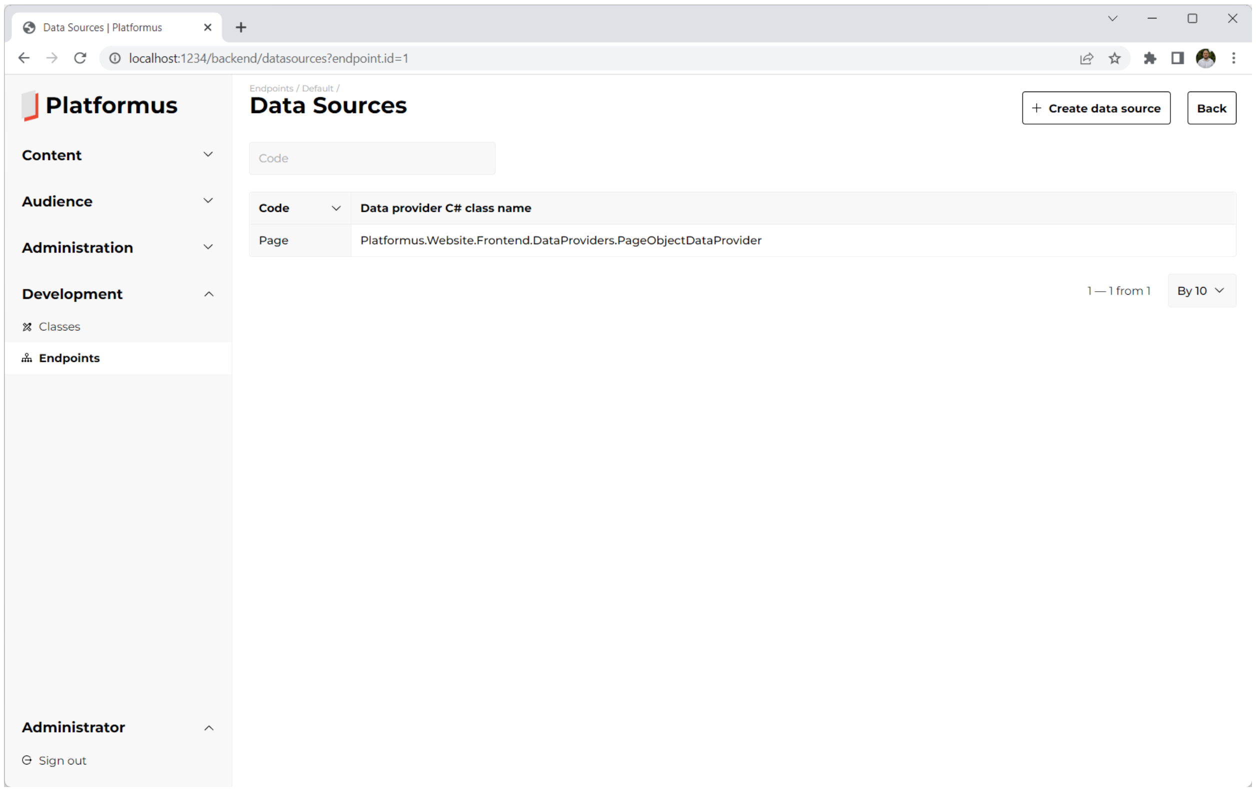
Response cache C# class name not specified

Response caching/Response cache C# class name

It allows you to specify, which C# class (implementation of the IResponseCache is used to cache the endpoint response. There are several built-in implementations of this interface, but you can write your own ones.

Data Sources

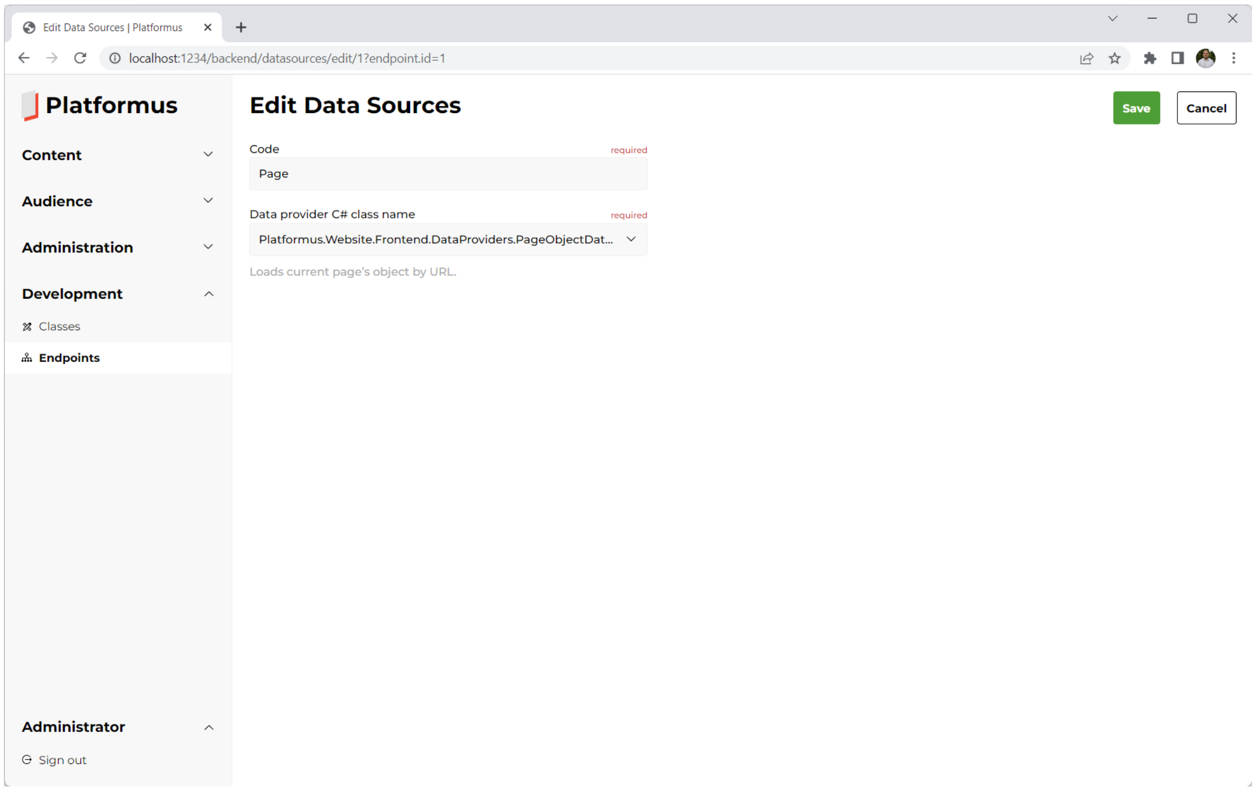
Data sources provide the *endpoints* with data (end then endpoints represent that data in some way and return as the responses). Data sources use the specified implementation of the IDataProvider interface to get, prepare, or generate data (it can be anything: the database records, weather forecast, some hardcoded text). It can be useful because a developer can prepare different data providers that can be then used and combined by a user to provide all the required data for, for example, the specific website pages. You can manage them (create, edit, and delete) from the backend using the *Data sources* section of the *endpoints*:



Data Sources

Code	Data provider C# class name
Page	Platformus.Website.Frontend.DataProviders.PageObjectDataProvider

Each data source has code and data provider C# class name:



Edit Data Sources

Code required: Page

Data provider C# class name required: Platformus.Website.Frontend.DataProviders.PageObjectDat...

Loads current page's object by URL.

Code

Code is used as the model property name. So, if you have specified some code, your model (for example, a view model) will have a property with this name.

Data provider C# class

It allows you to specify, which C# class (implementation of the IDataProvider interface) will provide data for this data source.

There are few built-in data providers. PageObjectDataProvider loads the object by its `URL` property value. ObjectsDataProvider loads the objects of the specified class. RelatedObjectsDataProvider loads the objects that are related to the one, which `URL` property value matches current request's URL.

Advanced

Custom Form Handlers

The *Platformus.Website* extension offers the great *forms features*. You can describe your forms in the backend, and then render them and get user feedback on any frontend view. When user fills the form and clicks the *Submit* button, data is sent to a server and might be processed in any way you want. User input is handled by the implementations of the IFormHandler interface which can return any `IActionResult` as the result.

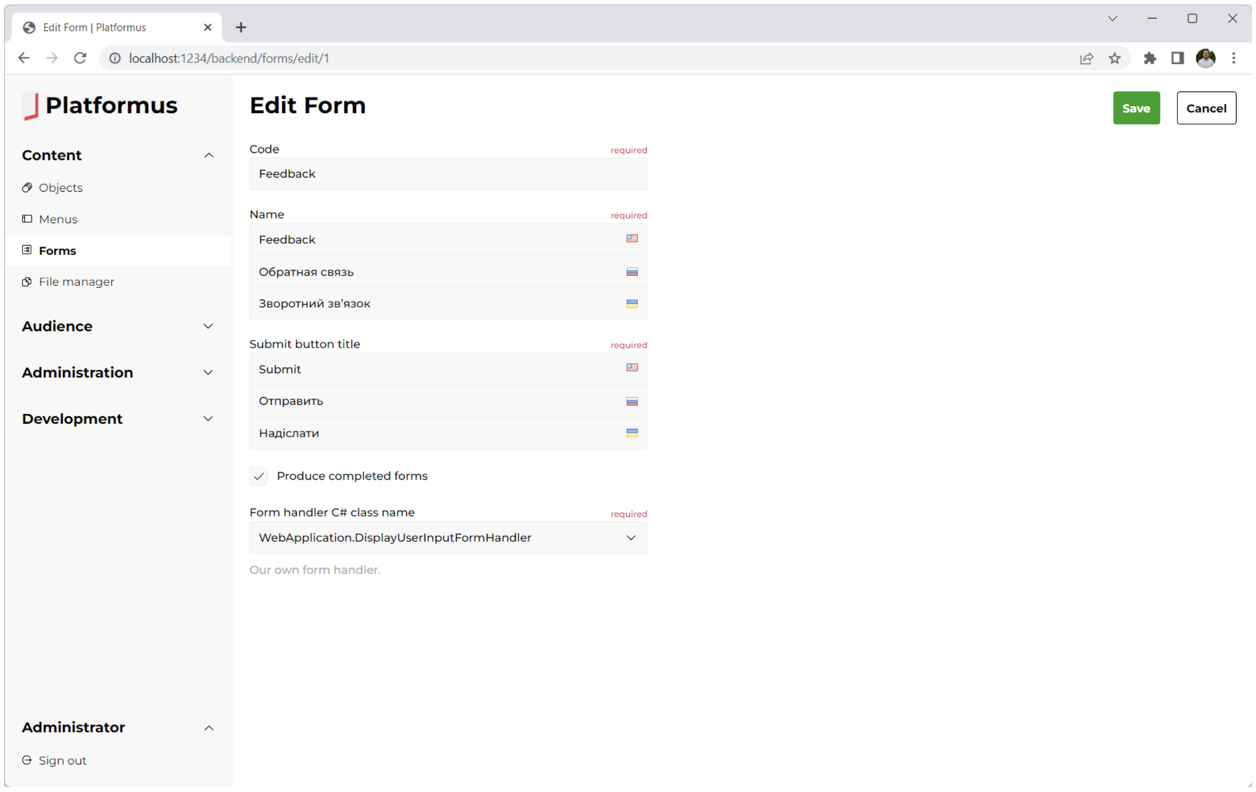
The selected implementation receives the form object, all the user input (string values by field objects), and all the attachments user has uploaded. Field values can be validated using the implementations of the IFieldValidator interface (you can use the ReCaptchaFieldValidator as an example).

Platformus has the only one built-in implementation of the `IFormHandler` interface: the EmailFormHandler class. It sends the user input to the specified email recipients.

Let's implement our own form handler, which will just display the user input (but it could do anything else as well). Create the `DisplayUserInputFormHandler` class inside the main web application project and implement the `IFormHandler` interface there:

```
public class DisplayUserInputFormHandler : IFormHandler
{
  public IEnumerable<ParameterGroup> ParameterGroups => new ParameterGroup[] { };
  public string Description => "Our own form handler.";

  public async Task<IActionResult> HandleAsync(HttpContext httpContext, string origin,
↪ Form form, IDictionary<Field, string> valuesByFields, IDictionary<string, byte[]>␣
↪attachmentsByFilenames)
  {
    StringBuilder body = new StringBuilder();

    foreach (KeyValuePair<Field, string> valueByField in valuesByFields)
      body.AppendFormat("<p>{0}: {1}</p>", valueByField.Key.Name.
↪GetLocalizationValue(), valueByField.Value);

    return new ContentResult() { Content = body.ToString() };
  }
}
```

Now, when our form handler class is added, navigate to the backend's *Content/Forms* section and create or edit a form:



Edit Form

Code (required): Feedback

Name (required): Feedback / Обратная связь / Зворотний зв'язок

Submit button title (required): Submit / Отправить / Надіслати

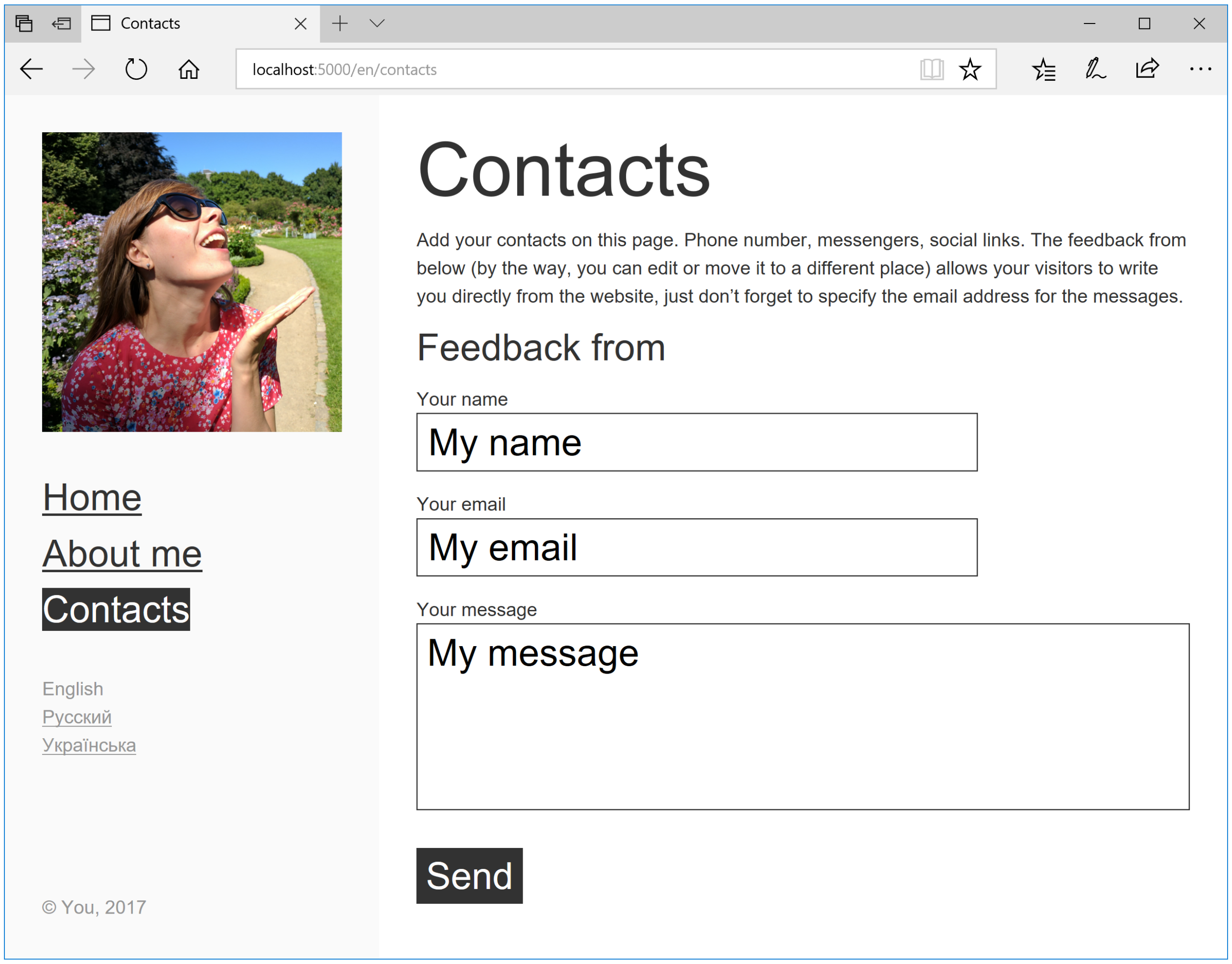
✓ Produce completed forms

Form handler C# class name (required): WebApplication.DisplayUserInputFormHandler

Our own form handler.

Please note, that our new form handler C# class is automatically resolved and added to the drop down list. Click the *Save* button.

Now navigate to /en/contacts and fill out the form:



Contacts

Add your contacts on this page. Phone number, messengers, social links. The feedback from below (by the way, you can edit or move it to a different place) allows your visitors to write you directly from the website, just don't forget to specify the email address for the messages.

Feedback from

Your name

My name

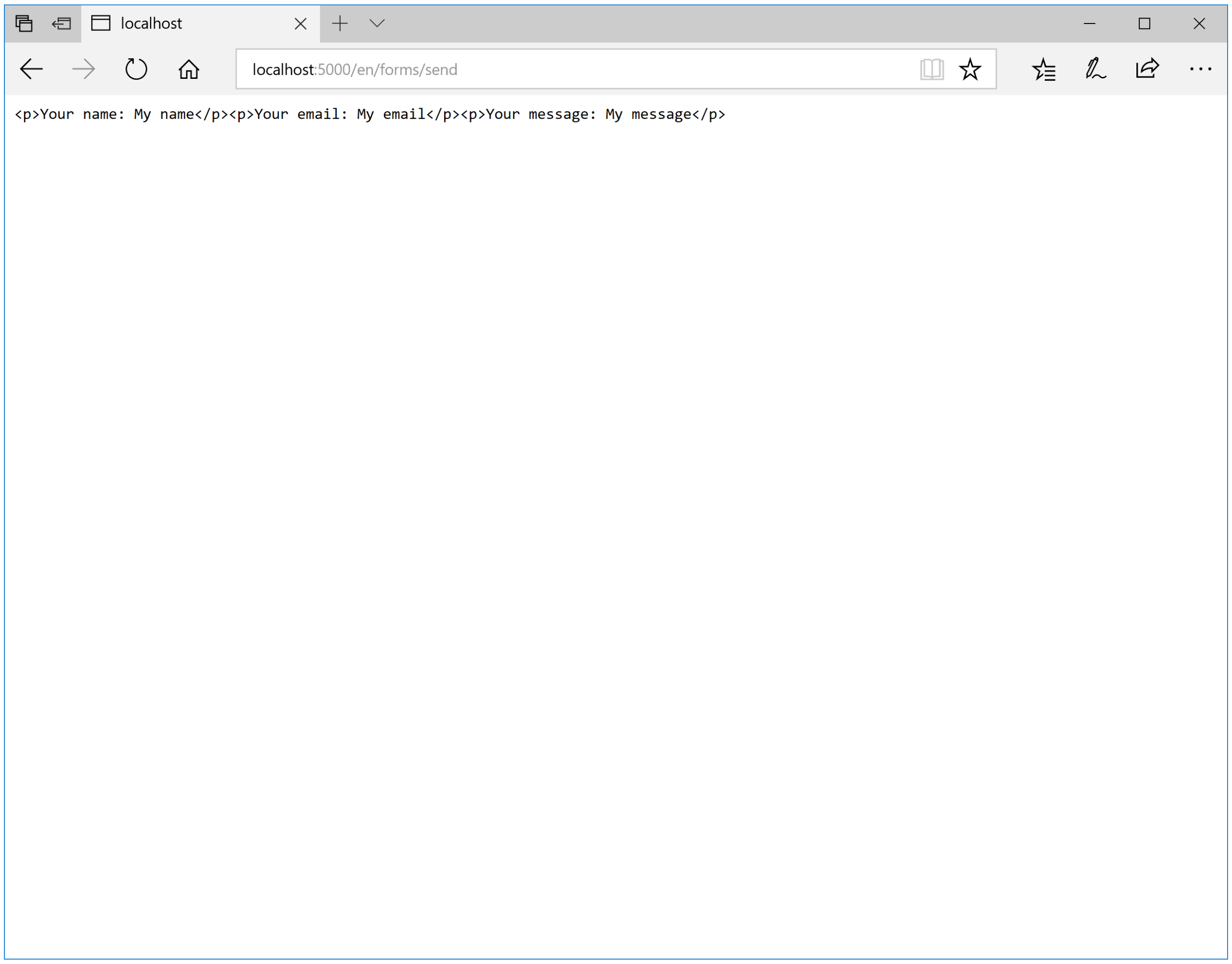
Your email

My email

Your message

My message

Click the *Send* button. Output from our form handler is displayed:



```
<p>Your name: My name</p><p>Your email: My email</p><p>Your message: My message</p>
```

We could return some view, redirect, or any other action result we need.

Form Handler Parameters and Parameter Groups

As well as the endpoints and data sources, form handlers support parameters and parameter groups. The implementation is absolutely the same, so please just take a look at the *endpoints* for the sample. Also, please take a look at the built-in form handler to see how it gets the parameter value.

Custom Data Sources

As we know, data providers are used by *data sources* to provide *endpoints* with data. It is a regular C# class that implements the IDataProvider interface. There are several built-in data providers, but all of them work with the *objects*. Let's create a custom one that will just provide a hardcoded text.

Create the `MyDataProvider` class inside the main web application project and implement the `IDataProvider` interface:

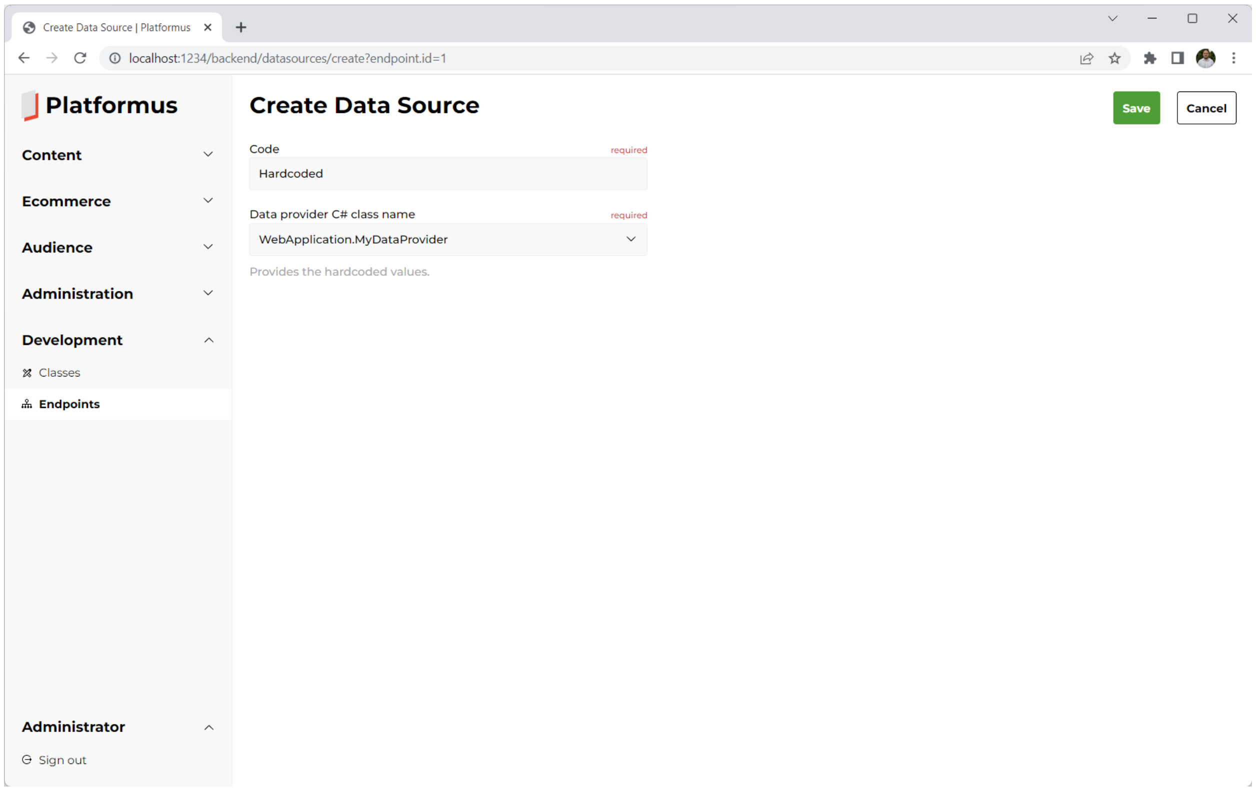
```
public class MyDataProvider : IDataProvider
{
  public string Description => "Provides the hardcoded values.";

```

(continues on next page)

(continued from previous page)

```
  public IEnumerable<ParameterGroup> ParameterGroups => new ParameterGroup[] { };

  public async Task<dynamic> GetDataAsync(HttpContext httpContext, DataSource␣
↪dataSource)
  {
    ExpandoObjectBuilder expandoObjectBuilder = new ExpandoObjectBuilder();

    expandoObjectBuilder.AddProperty("Value", "Some hardcoded value from our custom␣
↪data source.");
    return expandoObjectBuilder.Build();
  }
}
```

Now, when our data provider class is added, navigate to the backend's *Development/Endpoints* section, then to the data source list of the Default endpoint, and create one more data source:

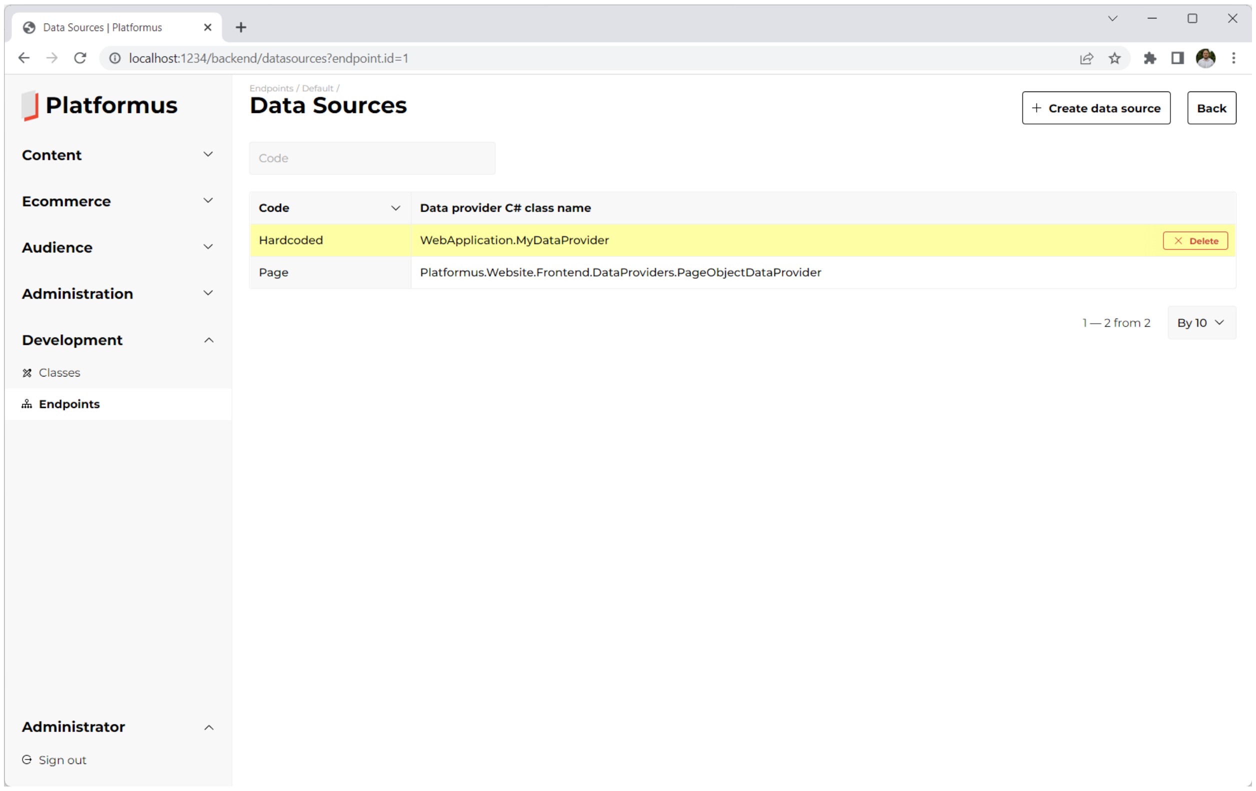


Code: Hardcoded

Data provider C# class name: WebApplication.MyDataProvider

Provides the hardcoded values.

Please note, that our new data provider C# class is automatically resolved and added to the drop down list. Click the *Save* button. Data source is created:



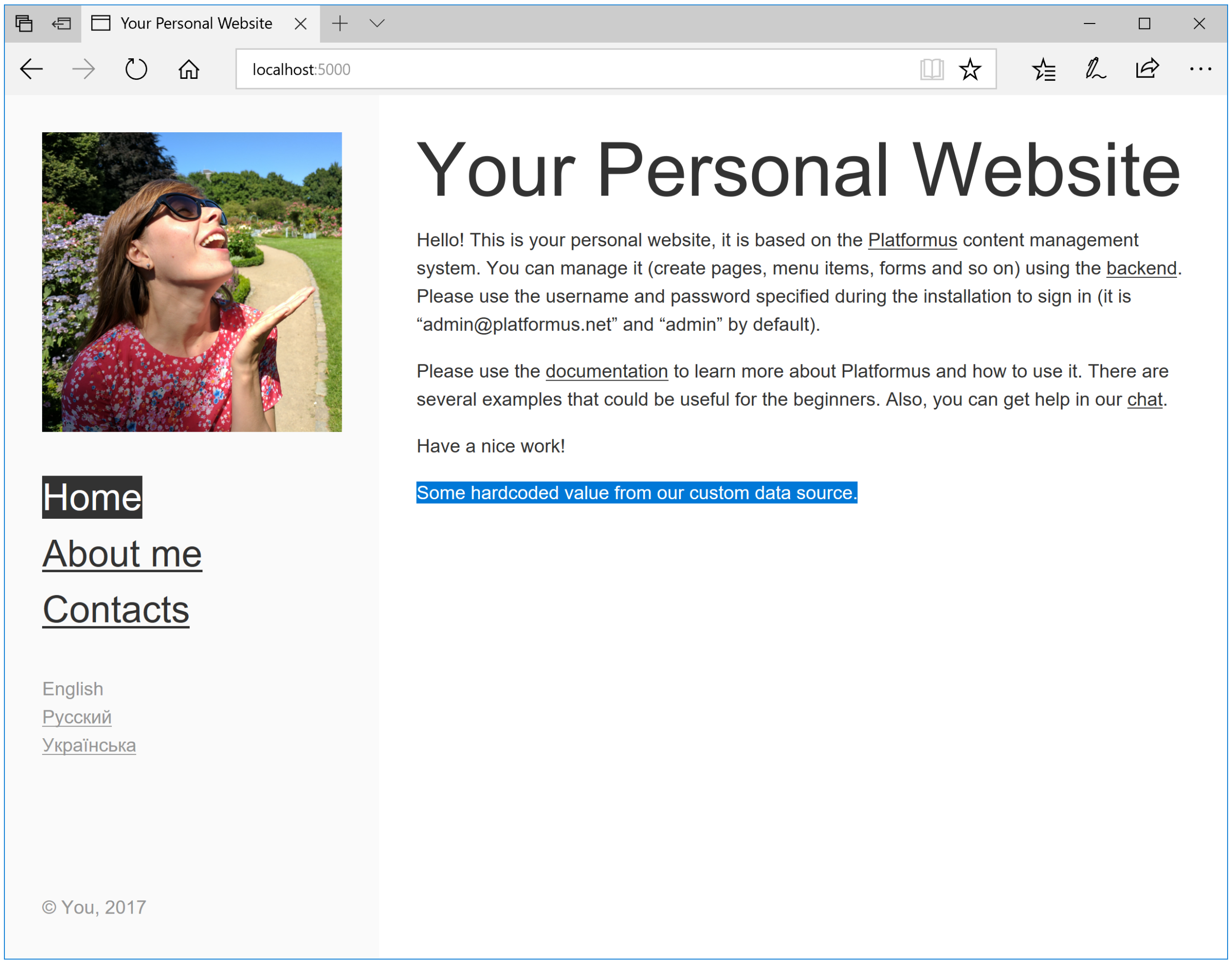
Code	Data provider C# class name
Hardcoded	WebApplication.MyDataProvider
Page	Platformus.Website.Frontend.DataProviders.PageObjectDataProvider

All the regular pages now have the `Hardcoded` property inside their view models (thanks to the DefaultRequestProcessor request processor; your own request processor might act in a different way to process provided data).

Let’s update the RegularPage.cshtml view to display the value from our new data source:

```
<p>@Model.Hardcoded.Value</p>
```

Run the web application and check the output:



Your Personal Website

Hello! This is your personal website, it is based on the Platformus content management system. You can manage it (create pages, menu items, forms and so on) using the backend. Please use the username and password specified during the installation to sign in (it is “admin@platformus.net” and “admin” by default).

Please use the documentation to learn more about Platformus and how to use it. There are several examples that could be useful for the beginners. Also, you can get help in our chat.

Have a nice work!

Some hardcoded value from our custom data source.

Good. Everything works as expected.

Data Source Parameters and Parameter Groups

As well as the form handlers and endpoints, data providers support parameters and parameter groups. The implementation is absolutely the same, so please just take a look at the *endpoints* for the sample. Also, please take a look at the built-in data providers to see how they get the parameter values.

Custom Endpoints

As we know, *endpoints* process the requests and prepare responses using the data provided by the *data sources*. Endpoints use the implementation of the IRequestProcessor interface for that. It can return any `IActionResult`. The idea is that by changing the request processor class you can represent data in a different way. That’s why the request processor shouldn’t provide data itself but use the one provided with data sources.

The only one built-in implementation of the IRequestProcessor interface (the DefaultRequestProcessor class) returns Razor views with the view model combined from the data provided by the data sources. Let’s create our own request processor which will return JSON instead:

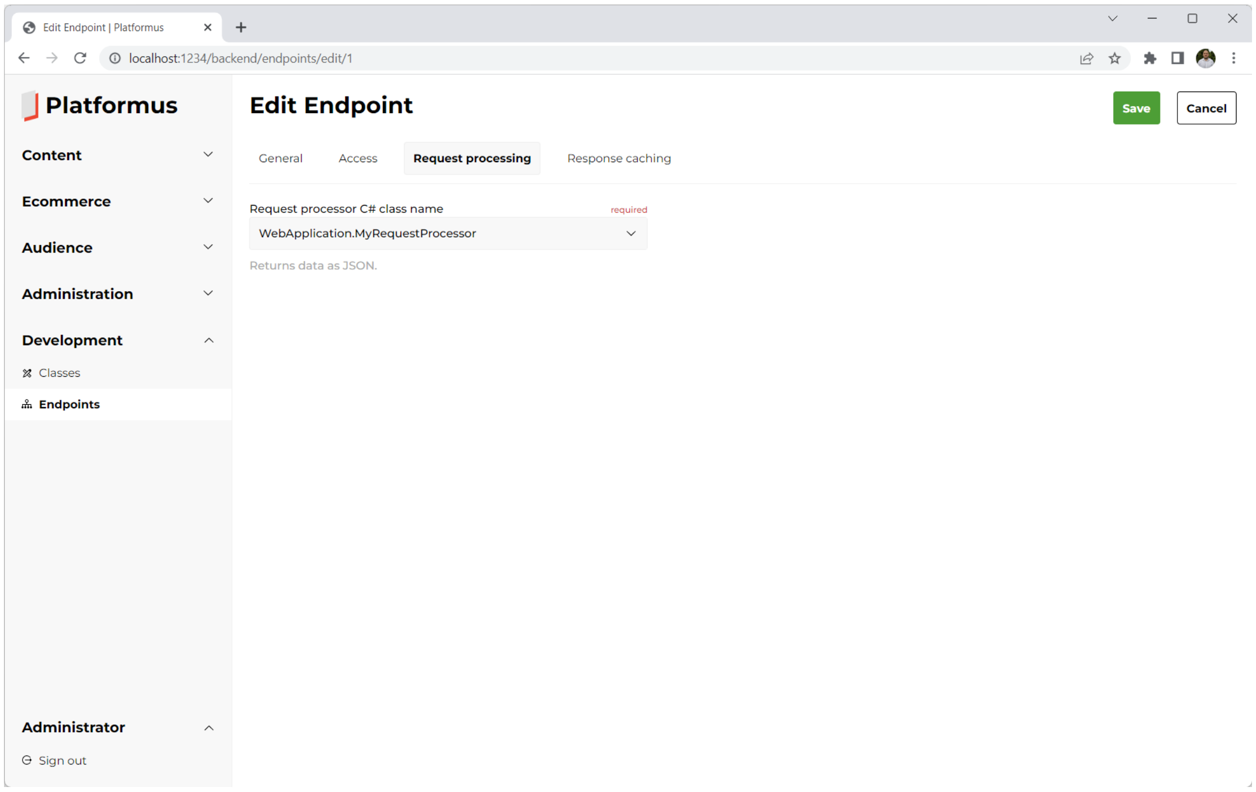
```
public class MyRequestProcessor : IRequestProcessor
{
```

(continues on next page)

(continued from previous page)

```
  public IEnumerable<ParameterGroup> ParameterGroups => new ParameterGroup[] { };
  public string Description => "Returns data as JSON.";

  public async Task<IActionResult> ProcessAsync(HttpContext httpContext, Platformus.
↪Website.Data.Entities.Endpoint endpoint)
  {
    dynamic viewModel = await this.CreateViewModelAsync(httpContext, endpoint);

    if (viewModel == null)
      return null;

    return new JsonResult(viewModel);
  }

  private async Task<dynamic> CreateViewModelAsync(HttpContext httpContext,␣
↪Platformus.Website.Data.Entities.Endpoint endpoint)
  {
    ExpandoObjectBuilder expandoObjectBuilder = new ExpandoObjectBuilder();

    foreach (DataSource dataSource in endpoint.DataSources)
    {
      dynamic viewModel = await this.GetDataProvider(dataSource).
↪GetDataAsync(httpContext, dataSource);

      if (viewModel == null)
        return null;

      expandoObjectBuilder.AddProperty(dataSource.Code, viewModel);
    }

    return expandoObjectBuilder.Build();
  }

  private IDataProvider GetDataProvider(DataSource dataSource)
  {
    return StringActivator.CreateInstance<IDataProvider>(dataSource.
↪DataProviderCSharpClassName);
  }
}
```

Now, when our request processor class is added, navigate to the backend's *Development/Endpoints* section and change the default endpoint's request processor C# class name:

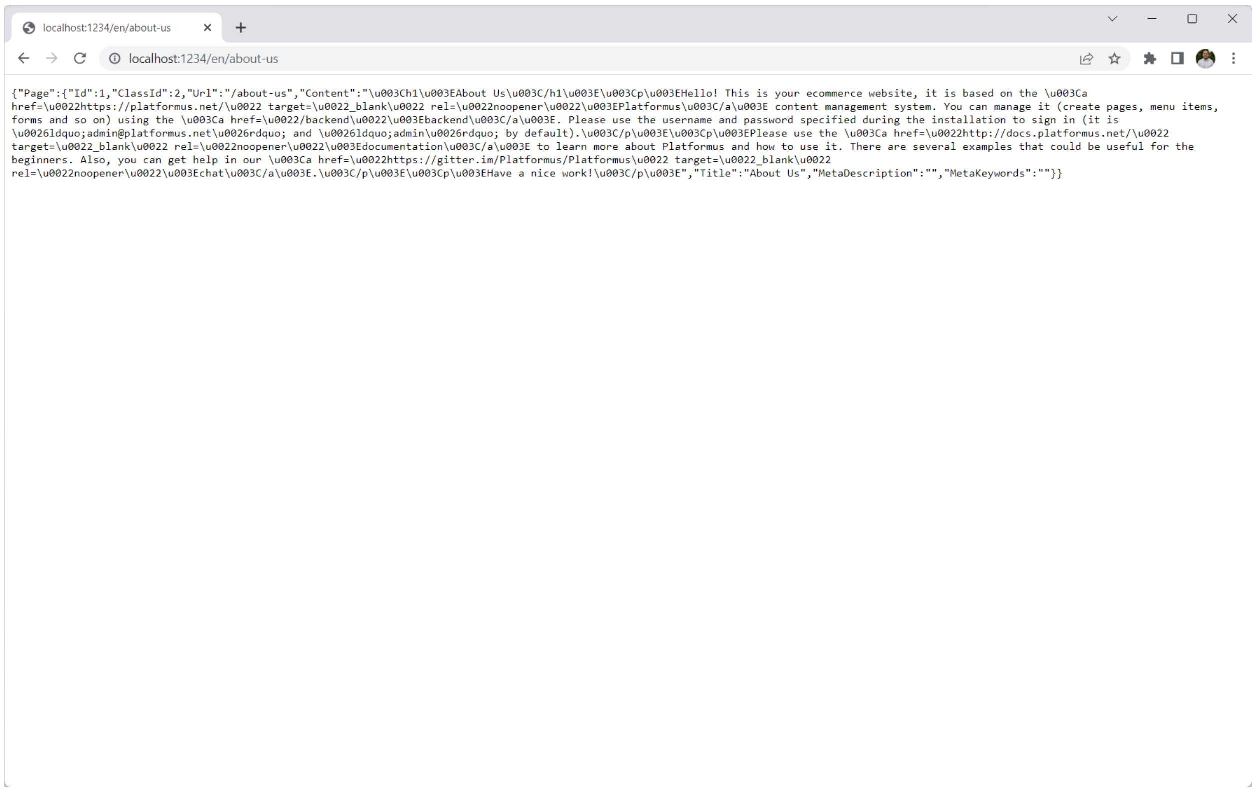


Request processor C# class name (required): WebApplication.MyRequestProcessor

Returns data as JSON.

Please note, that our new request processor C# class is automatically resolved and added to the drop-down list. Click the *Save* button.

Now navigate to /en/about and our new request processor will process the request:



```
{"Page":{"Id":1,"ClassId":2,"Url":"/about-us","Content":"\u003Ch1\u003EAbout Us\u003C/h1\u003E\u003Cp\u003EHello! This is your ecommerce website, it is based on the \u003Ca
href=\u0022https://platformus.net/\u0022 target=\u0022_blank\u0022 rel=\u0022noopener\u0022\u003EPlatformus\u003C/a\u003E content management system. You can manage it (create pages, menu items,
forms and so on) using the \u003Ca href=\u0022/backend\u0022\u003Ebackend\u003C/a\u003E. Please use the username and password specified during the installation to sign in (it is
\u0026ldquo;admin@platformus.net\u0026rdquo; and \u0026ldquo;admin\u0026rdquo; by default).\u003C/p\u003E\u003Cp\u003EPlease use the \u003Ca href=\u0022http://docs.platformus.net/\u0022
target=\u0022_blank\u0022 rel=\u0022noopener\u0022\u003Edocumentation\u003C/a\u003E to learn more about Platformus and how to use it. There are several examples that could be useful for the
beginners. Also, you can get help in our \u003Ca href=\u0022https://gitter.im/Platformus/Platformus\u0022 target=\u0022_blank\u0022
rel=\u0022noopener\u0022\u003Echat\u003C/a\u003E.\u003C/p\u003E\u003Cp\u003EHave a nice work!\u003C/p\u003E","Title":"About Us","MetaDescription":"","MetaKeywords":""}}
```

Good. Everything works as expected.

Endpoint Parameters and Parameter Groups

Now let's assume that we want to make it possible to specify the data format for our request processor output. It can be done using the parameters. Override the `ParameterGroups` property in our endpoint class:

```
public IEnumerable<ParameterGroup> ParameterGroups => new ParameterGroup[]
{
  new ParameterGroup(
    "Serialization",
    new Parameter(
      "Format",
      "Format",
      ParameterEditorCodes.RadioButtonList,
      new Option[] {
        new Option("JSON", "json"),
        new Option("XML", "xml")
      }
    )
  )
};
```

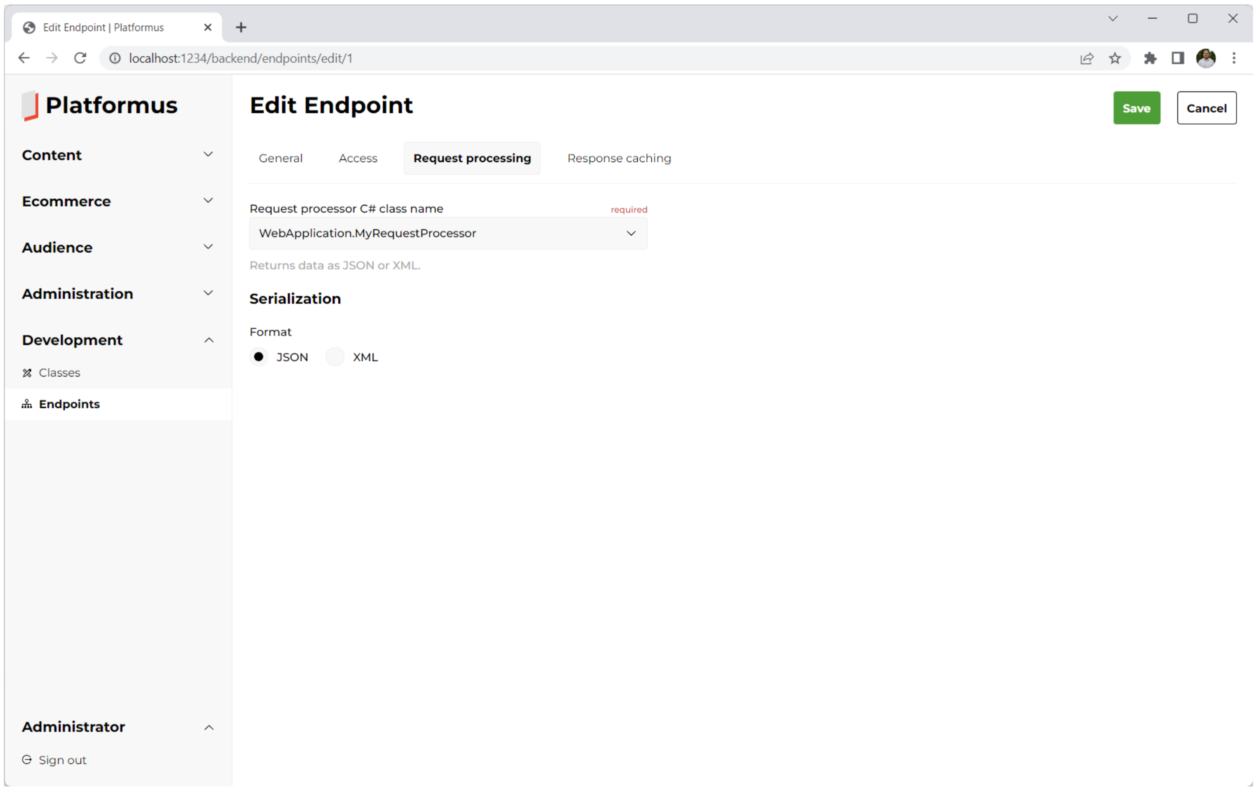
This property just returns the parameter groups. Each of them can contain different parameters defined by developer. When user selects the request processor C# class, these parameters will be available for him. Parameters can have different editor types. All the built-in ones are defined inside the ParameterEditorCodes enum, but you can also add your own ones. Using the parameter's code, you will be able to get the value entered by a user.

Let's update also the `Description` property to indicate that now we can return either JSON or XML:

```
public override string Description => "Returns data as JSON or XML.";
```

Description is also presented to a user when the request processor is selected.

Now open our endpoint in the backend one more time:



Request processor C# class name (required): WebApplication.MyRequestProcessor

Returns data as JSON or XML.

Serialization

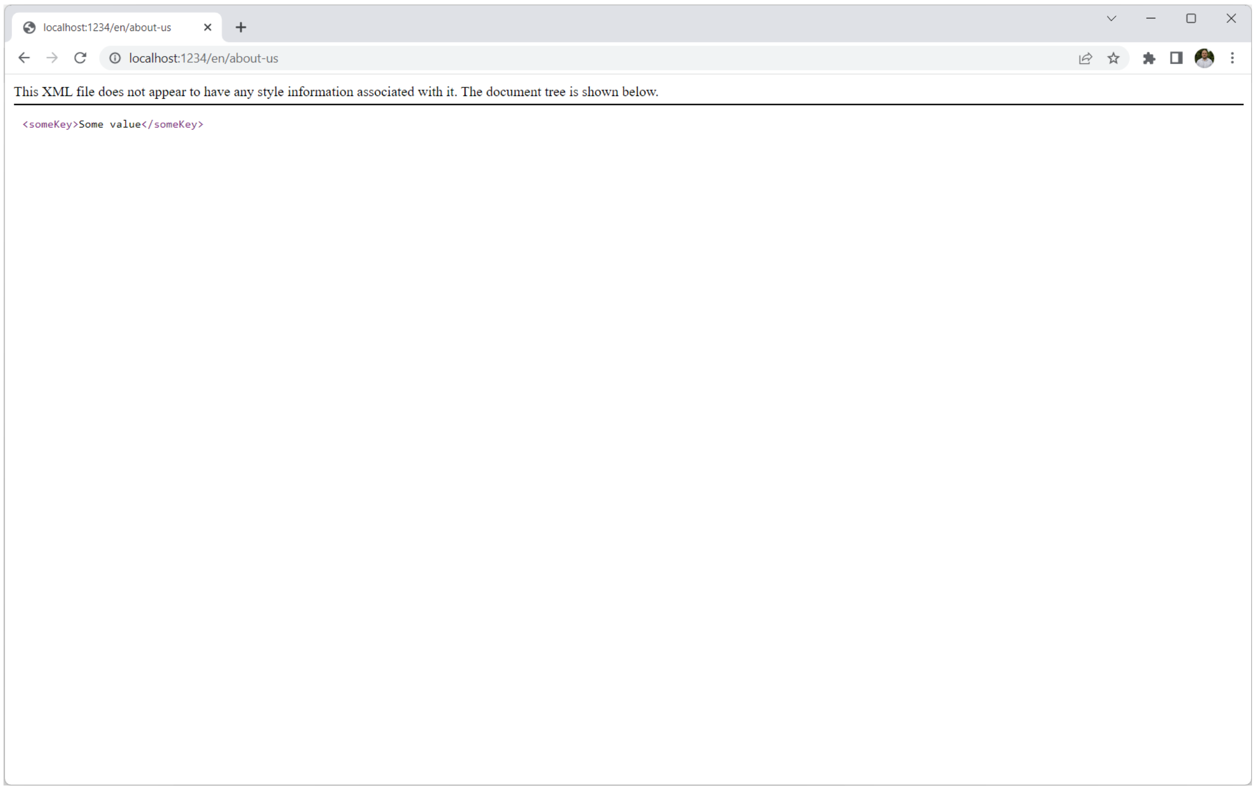
Format: [✓ JSON] XML

The request processor parameter is displayed. Let's see how to get and use the selected value from the code:

```
public async Task<IActionResult> ProcessAsync(HttpContext httpContext, Platformus.
↪Website.Data.Entities.Endpoint endpoint)
{
  dynamic viewModel = await this.CreateViewModelAsync(httpContext, endpoint);

  if (viewModel == null)
    return null;

  string format = new ParametersParser(endpoint.RequestProcessorParameters).
↪GetStringParameterValue("Format");

  if (format == "json")
    return new JsonResult(viewModel);

  XDocument document= new XDocument(
    new XElement("someKey", "Some value")
  );

  return new ContentResult() { Content = document.ToString(), ContentType =
↪"application/xml" };
}
```

Please note that we only use the hardcoded XML here, because converting dynamic object into an XML might look complex and isn't a subject of the article.

Now if you change the data format in the backend, the endpoint output will also be changed:



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<someKey>Some value</someKey>
```

1.4.2 Packages

- Platformus.Website
- Platformus.Website.Backend
- Platformus.Website.Data.Entities
- Platformus.Website.Data.EntityFramework.PostgreSql
- Platformus.Website.Data.EntityFramework.Sqlite
- Platformus.Website.Data.EntityFramework.SqlServer
- Platformus.Website.Frontend

1.5 Platformus.ECommerce

Implements standard ecommerce features like categories, products, photos, carts, and orders in a very basic way. Can be used as a starting point for ecommerce projects.

1.5.1 Admin Panel Sections

Categories

Products

Carts

Orders

Order States

Delivery Methods

Payment Methods

1.5.2 Packages

- Platformus.ECommerce
- Platformus.ECommerce.Backend
- Platformus.ECommerce.Data.Entities
- Platformus.ECommerce.Data.EntityFramework.PostgreSql
- Platformus.ECommerce.Data.EntityFramework.Sqlite
- Platformus.ECommerce.Data.EntityFramework.SqlServer
- Platformus.ECommerce.Frontend

1.6 Platformus.Images

Provides basic image processing (cropping and resizing).

1.6.1 Packages

- Platformus.Images

1.7 Custom Extensions

It is not difficult to develop a custom Platformus CMS extension. You can add dependency on any C# project in your Platformus-based application like in any other ASP.NET app, controllers and other features will work as you expect. But Platformus provides you with some public API, so you can extend it, add backend (admin panel) sections, security policies etc.

There are 2 main purposes to have custom Platformus extensions.

1. You can have all your code in the isolated projects, so CMS itself can be updated independently.

2. You can decrease development time reusing code and combining your apps from the existing parts. It could be useful when you develop a lot of apps and have standard approaches of fixing standard tasks. For example, most of the apps have authentication part, some Firebase cloud messaging features. Many of them also have chats on SignalR.

As Platformus CMS is built on top of the ExtCore framework, you can use your custom extension in the different ways: as NuGet packages, source code, or even DLL-files.

1.7.1 Backend Dashboard Widgets

Most of the apps have some key metrics, analytics, or statistics that should be available in a fast and convenient way. It could be number of the registered users or orders for the last week, sales amount etc.

The home page of the Platformus CMS backend (admin panel) is a dashboard where you can add your own widgets. Each widget is a regular view component, so it has its own view and can look and behave in any way.

To add your view component(s) to the dashboard you need to implement the IMetadata interface. It is preferable to use the MetadataBase class to be able to override only the methods you want:

```
public class MyMetadata : MetadataBase
{
  public override IEnumerable<DashboardWidget> GetDashboardWidgets(HttpContext␣
↪httpContext)
  {
    return new DashboardWidget[]
    {
      new DashboardWidget("MyViewComponent", 1000)
    };
  }
}
```

This file can be placed anywhere in the project, it will be resolved automatically by the default implementation of the IDashboardWidgetsProvider interface.

Let’s look at the DashboardWidget class’s properties.

`ViewComponentName` is the view component name.

`Position` is used to sort the widgets. Widgets with a lower position are placed higher.

1.7.2 Backend Menu

When you develop a custom Platformus CMS extension you might need to add items to the backend (admin panel) menu. All the backend menu groups and items are defined in the extensions. Each extension can provide one or more implementations of the IMetadata interface which allows to specify, among other things, the menu items grouped by the menu groups. It is preferable to use the MetadataBase class to be able to override only the methods you want:

```
public class MyMetadata : MetadataBase
{
  public override IEnumerable<MenuGroup> GetMenuGroups(HttpContext httpContext)
  {
    IStringLocalizer<Metadata> localizer = httpContext.GetStringLocalizer<Metadata>();

    return new MenuGroup[]
    {
      new MenuGroup(
        localizer["My Group"],
        1000,
        new MenuItem[]
        {
          new MenuItem("icon--icon1", "/backend/something-1", localizer["Something 1
↪"], 1000, "ManageSomething1"),
          new MenuItem("icon--icon2", "/backend/something-2", localizer["Something 2
↪"], 2000, "ManageSomething2"),
          new MenuItem("icon--icon3", "/backend/something-3", localizer["Something 3
↪"], 3000, "ManageSomething3")
```

(continues on next page)

(continued from previous page)

```
          }
        ),
      };
    }
  }
```

This file can be placed anywhere in the project, it will be resolved automatically by the default implementation of the IMenuGroupsProvider interface.

If the provided menu group's name matches name of the existing one, the menu items of both will be merged into the single group according to the menu item positions.

Let's look at the MenuItem class's properties.

`CssClass` allows to specify the CSS class that will be added to the menu item HTML tag. Intended to provide a custom icon but can also be used to apply another styling.

`Url` is the URL where user is navigated when clicks the menu item.

`Name` is displayed in the menu.

`Position` is used to sort the menu items (and menu groups). Items with a lower position are placed higher.

`PermissionCodes` contains an array of the codes of the *permissions* which are required for user to have in order to see the menu item.

1.7.3 Backend Styles and Scripts

When you develop a custom Platformus CMS extension you might need to add your own CSS styles and JavaScript scripts on the backend (admin panel) pages. Same as for the *menu*, it can be done by implementing the IMetadata interface or inheriting the MetadataBase class.

Please, take a look at the implementation from the *Platformus.Core* extension for a sample.

This file can be placed anywhere in the project, it will be resolved automatically by the default implementation of the IStyleSheetsProvider interface and the default implementation of the IScriptsProvider interface.

It is preferable to use links to minified CSS and JavaScript files to save traffic and speed up page loading. Also, you might need to *embed these files* into your extension to make it atomic.

1.7.4 Embedded Resources

Usually, your extension might need some resources (images, CSS, or JavaScript files etc.). You can add them to the host web application as regular static files, but in most cases, you would like your extensions to be atomic, especially if they are distributed as NuGet packages. It can be done by defining the resources that should be embedded in your *.csproj file:

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

After the resource is embedded, the underlying ExtCore framework will make it available using the HTTP requests. For example, if you embed a file as /wwwroot/images/photo.jpg, it will be available as /wwwroot.images.photo.jpg.

Please look how the static files are embedded and then used in the standard *Platformus.Core* extension.

1.7.5 Styles and Scripts Minification

Good practice is to minimize number of the HTTP requests by combining the CSS and JavaScript files. Also, these files can be minified to reduce their size by removing extra spaces, line breaks, replacing long variable names with the shorter ones etc. (The JavaScript files could be also obfuscated, which makes reverse engineering much harder.)

By default, Platformus uses the BundlerMinifier.Core package for that. To start using it add the corresponding reference to your *.csproj file:

```
<ItemGroup>
  <DotNetCliToolReference Include="BundlerMinifier.Core" Version="3.2.449" />
</ItemGroup>
```

Now, add the following lines to the same project file to run the bundling and minification process automatically on build:

```
<Target Name="PrecompileScript" BeforeTargets="BeforeBuild">
  <Exec Command="dotnet bundle" />
</Target>
```

Finally, the BundlerMinifier.Core package needs the bundleconfig.json file to be present in the project's root folder:

```
[
  {
    "outputFileName": "wwwroot/areas/backend/css/output.min.css",
    "inputFiles": [
      "Areas/Backend/Styles/input1.css",
      "Areas/Backend/Styles/input2.css",
      "Areas/Backend/Styles/input3.css"
    ]
  },
  {
    "outputFileName": "wwwroot/areas/backend/js/output.min.js",
    "inputFiles": [
      "Areas/Backend/Scripts/input1.js",
      "Areas/Backend/Scripts/input2.js",
      "Areas/Backend/Scripts/input3.js"
    ],
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    "sourceMap": false
  }
]
```

In this file you can specify which files and how exactly should be minified and combined. You can use the bundleconfig.json file from the *Platformus.Core* extension as a sample.

1.7.6 Tutorial: Creating a Custom Extension

When you create a Platformus-based application, unless it's something quite simple, you'll probably need to create your own Platformus extension. It will contain everything that is related to your app (entities, models, DTOs, services, APIs, UI etc.) to keep it separated and independent from the basic CMS.

Creating an extension is simple. We will go through all the aspects and create a sample mobile app backend with authentication (phone number validation using a fake code from SMS), categories and products API, and corresponding admin panel sections to manage categories and products etc.

Preparing the Solution

Let's start from the beginning. First, please follow the *Using as the NuGet Packages* tutorial to create an empty ASP.NET Core Platformus-based web application but keep only the Platformus.WebApplication and Platformus.Core packages dependencies as we do not need the others.

Recommended Solution Structure

You can organize your code in an any way you want, but it is recommended to use the following solution structure.

WebApplication

It is the executable main web application. It provides logging, configuration, stores static content, initializes Platformus.

Shouldn't be referenced from another projects.

WebApplication.Frontend

Contains everything related to the frontend: DTOs and view models, APIs with authentication configuration and validation rules, views, pages etc.

Shouldn't be referenced from other projects except the WebApplication one.

WebApplication.Backend

Contains everything related to the backend (admin panel): DTO and view models, controllers, views, pages etc.

Shouldn't be referenced from other projects except the WebApplication one.

WebApplication.Domain.Models

Contains only the domain models.

WebApplication.Domain.Abstractions

Contains the domain service interfaces (for those models which do not use and need to replace the generic domain service provided by the Magicalizer).

WebApplication.Domain.Defaults

Contains the domain service implementations (for those models which do not use and need to replace the generic domain service provided by the Magicalizer).

WebApplication.Data.Entities

Contains only the entities.

WebApplication.Data.Abstractions

Contains the repository interfaces (for those entities which do not use and need to replace the generic repository provided by the Magicalizer).

WebApplication.Data.EntityFramework.SqlServer

Contains the repository implementations for the specific database (for those entities which do not use and need to replace the generic repository provided by the Magicalizer) and the Entity Framework context configuration.