

Retail Manager Development Manual

Michael Kaip

August 9, 2019

Contents

1	Introduction	1
1.1	Project Summary	1
2	Initial Plan	2
2.1	Outline	2
2.2	Technologies	2
3	Initial Setup in Visual Studio	3
4	Creating a WebAPI with Authentification	4
4.1	Indentity Configuration	4
4.2	Getting authorized for development	4
4.2.1	Postman	4
4.2.2	Getting User Information	5
5	Installing and configuring SWAGGER	6
5.1	Installing SWAGGER	6
5.2	Channging the configuration of SWAGGER	6
5.3	Adding OAuth ability	7
6	SQL Database Setup	11
7	WPF with MVVM Project Setup	11
8	Dependency Injection in WPF	11
9	Planning the Register	11
10	SQL Database Table Creation	11
11	WPF Login Form Creation	11
12	Wiring up the WPF Login Form to the API	11
13	Login Form Error Handling	11

1 Introduction

1.1 Project Summary

The goal of this project is to build a desktop app that runs a cash register, handles inventory and manages an entire retail store. Creating and implementing a **WebAPI layer**, will allow the whole project to grow. This layer will be able to serve each kind of application (desktop, mobile, web, ...).

2 Initial Plan

2.1 Outline

The App is going to be build as a MVP (Minimum Viable Product) that can be expanded to cover all of the features, which are needed over time - so it can grow into a full featured application. First step is getting all of the major pieces set up, including:

- Git on Azure DevOps
- SQL Database (SSDT)
- WebAPI with Authentification
- WPF application that can log into the API

2.2 Technologies

- Unit Testing
- Dependency Injection
- WPF
- MVVM with Caliburn Micro
- ASP.NET MVC (Web Frontend)
- .NET Framework
- .NET Core 3.0
- SSDT - SQL Server Data Tools
- Git
- Azure DevOps
- Async
- Reporting
- WebAPI
- Logging
- Data Validation
- HTML
- CSS
- JavaScript
- Authentication

3 Initial Setup in Visual Studio

1. Setting up a Git-Repository, including README, GitIgnore (for VS) and License
2. Creating a **Blank Solution**: Other Project Types → Blank Solution
Such type of solution isn't language specific.

4 Creating a WebAPI with Authentication

1. Adding new Project to the Solution:

Web → ASP.NET Web Application (.NET Framework) → WebAPI

Add folders and references for:

- MVC
- Web API

Change Authentication to

- Individual User Accounts

2. Upgrading all NuGet-Packages

4.1 Identity Configuration

App_Start → IdentityConfig.cs

In there are some settings for setting up the WebAPI, especially for authentication:

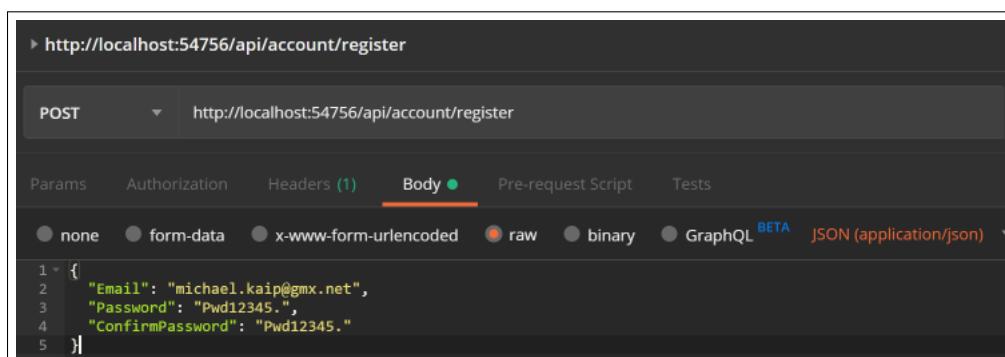
- UserValidator
- PasswordValidator

4.2 Getting authorized for development

4.2.1 Postman

The following calls has to be applied in the given order:

1. POST



Creates a new user account and stores this information into the user database.

If Status: **200 OK**, username and password has been successfully created.

2. GET

The screenshot shows a Postman interface with a GET request to `http://localhost:54756/token`. The **Body** tab is active, set to `x-www-form-urlencoded`. It contains three fields:

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> grant_type	password	
<input checked="" type="checkbox"/> username	michael.kaip@gmx.net	
<input checked="" type="checkbox"/> password	Pwd12345.	

It will return an **access_token** which is, by default, valid for 14 days. Token is needed for all further interaction with the server. Can be also configured for shorter valid periods.

3. POST

The screenshot shows a Postman interface with a POST request to `http://localhost:54756/api/values`. The **Headers** tab is active, containing one header:

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> Authorization	Bearer VuQDu7NWP8j-yd-tCRvkiqFwjBMZlaURbN...	
Key	Value	Description

4.2.2 Getting User Information

In order to get the Identity of users returned, some changes have to be implemented. Through this it's becomes possible to apply different accessibility rules, based on the user-group a certain user is part of.

1. `RMDataManager.Controllers.ValuesController`

```
using System.Web.Http;
using Microsoft.AspNet.Identity; // Needed for getting information about the user

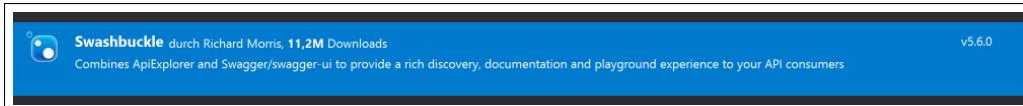
namespace RMDataManager.Controllers
{
    [Authorize]
    public class ValuesController : ApiController
    {
        // GET api/values
        public IEnumerable<string> Get()
        {
            // Stores the ID of each user
            var userId = RequestContext.Principal.Identity.GetUserId();
            return new string[] { "value1", "value2", userId };
        }
    }
}
```

5 Installing and configuring SWAGGER

SWAGGER is an API documentation and demonstration tool.

5.1 Installing SWAGGER

1. NuGet-Manager



Adds a SWAGGER to WebAPI-Projects.

2. Starting SWAGGER

The screenshot shows the Swagger UI interface running at <http://localhost:54756/swagger/docs/v1>. The title is "RMDataManager". The interface is divided into sections for "Account" and "Values".

- Account:**
 - GET /api/Account/UserInfo
 - POST /api/Account/Logout
 - GET /api/Account/ManageInfo
 - POST /api/Account/ChangePassword
 - POST /api/Account/SetPassword
 - POST /api/Account/AddExternalLogin
 - POST /api/Account/RemoveLogin
 - GET /api/Account/ExternalLogin
 - GET /api/Account/ExternalLogins
 - POST /api/Account/Register
 - POST /api/Account/RegisterExternal
- Values:**
 - GET /api/Values
 - POST /api/Values
 - DELETE /api/Values/{id}
 - GET /api/Values/{id}
 - PUT /api/Values/{id}

5.2 Channging the configuration of SWAGGER

RMDaTaManager.App_Start.SwaggerConfig.cs

1. Changing title

```
// Use "SingleApiVersion" to describe a single version API. Swagger 2.0 includes an "Info" object to
// hold additional metadata for an API. Version and title are required but you can also provide
// additional fields by chaining methods off SingleApiVersion.
//
c.SingleApiVersion("v1", title:"Retail Manager API"); // Changed to a proper name
```

2. Enabling proper printing of documents

```
// If you want the output Swagger docs to be indented properly, enable the "PrettyPrint" option.
//
c.PrettyPrint(); // enabled
```

3. Treating Enums as Strings

```
// In accordance with the built in JsonSerializer, Swashbuckle will, by default, describe enums as integers.
// You can change the serializer behavior by configuring the StringEnumConverter globally or for a given
// enum type. Swashbuckle will honor this change out-of-the-box. However, if you use a different
// approach to serialize enums as strings, you can also force Swashbuckle to describe them as strings.
//
c.DescribeAllEnumsAsStrings(); // enabled
```

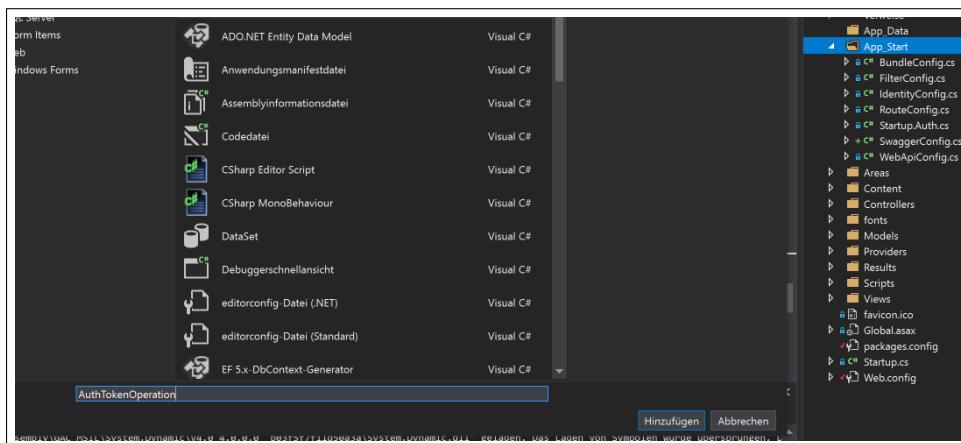
4. Changing document title

```
.EnableSwaggerUi(configure: c =>
{
    // Use the "DocumentTitle" option to change the Document title.
    // Very helpful when you have multiple Swagger pages open, to tell them apart.
    //
    c.DocumentTitle("RM API"); // changed the name
```

5.3 Adding OAuth ability

1. Enabling token endpoint allowance in the SWAGGER documentation

(a) Adding a new Class to `RMDaDataManager.App_Start`



(b) Implementing the required Interface

```
public class AuthTokenOperation : IDocumentFilter
{
    public void Apply(SwaggerDocument swaggerDoc, SchemaRegistry schemaRegistry, IApiExplorer apiExplorer)
    {
        swaggerDoc.paths.Add("/token", new PathItem
        {
            post = new Operation
            {
                tags = new List<string> { "Auth" },
                consumes = new List<string>
                {
                    "application/x-www-form-urlencoded"
                },
                parameters = new List<Parameter>
                {
                    new Parameter
                    {
                        type = "string",
                        name = "grant_type",
                        required = true,
                        @in = "formData",
                        @default = "password"
                    },
                    new Parameter
                    {
                        type = "string",
                        name = "username",
                        required = false,
                        @in = "formData"
                    },
                    new Parameter
                    {
                        type = "string",
                        name = "password",
                        required = false,
                        @in = "formData"
                    }
                }
            });
    }
}
```

(c) Applying it to SwaggerConfig.cs

```
GlobalConfiguration.Configuration
.EnableSwagger(configureC =>
{
    c.DocumentFilter<AuthTokenOperation>(); // adding the implemented document filter
})
```

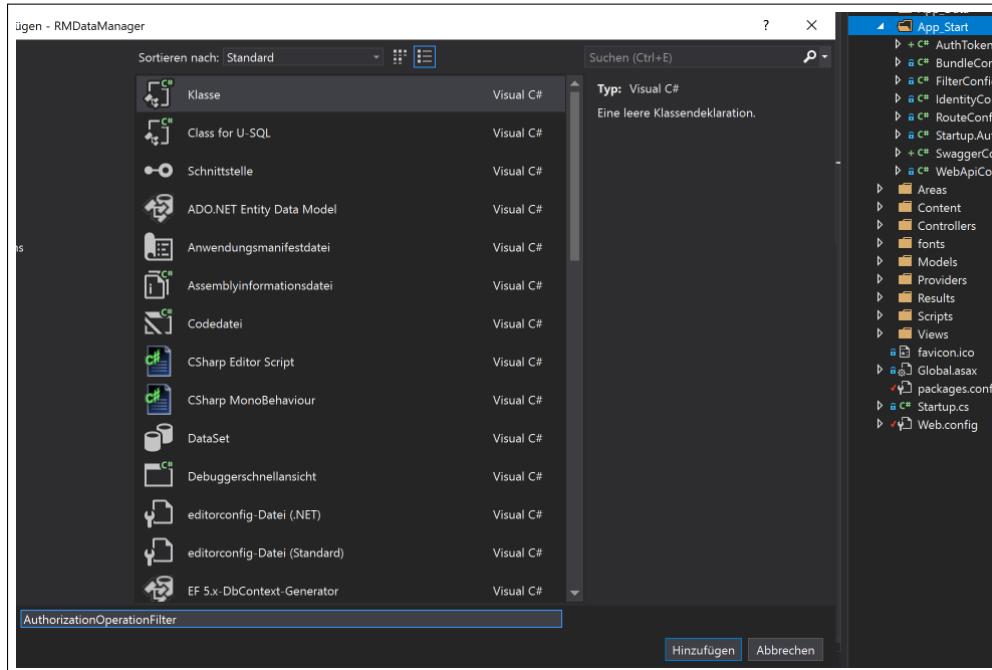
(d) Logging into the application using SWAGGER and get the token

The screenshot shows the Swagger UI for the Retail Manager API. The URL is `http://localhost:54756/swagger`. The `/token` endpoint is selected under the `Auth` section. The `Try it out!` button is highlighted. The response body shows a JSON object representing an access token:

```
{
  "access_token": "WofCgsbIKqtIOwsij4fhp4BuJy3HFLYxtqp-c3G59v01Kzc-tbE92xVP-RcBT9diUjZuhvEqPtl-4dQhKoxKhLQhHeSek8JN25XK-pTr
  "token_type": "bearer",
  "expires_in": 1209599,
  "username": "michael kaip@gmx.net",
  ".issued": "Thu, 08 Aug 2019 12:18:18 GMT",
  ".expires": "Thu, 22 Aug 2019 12:18:18 GMT"
}
```

2. Enabling to paste in the bearer token in order to authorize restricted commands

- (a) Adding a new Class to `RMDDataManager.App_Start`



- (b) Implementing the required Interface

```
0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
public class AuthorizationOperationFilter : IOperationFilter
{
    0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    public void Apply(Operation operation, SchemaRegistry schemaRegistry, ApiDescription apiDescription)
    {
        // Adding a parameter to each operation.
        if (operation.parameters == null)
        {
            operation.parameters = new List<Parameter>();
        }

        operation.parameters.Add(new Parameter
        {
            name = "Authorization",
            @in = "header",
            description = "access token",
            type = "string"
        });
    }
}
```

- (c) Applying it to SwaggerConfig.cs

```
.EnableSwagger(configure: c =>
{
    c.DocumentFilter<AuthTokenOperation>(); // adding the implemented document filter
    c.OperationFilter<AuthorizationOperationFilter>(); // adding the implemented operation filter
})
```

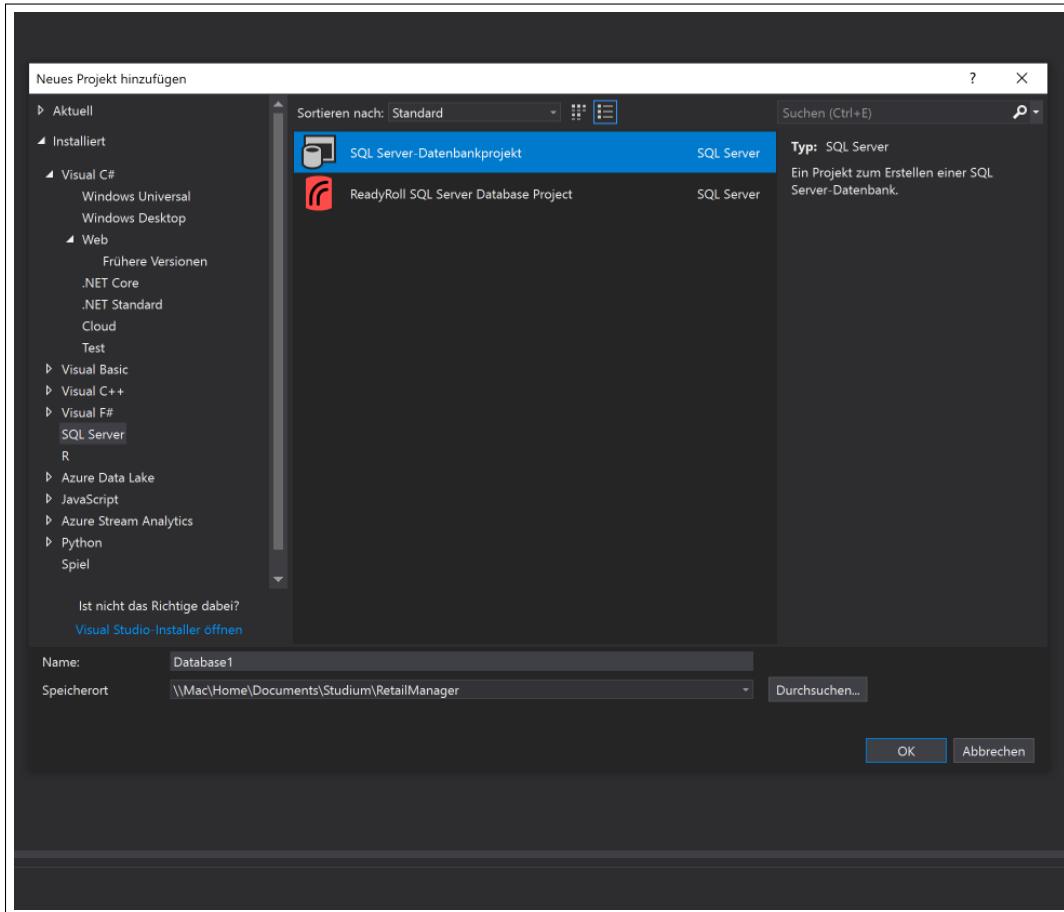
- (d) Get user information from the application via SWAGGER using the token

The screenshot shows the Swagger UI interface for a .NET Core API. The URL is `/api/values`. The response body is a JSON array containing a single string: `["string"]`. The response code is `200`. The response headers include:

```
{ "pragma": "no-cache", "date": "Thu, 08 Aug 2019 13:04:28 GMT", "server": "Microsoft-IIS/10.0", "x-powered-by": "ASP.NET", "content-type": "application/json; charset=utf-8", "x-content-type": "application/json", "x-sourcefiles": "\"JTF-8f87XFxNvNcS69t2vEb2N1bvVudHnCu3R1ZG11bVx5ZRhalxNvShZ2VyXFJNRGF0YU1hbmfzXcYXBpxFZhHV1cw==\"", "content-length": "58", "expires": "-1" }
```

6 SQL Database Setup

1. Adding new Database Project to the solution



7 WPF with MVVM Project Setup

8 Dependency Injection in WPF

9 Planning the Register

10 SQL Database Table Creation

11 WPF Login Form Creation

12 Wiring up the WPF Login Form to the API

13 Login Form Error Handling