

Retail Manager Development Manual

Michael Kaip

August 22, 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	Identity 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
6.1	Adding new Database Project to the solution	11
6.2	Adding several folders to the project	11
6.3	Creating a profile and publishing the Database	12
7	WPF with MVVM Project Setup	13
7.1	Adding the WPF Project to the solution	13
7.2	Changing the Assamly Name to the name of the solution in Properties	13
7.3	Adding Caliburn Micro MVVM-Framework	14
7.3.1	Adding the folder structure for the MVVM-Framework	14
7.3.2	Adding a new ShellViewModel class and a SchellView window	14
7.3.3	Adding a Bootstrapper class to DesktopUI	14
7.3.4	Removing StartUpURI from App.xaml and adding a new Ressource Dictionary	15
8	Dependency Injection in WPF	16
8.1	SimpleContainer in Caliburn Micro	16
8.1.1	Implementing SimpleContainer in Bootstrapper.cs	16
8.2	Overriding Configure() Method for the container	17
9	Datamodel - planning and setup	18
9.1	Planning the Register	18
9.2	SQL Database Table Creation	19
10	WPF Login Form Creation	20
10.1	Inheritance from the conductor class in Caliburn Micro	20
10.2	Implementing the menue bar	20
10.3	Adding a UserControl	20
10.3.1	Adding a class LoginViewModel (public) and UserControl LoginView	20
10.3.2	Designing the UserControl	21
10.3.3	Activating the LoginView on startup in the ShellView	21
10.3.4	Implementing LoginViewModel.cs	22
10.3.5	Connecting the LoginViewModel to Caliburn.Micro	23
11	Wiring up the WPF Login Form	24

11.1 Implementing a class AuthenticatedUser.cs	24
11.2 Implementing a helper class to handle API call interactions	25
11.3 Implementing a Interface IAPIHelper.cs	26
11.4 Adding <appsettings> to App.Config	26
11.5 Adding APIHelper and IAPIHelper to the container in Bootstrapper.cs	26
11.6 Applying some changings to LoginViewModel.cs	26
11.6.1 Adding a new private property as a backing field	26
11.6.2 Adding a constructor and initializing the property from within	27
11.6.3 Implementig the Login() method	27
11.7 Enabling the solution to start multiple projects	27
12 Login Form Error Handling	29
12.1 Displaying an login error message within the login form	29
13 Getting User Data	31
13.1 Adding a class library for the API	31
13.1.1 Installing Dapper	31
13.1.2 The SqlDataAccess class	32
13.1.3 The UserData class	33
13.1.4 Adding a UserController.cs to RMDDataManager.Controllers	34
14 Sales Page Creation	36
15 Event Aggregation in WPF	36
16 Displaying Product data	36
17 Wiring up WPF Shopping Cart	36
18 Modifying SQL, the API and WPF to add Taxes	36

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:

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:

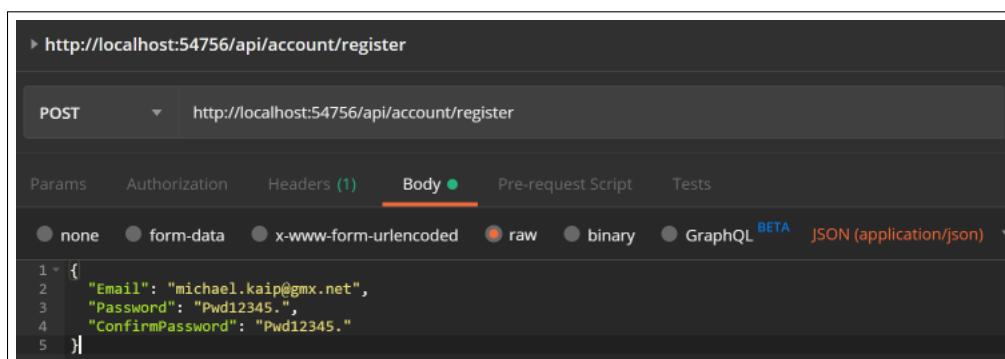
- UserValidator
- PasswordValidator

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

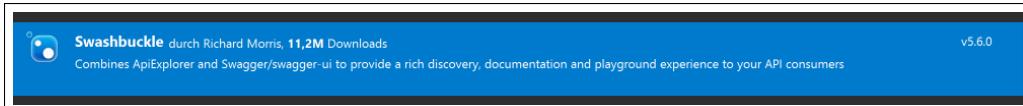
namespace RMDataManager.Controllers
{
    [Authorize]
    public class ValuesController : ApiController
    {
        // GET api/values
        public IEnumerable<string> Get()
        {
            // Stores the ID of each user
            var userId = RequestContext.Principal.Identity.GetUserId();
            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 for the RMDataManager API. The URL is `http://localhost:54756/swagger/docs/v1`. The interface is organized into sections:

- 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
```

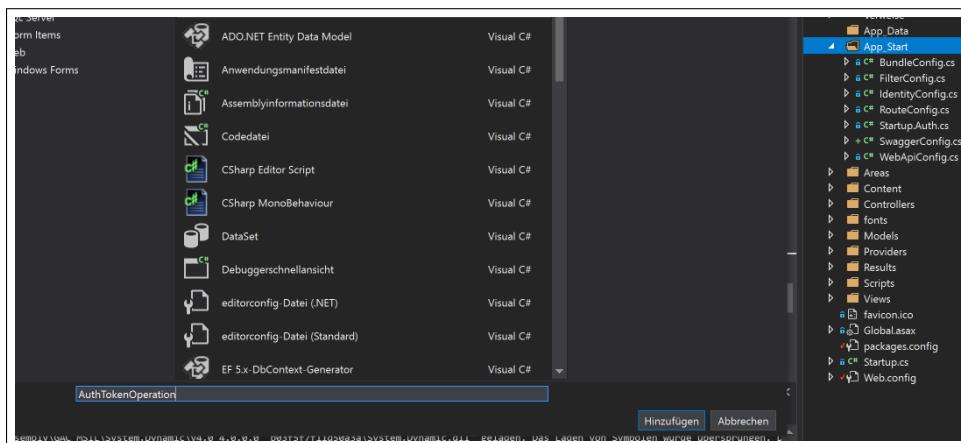
4. Changing document title

```
.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
})
```

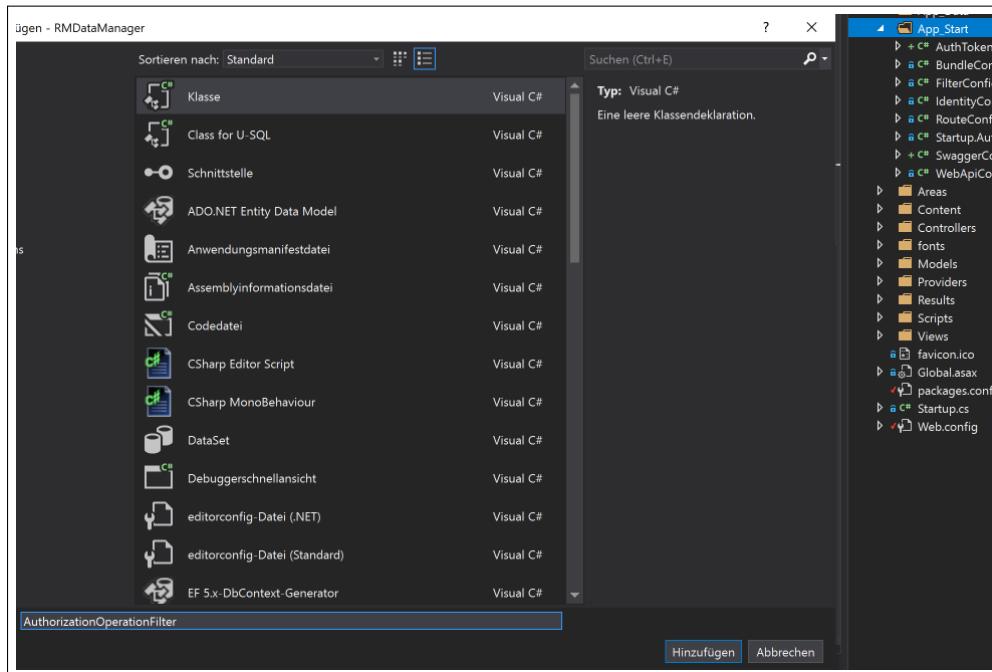
(d) Logging into the application using SWAGGER and get the token

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
})
```

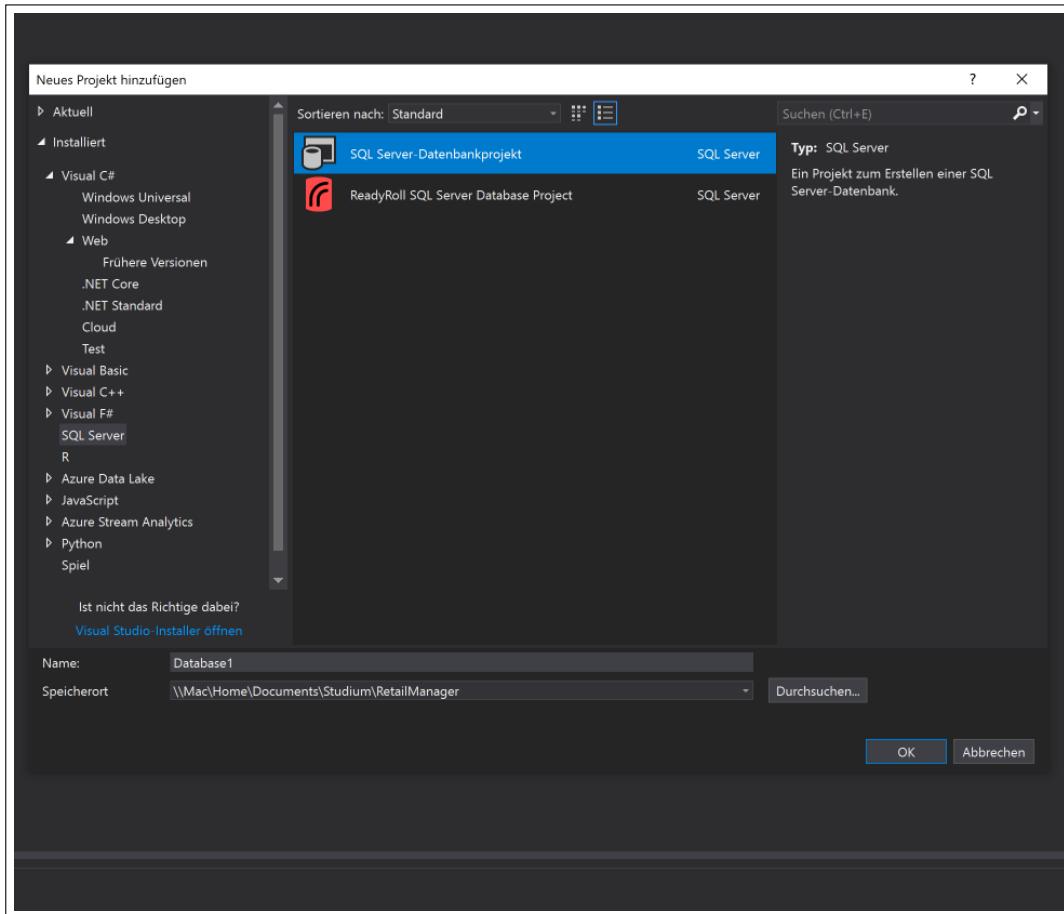
- (d) Get user information from the application via SWAGGER using the token

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:

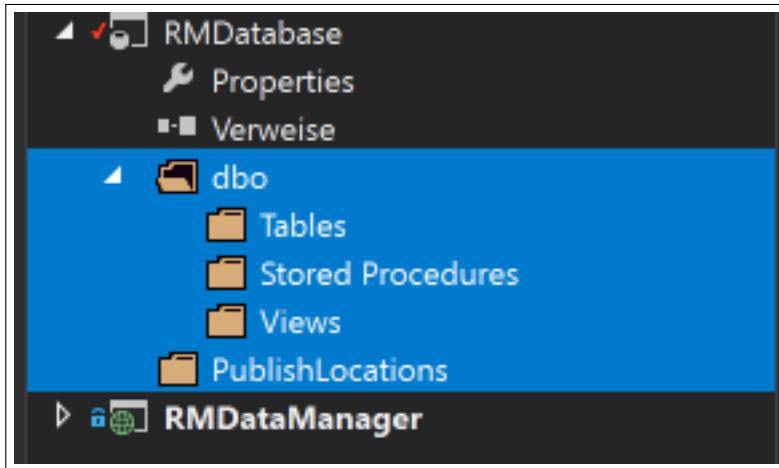
```
{ "pragma": "no-cache", "date": "Thu, 08 Aug 2019 13:04:28 GMT", "server": "Microsoft-IIS/10.0", "x-powered-by": "ASP.NET", "content-type": "application/json; charset=utf-8", "x-content-type": "application/json", "x-sourcefiles": "\"JTF-8f87XFxNvNcS69t2vEb2N1bwVudHnCu3R1ZG11bVx5ZRhalxNvShZ2VyXFJNRGF0YU1hbmfzXcYXBpxFZhHV1cw==\"", "content-length": "58", "expires": "-1" }
```

6 SQL Database Setup

6.1 Adding new Database Project to the solution

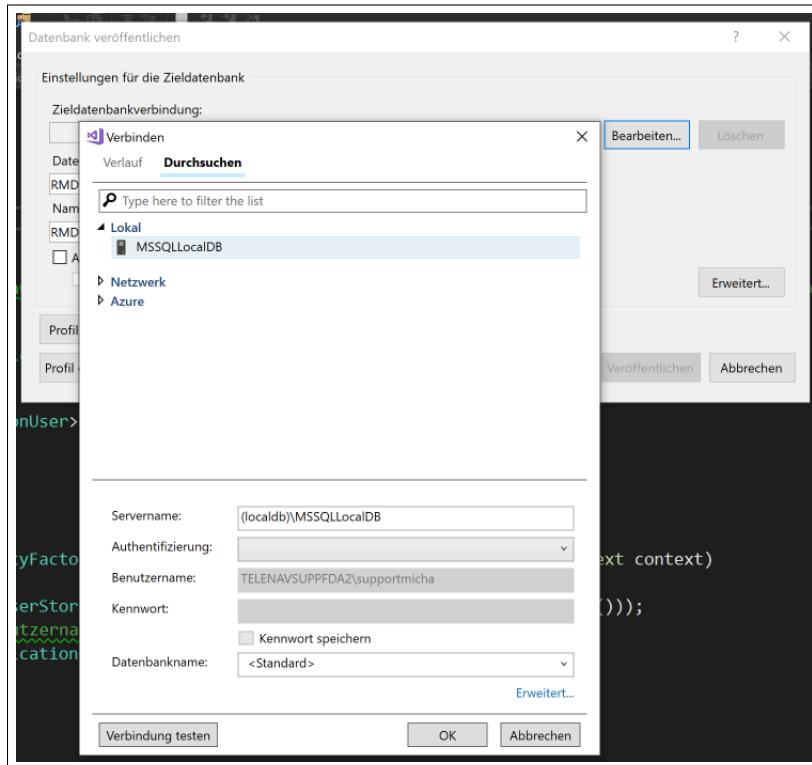


6.2 Adding several folders to the project

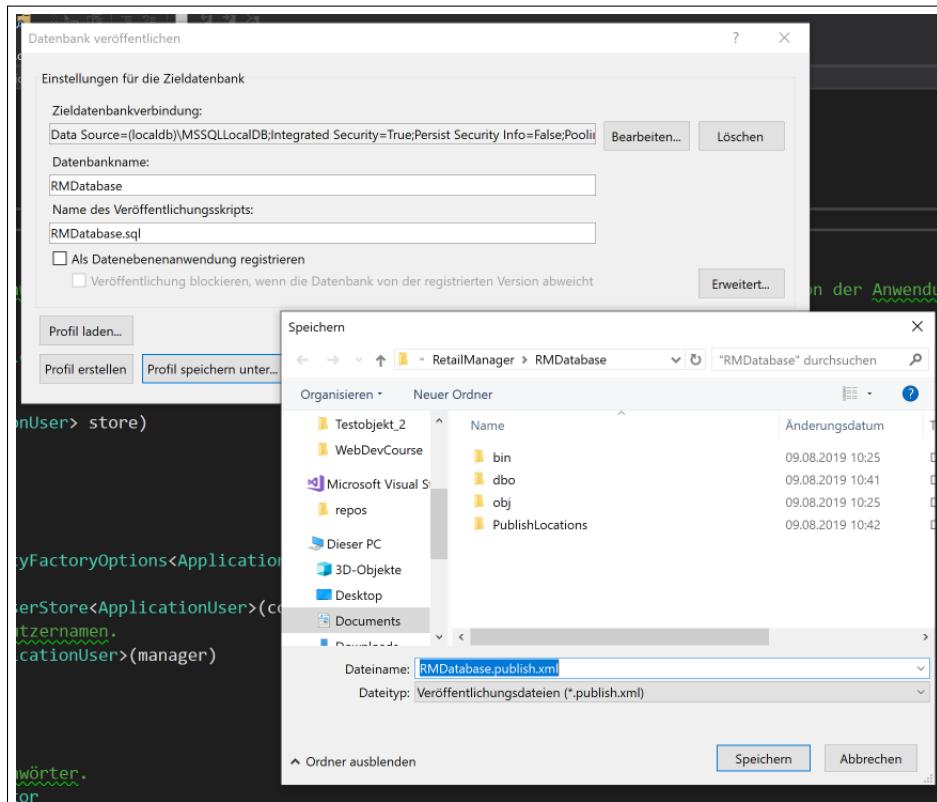


6.3 Creating a profile and publishing the Database

- RightClick on RMDatabase → Publish → Edit → Browse

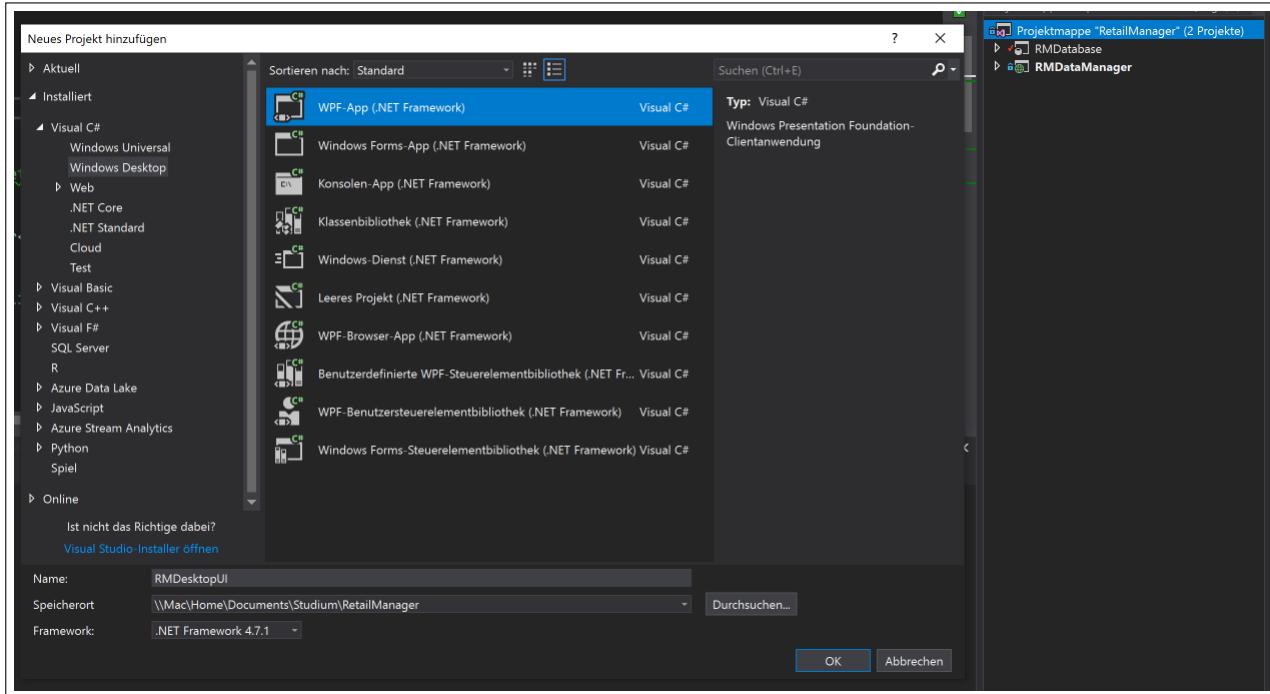


- Naming and saving profile to PublishLocations

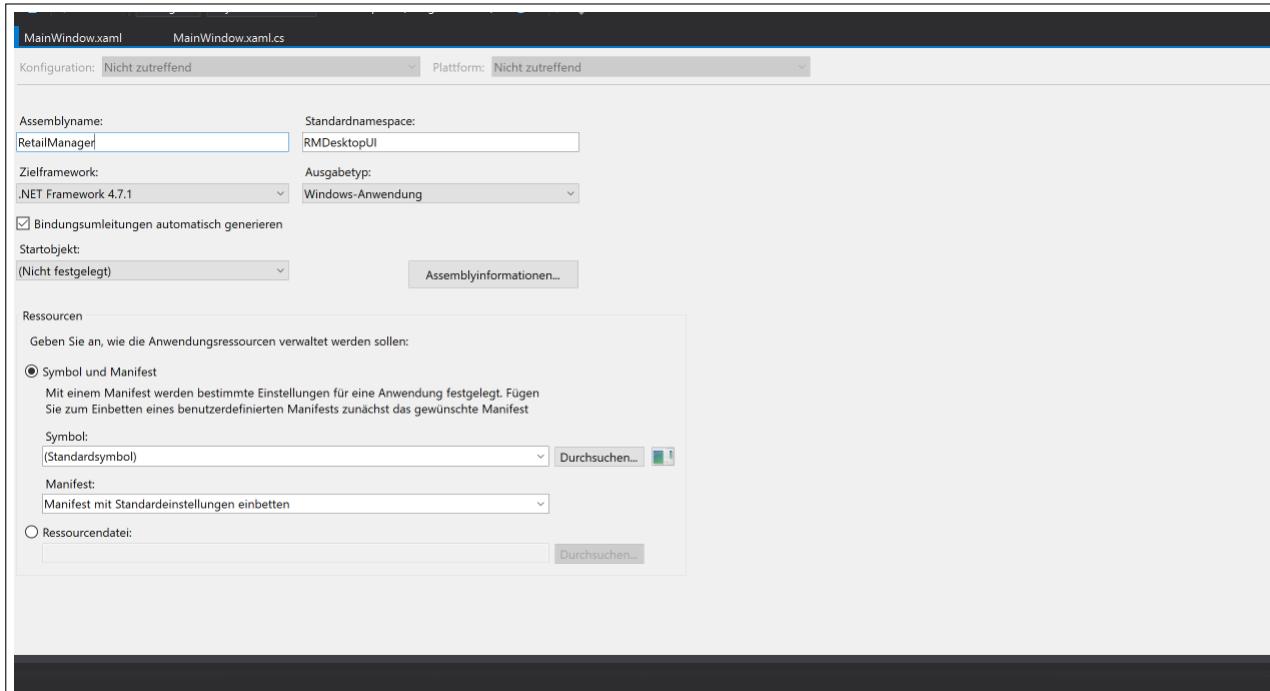


7 WPF with MVVM Project Setup

7.1 Adding the WPF Project to the solution



7.2 Changing the Assembly Name to the name of the solution in Properties

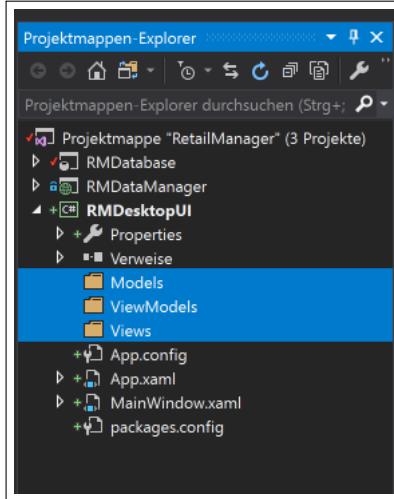


Also set project as the default startup-project.

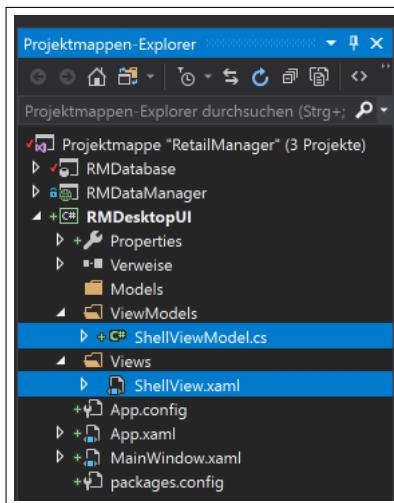
7.3 Adding Caliburn Micro MVVM-Framework

Add NuGet-Package to references.

7.3.1 Adding the folder structure for the MVVM-Framework



7.3.2 Adding a new ShellViewModel class and a ShellView window



7.3.3 Adding a Bootstrapper class to DesktopUI

```

using System.Windows;
using Caliburn.Micro;
using RMDesktopUI.ViewModels;

namespace RMDesktopUI
{
    public class Bootstrapper : BootstrapperBase
    {
        // Constructor
        public Bootstrapper()
        {
            Initialize();
        }

        // Enables, that on start up ShellViewModel is getting launched as the base view.
        protected override void OnStartup(object sender, StartupEventArgs e)
        {
            DisplayRootViewFor<ShellViewModel>();
        }
    }
}

```

7.3.4 Removing StartUpURI from App.xaml and adding a new Ressource Dictionary

The screenshot shows the Visual Studio IDE. On the left, the code editor displays `Bootstrapper.cs` with the following XAML code:

```
<Application x:Class="RMDesktopUI.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:RMDesktopUI">

    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary>
                    <local:Bootstrapper x:Key="Bootstrapper" />
                </ResourceDictionary>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

On the right, the Project Explorer window shows the project structure for "RetailManager" (3 projects):

- Projektmappe "RetailManager" (3 Projekte)
 - RMDatabase
 - RMDDataManager
 - + RMDesktopUI
 - Properties
 - Verweise
 - Models
 - ViewModels
 - ShellViewModel.cs
 - Views
 - ShellView.xaml
 - App.config
 - App.xaml
 - Bootstrapper.cs
 - packages.config

MainWindow.xaml can be deleted afterwards!!!

8 Dependency Injection in WPF

8.1 SimpleContainer in Caliburn Micro

Caliburn.Micro comes pre-bundled with a Dependency Injection container called SimpleContainer. A dependency injection container is an object that is used to hold dependency mappings for use later in an app via Dependency Injection. Dependency Injection is actually a pattern typically using the container element instead of manual service mapping.

8.1.1 Implementing SimpleContainer in Bootstrapper.cs

```

1-Verweis | 0 Änderungen | 0 Autoren, 0 Änderungen
public class Bootstrapper : BootstrapperBase
{
    private SimpleContainer _container = new SimpleContainer();
    // Constructor
    0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    public Bootstrapper()
    {
        Initialize();
    }

    // Enables, that on start up ShellViewModel is getting launched as the base view.
    0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    protected override void OnStartup(object sender, StartupEventArgs e)
    {
        DisplayRootViewFor<ShellViewModel>();
    }

    0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    protected override object GetInstance(Type service, string key)
    {
        return _container.GetInstance(service, key);
    }

    0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    protected override IEnumerable<object> GetAllInstances(Type service)
    {
        return _container.GetAllInstances(service);
    }

    0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    protected override void BuildUp(object instance)
    {
        _container.BuildUp(instance);
    }
}

```

The screenshot shows the Visual Studio IDE. The left pane displays the code for the `Bootstrapper.cs` file. The right pane shows the `Projektmappen-Explorer` (Project Explorer) with the project structure:

- Projektmappe "RetailManager"
 - RMDatabase
 - RMDDataManager
 - RMDesktopUI
 - Properties
 - Verweise
 - Models
 - ViewModels
 - ShellViewModel.cs
 - Views
 - ShellView.xaml
 - App.config
 - App.xaml
 - Bootstrapper.cs
 - packages.config

8.2 Overriding Configure() Method for the container

```
1-Verweis | 0 Änderungen | 0 Autoren, 0 Änderungen
public class Bootstrapper : BootstrapperBase
{
    private SimpleContainer _container = new SimpleContainer();
    // Constructor
    0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    public Bootstrapper()
    {
        Initialize();
    }

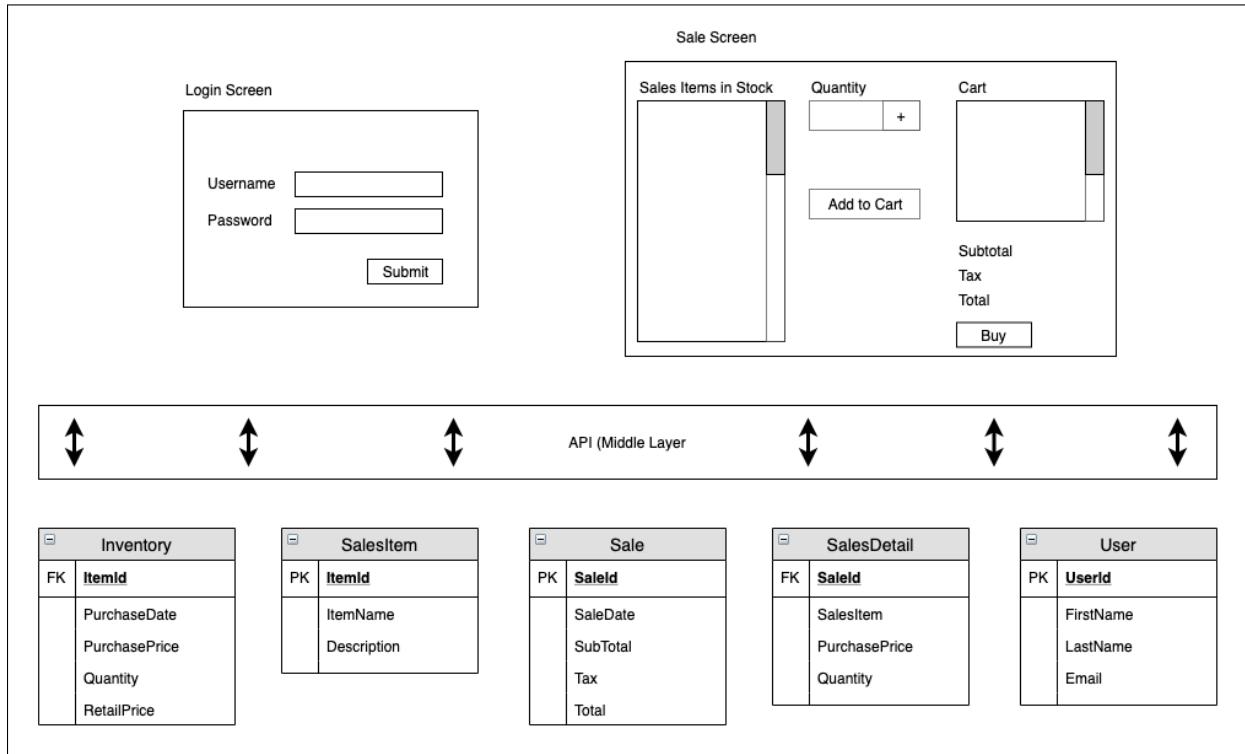
    0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    protected override void Configure()
    {
        _container.Instance(_container);

        _container
            .Singleton<IWindowManager, WindowManager>()
            .Singleton<EventAggregator, EventAggregator>();

        // Connecting the ViewModel to the Views using reflection
        GetType().Assembly.GetTypes()
            .Where(type => type.IsClass)
            .Where(type => type.Name.EndsWith("ViewModel"))
            .ToList()
            .ForEach(action: viewmodelType => _container.RegisterPerRequest(
                service: viewmodelType, viewmodelType.ToString(), implementation: viewmodelType));
    }
}
```

9 Datamodel - planning and setup

9.1 Planning the Register



9.2 SQL Database Table Creation

User.sql

Skriptdatei: User.sql

Name	Datentyp	NULL-Werte zulassen	Standard
Id	nvarchar(128)	<input checked="" type="checkbox"/>	
FirstName	nvarchar(50)	<input checked="" type="checkbox"/>	
LastName	nvarchar(50)	<input checked="" type="checkbox"/>	
EmailAddress	nvarchar(256)	<input checked="" type="checkbox"/>	
CreatedDate	datetime2(7)	<input checked="" type="checkbox"/>	getutcdate()

```
Design T-SQL
1 CREATE TABLE [dbo].[User]
2 (
3     [Id] NVARCHAR(128) NOT NULL,
4     [FirstName] NVARCHAR(50) NOT NULL,
5     [LastName] NVARCHAR(50) NOT NULL,
6     [EmailAddress] NVARCHAR(256) NOT NULL,
7     [CreatedDate] DATETIME2 NOT NULL DEFAULT getutcdate()
8 )
9
```

Product.sql

Skriptdatei: Product.sql

Name	Datentyp	NULL-Werte zulassen	Standard
Id	int	<input checked="" type="checkbox"/>	
ProductName	nvarchar(100)	<input checked="" type="checkbox"/>	
Description	nvarchar(MAX)	<input checked="" type="checkbox"/>	
RetailPrice	money	<input checked="" type="checkbox"/>	
CreateDate	datetime2(7)	<input checked="" type="checkbox"/>	getutcdate()
LastModified	datetime2(7)	<input checked="" type="checkbox"/>	getutcdate()

```
Design T-SQL
1 CREATE TABLE [dbo].[Product]
2 (
3     [Id] INT NOT NULL PRIMARY KEY IDENTITY,
4     [ProductName] NVARCHAR(100) NOT NULL,
5     [Description] NVARCHAR(MAX) NOT NULL,
6     [RetailPrice] MONEY NOT NULL,
7     [CreateDate] DATETIME2 NOT NULL DEFAULT getutcdate(),
8     /* Has to be modified manually everytime the entry gets modified.*/
9     [LastModified] DATETIME2 NOT NULL DEFAULT getutcdate()
10)
```

Sale.sql

Skriptdatei: Sale.sql

Name	Datentyp	NULL-Werte zulassen	Standard
Id	int	<input checked="" type="checkbox"/>	
CashierId	nvarchar(128)	<input checked="" type="checkbox"/>	
SaleDate	datetime2(7)	<input checked="" type="checkbox"/>	
SubTotal	money	<input checked="" type="checkbox"/>	
Tax	money	<input checked="" type="checkbox"/>	
Total	money	<input checked="" type="checkbox"/>	

```
Design T-SQL
1 CREATE TABLE [dbo].[Sale]
2 (
3     [Id] INT NOT NULL PRIMARY KEY IDENTITY, /* IDENTITY Makes the Id auto increment */
4     [CashierId] NVARCHAR(128) NOT NULL,
5     [SaleDate] DATETIME2 NOT NULL,
6     [SubTotal] MONEY NOT NULL,
7     [Tax] MONEY NOT NULL,
8     [Total] MONEY NOT NULL
9 )
```

SaleDetail.sql

Skriptdatei: SaleDetail.sql

Name	Datentyp	NULL-Werte zulassen	Standard
Id	int	<input checked="" type="checkbox"/>	
SaleId	int	<input checked="" type="checkbox"/>	
ProductId	int	<input checked="" type="checkbox"/>	
Quantity	nchar(10)	<input checked="" type="checkbox"/>	1
PurchasePrice	money	<input checked="" type="checkbox"/>	
Tax	money	<input checked="" type="checkbox"/>	0

```
Design T-SQL
1 CREATE TABLE [dbo].[SaleDetail]
2 (
3     [Id] INT NOT NULL PRIMARY KEY IDENTITY,
4     [SaleId] INT NOT NULL,
5     [ProductId] INT NOT NULL,
6     [Quantity] NCHAR(10) NULL DEFAULT 1,
7     [PurchasePrice] MONEY NOT NULL,
8     [Tax] MONEY NOT NULL DEFAULT 0,
9 )
```

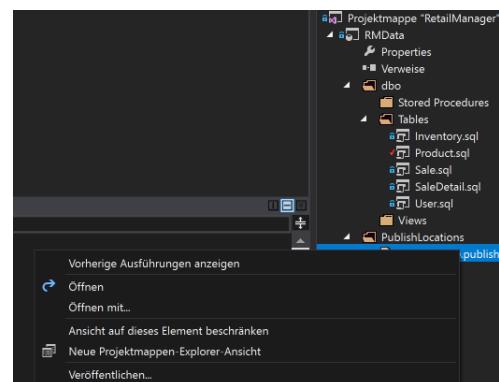
Inventory.sql

Skriptdatei: Inventory.sql

Name	Datentyp	NULL-Werte zulassen	Standard
Id	int	<input checked="" type="checkbox"/>	
ProductId	int	<input checked="" type="checkbox"/>	
Quantity	nchar(10)	<input checked="" type="checkbox"/>	1
PurchasePrice	money	<input checked="" type="checkbox"/>	
PurchaseDate	datetime2(7)	<input checked="" type="checkbox"/>	getutcdate()

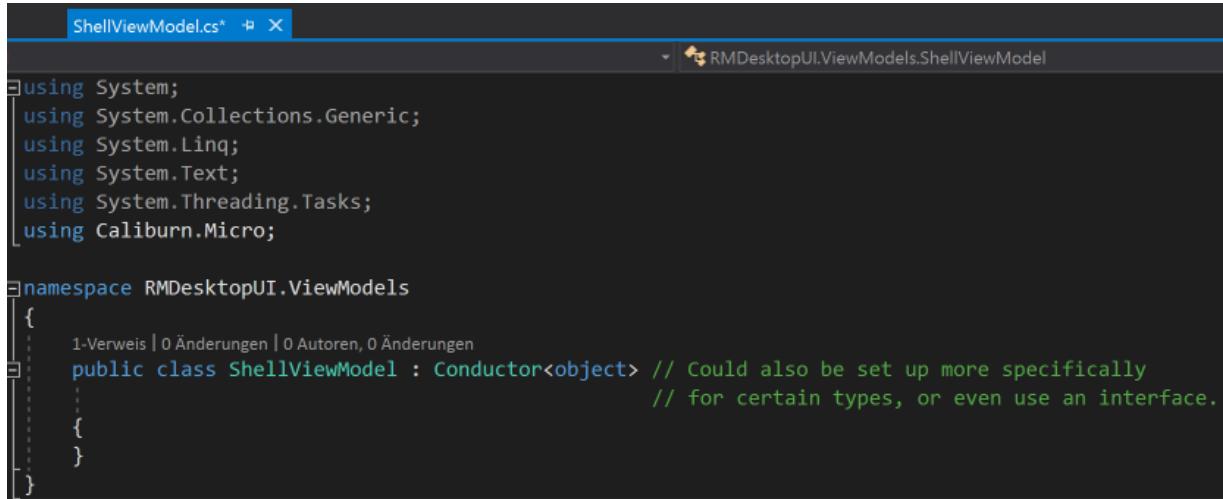
```
Design T-SQL
1 CREATE TABLE [dbo].[Inventory]
2 (
3     [Id] INT NOT NULL PRIMARY KEY IDENTITY,
4     [ProductId] INT NOT NULL,
5     [Quantity] NCHAR(10) NOT NULL DEFAULT 1,
6     [PurchasePrice] MONEY NOT NULL,
7     [PurchaseDate] DATETIME2 NULL DEFAULT getutcdate()
8 )
9
```

!!! Publishing Tables !!!



10 WPF Login Form Creation

10.1 Inheritance from the conductor class in Caliburn Micro



```

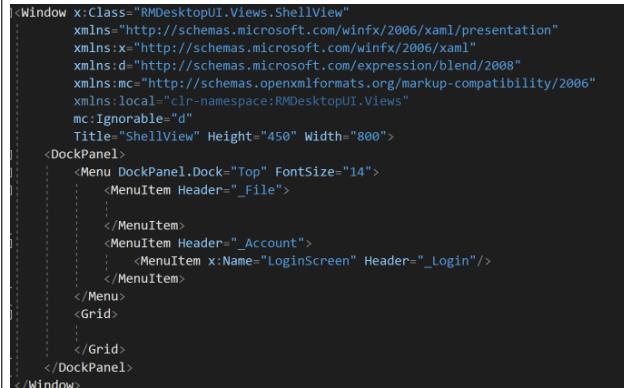
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Caliburn.Micro;

namespace RMDesktopUI.ViewModels
{
    public class ShellViewModel : Conductor<object> // Could also be set up more specifically
        // for certain types, or even use an interface.
}

```

Conductor is a base class which inherits from Screen. Its responsibility is to conduct other objects by managing an active item and maintain a strict lifecycle of this conducted item. The conductor exists in multiple variants such as the one item conductor simple called Conductor, the multiple item conductors such as `Conductor.Collection.OneActive` and `Conductor.Collection.AllActive`.

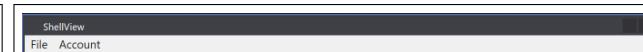
10.2 Implementing the menu bar



```

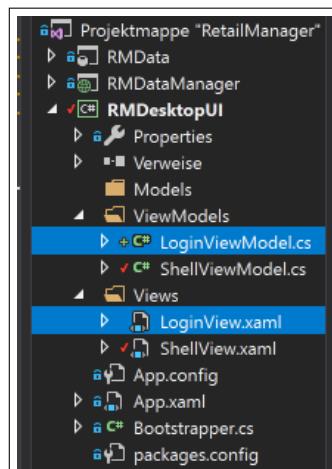
<Window x:Class="RMDesktopUI.Views.ShellView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:RMDesktopUI.Views"
    mc:Ignorable="d"
    Title="ShellView" Height="450" Width="800">
    <DockPanel>
        <Menu DockPanel.Dock="Top" FontSize="14">
            <MenuItem Header="_File">
                ...
            </MenuItem>
            <MenuItem Header="_Account">
                <MenuItem x:Name="LoginScreen" Header="_Login"/>
            </MenuItem>
        </Menu>
        <Grid>
            ...
        </Grid>
    </DockPanel>
</Window>

```



10.3 Adding a UserControl

10.3.1 Adding a class LoginViewModel (public) and UserControl LoginView



10.3.2 Designing the UserControl

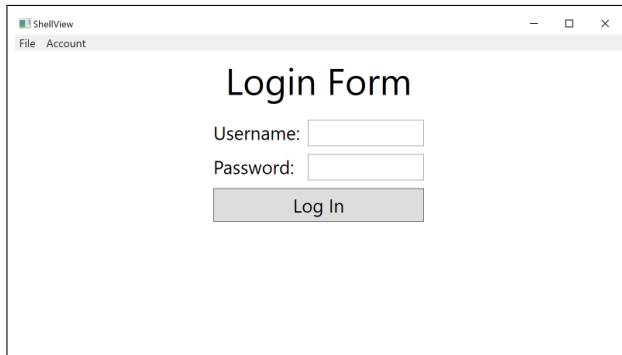
```
<UserControl x:Class="RMDesktopUI.Views.LoginView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:RMDesktopUI.Views"
    mc:Ignorable="d" Background="White" FontSize="24"
    d:DesignHeight="425" d:DesignWidth="800">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/> <!-- Username -->
            <RowDefinition Height="Auto"/> <!-- Password -->
            <RowDefinition Height="Auto"/> <!-- Button -->
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <TextBlock Grid.Row="0" Grid.Column="1" Grid.ColumnSpan="2"
            HorizontalAlignment="Center" FontSize="48" Margin="0 0 0 20">
            Login Form
        </TextBlock>

        <!-- Username Row -->
        <TextBlock Grid.Row="1" Grid.Column="1" Margin="0 0 10 0">
            Username:
        </TextBlock>
        <TextBox x:Name="userName" Grid.Row="1" Grid.Column="2" MinWidth="150"
            Margin="0 0 0 10"/>
        <!-- Password Row -->
        <TextBlock Grid.Row="2" Grid.Column="1" Margin="0 0 10 0">
            Password:
        </TextBlock>
        <TextBox x:Name="password" Grid.Row="2" Grid.Column="2" MinWidth="150"
            Margin="0 0 0 10"/>
        <!-- Button Row -->
        <Button Grid.Row="3" Grid.Column="1" Grid.ColumnSpan="2" Padding="10 5"
            HorizontalContentAlignment="Center" x:Name="loginButton">
            Log In
        </Button>
    </Grid>
</UserControl>
```



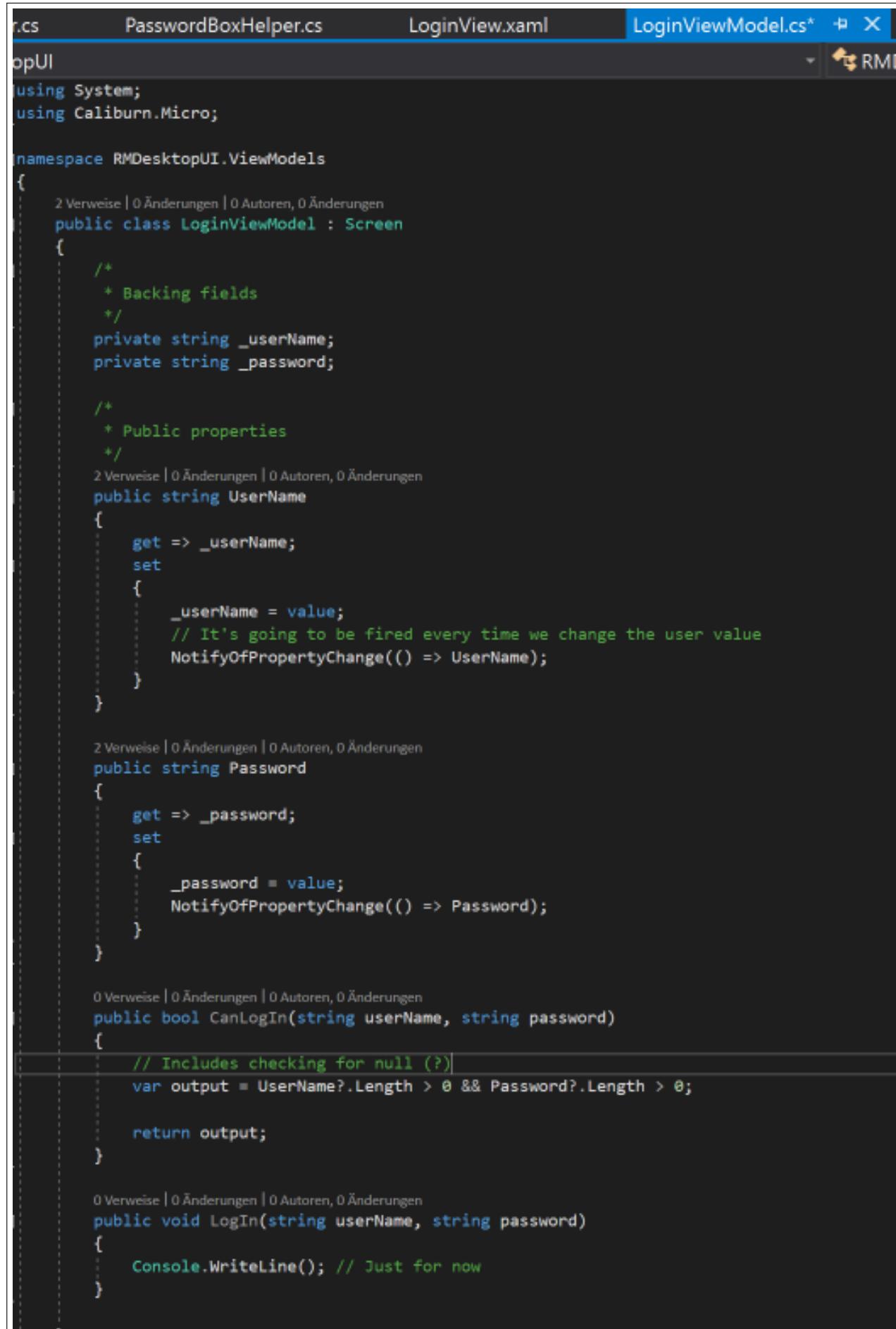
10.3.3 Activating the LoginView on startup in the ShellView

```
nView.xaml      LoginViewModel.cs      ShellView.xaml      ShellViewModel.cs  - x
RMDesktopUI
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using Caliburn.Micro;
7
8  namespace RMDesktopUI.ViewModels
9  {
10     // 2 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
11     public class ShellViewModel : Conductor<object> // Could also be set up more specific
12     // for certain types, or even use sealed
13
14     // Using constructor injection to pass in a new instance of loginVm
15     private LoginViewModel _loginVm;
16     // 0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
17     public ShellViewModel(LoginViewModel loginVm)
18     {
19         _loginVm = loginVm;
20         ActivateItem(_loginVm);
21     }
22
23     // Preventing from inheritance, because of virtual member call in constructor
24     public sealed override void ActivateItem(object item)
25     {
26         base.ActivateItem(item);
27     }
28 }
```



Sealed is used to restrict the users from inheriting. A class can be sealed by using the sealed keyword or a single method. The keyword tells the compiler that class or method cannot be extended. No class can be derived from a sealed class.

10.3.4 Implementing LoginViewModel.cs



The screenshot shows a Windows-based IDE interface with multiple tabs open. The active tab is 'LoginViewModel.cs'. The code within this tab implements a Caliburn.Micro 'Screen' for a login view. It contains backing fields for 'UserName' and 'Password', and public properties with corresponding get and set methods. The 'UserName' property includes logic to raise a 'NotifyPropertyChanged' event whenever its value changes. The 'CanLogIn' method checks if both the user name and password are provided (non-null and non-empty). The 'LogIn' method simply outputs a message to the console.

```
opUI
r.cs      PasswordBoxHelper.cs      LoginView.xaml      LoginViewModel.cs*  X
RMD
using System;
using Caliburn.Micro;

namespace RMDesktopUI.ViewModels
{
    2 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    public class LoginViewModel : Screen
    {
        /*
         * Backing fields
         */
        private string _userName;
        private string _password;

        /*
         * Public properties
         */
        2 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
        public string UserName
        {
            get => _userName;
            set
            {
                _userName = value;
                // It's going to be fired every time we change the user value
                NotifyOfPropertyChanged(() => UserName);
            }
        }

        2 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
        public string Password
        {
            get => _password;
            set
            {
                _password = value;
                NotifyOfPropertyChanged(() => Password);
            }
        }

        0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
        public bool CanLogIn(string userName, string password)
        {
            // Includes checking for null (?)
            var output = UserName?.Length > 0 && Password?.Length > 0;

            return output;
        }

        0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
        public void LogIn(string userName, string password)
        {
            Console.WriteLine(); // Just for now
        }
    }
}
```

10.3.5 Connecting the LoginViewModel to Caliburn.Micro

1. Adding a helper class (PasswordBoxHelper.cs) to RMDesktopUI

```

using System.Reflection;
using System.Windows;
using System.Windows.Controls;

namespace RMDesktopUI.Helpers
{
    /*
     * The aim of this class is to include a binding convention so that binding in Caliburn.Micro
     * Source: https://stackoverflow.com/questions/30631522/caliburn-micro-support-for-passwordbox
     */
    1-Verweis | 0 Änderungen | 0 Autoren, 0 Änderungen
    public static class PasswordBoxHelper
    {
        public static readonly DependencyProperty BoundPasswordProperty =
            DependencyProperty.RegisterAttached(name: "BoundPassword",
                propertyType: typeof(string),
                ownerType: typeof(PasswordBoxHelper),
                new FrameworkPropertyMetadata(defaultValue: string.Empty, OnBoundPasswordChanged));

        2 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
        public static string GetBoundPassword(DependencyObject d)
        {
            if (!(d is PasswordBox box)) return (string) d.GetValue(dp: BoundPasswordProperty);

            // this funny little dance here ensures that we've hooked the
            // PasswordChanged event once, and only once.
            box.PasswordChanged -= PasswordChanged;
            box.PasswordChanged += PasswordChanged;

            return (string)d.GetValue(dp: BoundPasswordProperty);
        }

        1-Verweis | 0 Änderungen | 0 Autoren, 0 Änderungen
        public static void SetBoundPassword(DependencyObject d, string value)
        {...
    }
}

```

2. Adding some lines to the constructor of Bootstrapper.cs

```

1-Verweis | 0 Änderungen | 0 Autoren, 0 Änderungen
public class Bootstrapper : BootstrapperBase
{
    private readonly SimpleContainer _container = new SimpleContainer();
    // Constructor
    0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    public Bootstrapper()
    {
        Initialize();

        // Source: https://stackoverflow.com/questions/30631522/caliburn-micro-support-for-passwordbox
        ConventionManager.AddElementConvention<PasswordBox>(
            bindableProperty: PasswordBoxHelper.BoundPasswordProperty,
            parameterProperty: "Password",
            eventName: "PasswordChanged");
    }
}

```

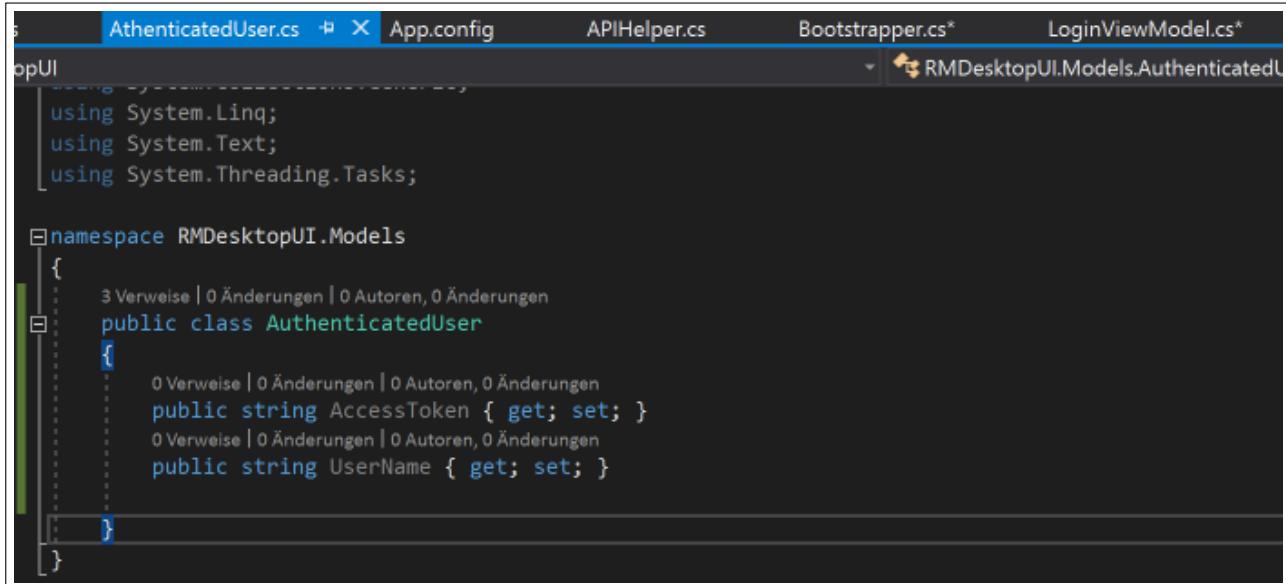
See the example on stackoverflow.com...

11 Wiring up the WPF Login Form

Connecting the login form button to the authentication API endpoint (/token). Gets back the bearer token or an exception if failed.

11.1 Implementing a class AuthenticatedUser.cs

Holds the information for already authenticated users.



The screenshot shows a code editor window with the file "AuthenticatedUser.cs" open. The code defines a class "AuthenticatedUser" with properties for AccessToken and UserName. The code is as follows:

```
opUI
AuthenticatedUser.cs  X  App.config      APIHelper.cs      Bootstrapper.cs*      LoginViewModel.cs*
RMDesktopUI
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RMDesktopUI.Models
{
    public class AuthenticatedUser
    {
        public string AccessToken { get; set; }
        public string UserName { get; set; }
    }
}
```

11.2 Implementing a helper class to handle API call interactions

```

User.cs          App.config        APIHelper.cs*      Bootstrapper.cs*      LoginViewModel.cs*
opUI
RMDesktopUI.Helpers
{
    /*
     * The aim of this class is to handle all API call interactions between the different views and the API.
     */
    2 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    public class APIHelper : IAPIHelper
    {
        4 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
        private HttpClient ApiClient { get; set; }

        0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
        public APIHelper()
        {
            InitializeClient();
        }

        1-Verweis | 0 Änderungen | 0 Autoren, 0 Änderungen
        private void InitializeClient()
        {
            var api = ConfigurationManager.AppSettings["api"];

            ApiClient = new HttpClient {BaseAddress = new Uri(apiString:"")};
            ApiClient.DefaultRequestHeaders.Accept.Clear();
            ApiClient.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue(mediaType:"application/json"));
        }

        2 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
        public async Task<AuthenticatedUser> Authenticate(string username, string password)
        {
            var data = new FormUrlEncodedContent(nameValueCollection: new []
            {
                new KeyValuePair<string, string>("grant_type", "password"),
                new KeyValuePair<string, string>("username", username),
                new KeyValuePair<string, string>("password", password),
            });

            // Getting the token
            using (HttpResponseMessage response = await ApiClient.PostAsync(requestUri:"/Token", data))
            {
                if (response.IsSuccessStatusCode)
                {
                    // ReadAsStringAsync needs Microsoft.AspNet.WebApi.Client (NuGet-Package)
                    var result = await response.Content.ReadAsStringAsync<AuthenticatedUser>();
                    return result;
                }
                else
                {
                    throw new Exception(response.ReasonPhrase);
                }
            }
        }
    }
}

```

11.3 Implementing a Interface IAPIHelper.cs

Needed for dependency injection, in order to add it to the Configure() method in Bootstrapper.cs

```

using System.Threading.Tasks;
using RMDesktopUI.Models;

namespace RMDesktopUI.Helpers
{
    public interface IAPIHelper
    {
        Task<AuthenticatedUser> Authenticate(string username, string password);
    }
}

```

11.4 Adding <appsettings> to App.Config

```

<configuration>
    <appSettings>
        <!--The value here is the project url, which can be found in properties/web.
            Can be changed at any time on runtime. It's possible to create several App.Config overrides.-->
        <add key="api" value="http://localhost:54756" />
    </appSettings>
    <startup>

```

11.5 Adding APIHelper and IAPIHelper to the container in Bootstrapper.cs

```

    _container
        .Singleton<IWindowManager, WindowManager>()
        .Singleton<IEventAggregator, EventAggregator>()
        .Singleton<IAPIHelper, APIHelper>(); // Enables keeping the http-client open
                                                // until the application gets closed

```

11.6 Applying some changings to LoginViewModel.cs

11.6.1 Adding a new private property as a backing field

```

    3 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    public class LoginViewModel : Screen
    {
        /*
         * Backing fields
         */
        private string _userName;
        private string _password;

        private readonly IAPIHelper _apiHelper;
    }

```

11.6.2 Adding a constructor and initializing the property from within

```

/*
 * Constructor
 */
0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
public LoginViewModel(IAPIHelper apiHelper)
{
    _apiHelper = apiHelper;
}

```

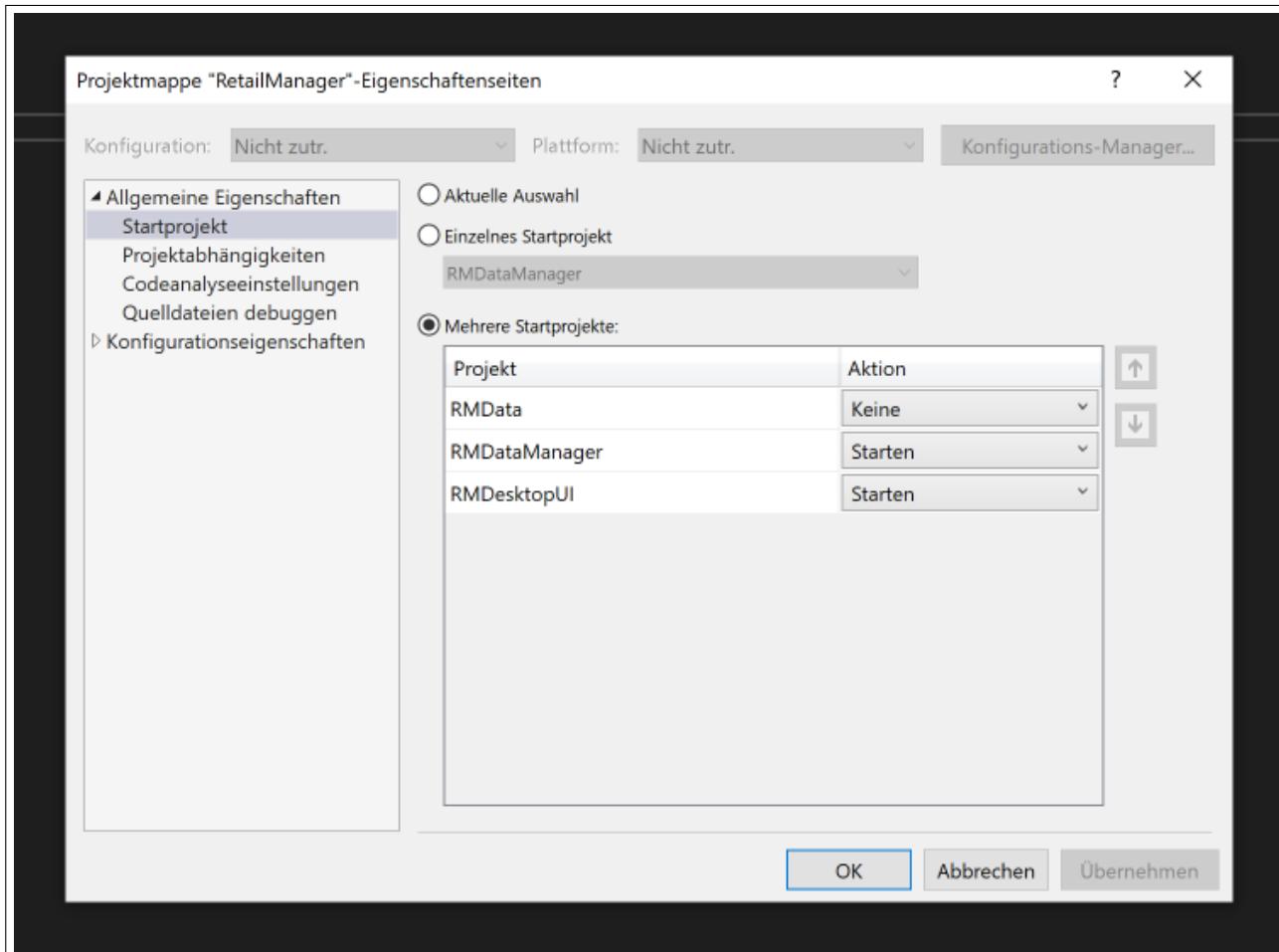
11.6.3 Implementing the Login() method

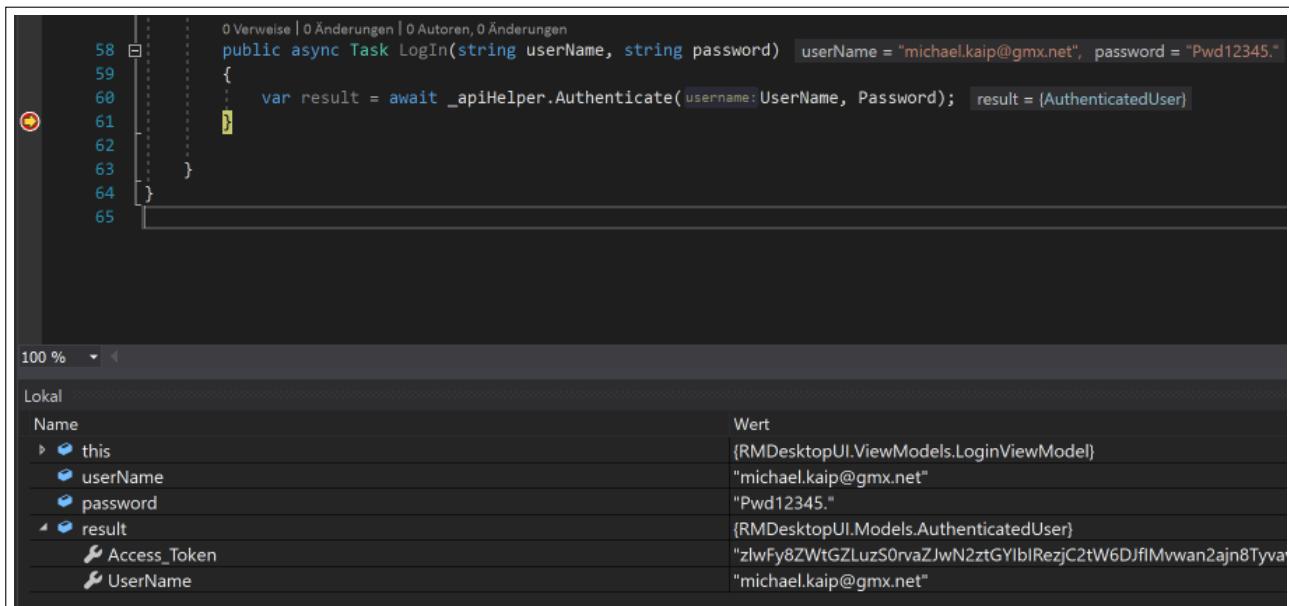
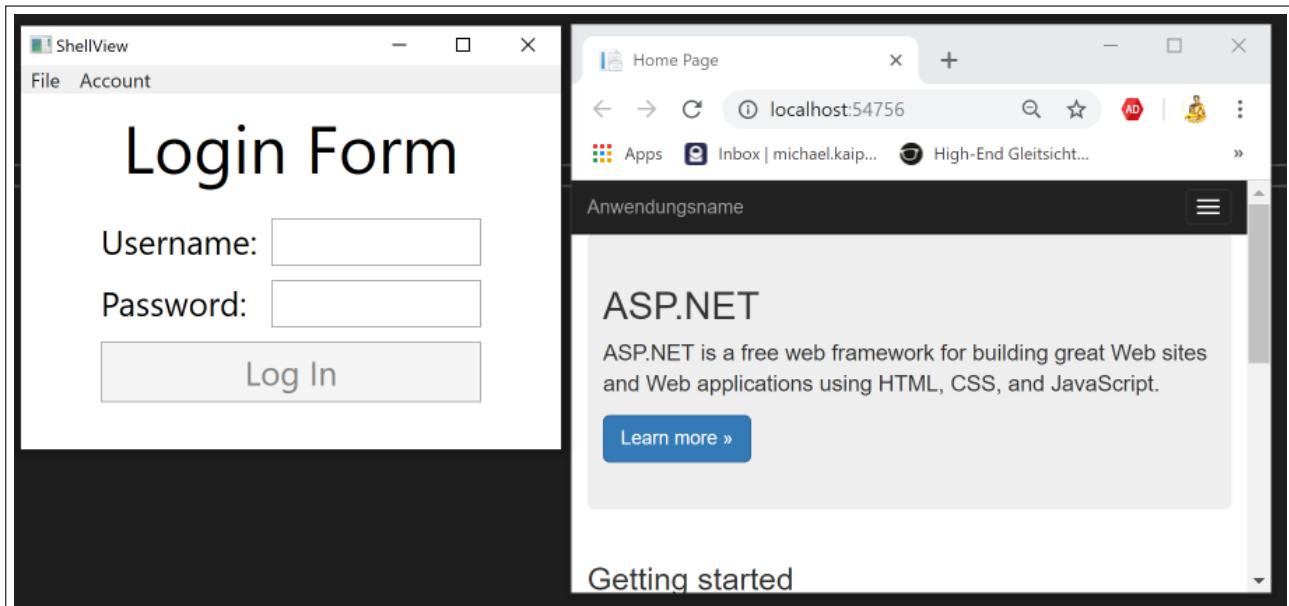
```

0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
public async Task LogIn(string userName, string password)
{
    try
    {
        var result = await _apiHelper.Authenticate(username:UserName, Password);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

```

11.7 Enabling the solution to start multiple projects





12 Login Form Error Handling

12.1 Displaying an login error message within the login form

```

<Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/> <!-- Error Message -->
    <RowDefinition Height="Auto"/> <!-- Username -->
    <RowDefinition Height="Auto"/> <!-- Password -->
    <RowDefinition Height="Auto"/> <!-- Button -->
    <RowDefinition Height="*"/>
</Grid.RowDefinitions>
<TextBlock Grid.Row="0" Grid.Column="1" Grid.ColumnSpan="2"
           HorizontalAlignment="Center" FontSize="48" Margin="0 0 0 20">
    Login
</TextBlock>

<!-- Error Message Row -->
<TextBlock x:Name="ErrorMessage"
           Grid.Row="1" Grid.Column="1" Grid.ColumnSpan="2"
           Margin="0 0 0 20" Foreground="Red" MaxWidth="270"
           Visibility="{Binding IsErrorVisible, Converter={StaticResource BooleanToVisibilityConverter}}"
           FontSize="12" TextWrapping="Wrap"/>

<!-- Username Row -->

```

Through `Visibility` the functionality of collapsing the error message space in case of no error is going to be displayed is added. In case of an error the field expands and the error is going to be displayed. To make it work, first the `BooleanToVisibilityConverter` has to be added to the RessourceDictionary in App.xaml:

```

3     <local:Bootstrapper x:Key="Bootstrapper" />
4   </ResourceDictionary>
5 </ResourceDictionary.MergedDictionaries>
6 <BooleanToVisibilityConverter x:Key="BooleanToVisibilityConverter"/>
7 </ResourceDictionary>
8 </Application.Resources>
9 </Application>

```

Afterwards properties for `ErrorMessage` and `IsErrorVisible` has to be implemented in Login-ViewModel.cs:

```

private string _errorMessage;

3 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
public string ErrorMessage
{
    get => _errorMessage;
    set
    {
        NotifyOfPropertyChanged(() => IsErrorVisible);
        NotifyOfPropertyChanged(() => ErrorMessage);
        _errorMessage = value;
    }
}

/*
 * Properties
 */
1-Verweis | 0 Änderungen | 0 Autoren, 0 Änderungen
public bool IsErrorVisible
{
    get
    {
        var output = ErrorMessage?.Length > 0;

        return output;
    }
}

```

And finally the Login() method has to be changed in the way, that in case of an exception, the exception message is stored into the `ErrorMessage` property:

```

0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
public async Task LogIn(string userName, string password)
{
    try
    {
        ErrorMessage = "";

        var result = await _apiHelper.Authenticate(username:UserName, Password);
    }
    catch (Exception ex)
    {
        ErrorMessage = ex.Message;
    }
}

```

13 Getting User Data

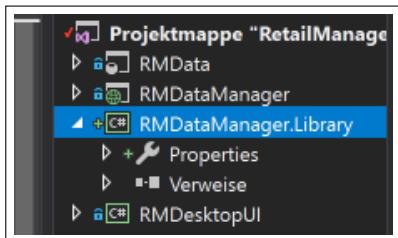
Getting all information about a logged in user from the database. This information is going to be stored in an singleton object and therefore can be used as long as the user is logged in. Once he's logged out, the corresponding object which holds the user information is going to be destroyed.

In order to make it work, following things has to be done:

- implementing a stored procedure
- implementing a model to store the data in the API
- creating an API endpoint
- implementing a method that calls the endpoint from the WPF application
- implementing a model that hold data in the WPF application

13.1 Adding a class library for the API

The aim of this library is to provide data access. It is separate from the WPF application and so it knows nothing about the database, nor have access to it. Putting code not directly into the API enables re-usability.



13.1.1 Installing Dapper

Dapper is an object-relational mapping (ORM) product for the Microsoft .NET platform: it provides a framework for mapping an object-oriented domain model to a traditional relational database. Its purpose is to relieve the developer from a significant portion of relational data persistence-related programming tasks.

Add NuGet-Package to references...

13.1.2 The SqlDataAccess class

```
namespace RMDataManager.Library.Internal.DataAccess
{
    // "Internal" makes sure, that the database can't be accessed
    // from outside this class library. Provides the data for the
    // methods in UserData.cs, which talk to the database.
    0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    internal class SqlDataAccess
    {
        // Gets the connection string with the matching name from Web.Config
        // (DefaultConnection) and returns that connection string.
        2 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
        public string GetConnectionString(string name)
        {
            return ConfigurationManager.ConnectionStrings[name].ConnectionString;
        }

        // Loading data from the database
        0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
        public List<T> LoadData<T, U>(string storedProcedure, U parameters, string connectionStringName)
        {
            string connectionString = GetConnectionString(connectionStringName);

            using ( IDbConnection connection = new SqlConnection(connectionString))
            {
                // Connects to the database and makes a query
                // and returns back a set of rows.
                List<T> rows = connection.Query<T>(oredProcedure,
                    commandType: CommandType.StoredProcedure).ToList();

                return rows;
            }
        }

        0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
        public void SaveData<T>(string storedProcedure, T parameters, string connectionStringName)
        {
            string connectionString = GetConnectionString(connectionString);

            using ( IDbConnection connection = new SqlConnection(connectionString))
            {
                // Stores data into the database
                connection.Execute(storedProcedure, commandType: CommandType.StoredProcedure);
            }
        }
    }
}
```

13.1.3 The UserData class

This class contains methods and the logic on **how** to get data.

1. Adding a stored Prodecure to `RMData.dbo.StoredProcedures`

```
CREATE PROCEDURE [dbo].[spUserLookup]
    @Id NVARCHAR(128)
AS
BEGIN
    SET NOCOUNT ON;

    SELECT Id, FirstName, LastName, EmailAddress, CreatedDate
        FROM [dbo].[user]
        WHERE Id = @Id;
END
```

2. Adding a UserModel class

```
namespace RMDataManager.Library.Models
{
    public class UserModel
    {
        public string Id { get; set; }

        public string FirstName { get; set; }

        public string LastName { get; set; }

        public string EmailAddress { get; set; }

        public DateTime CreatedDate { get; set; }
    }
}
```

3. Adding the UserData class

```

namespace RMDataManager.Library.DataAccess
{
    public class UserData
    {
        public List<UserModel> GetUserById(string Id)
        {
            var sql = new SqlDataAccess();

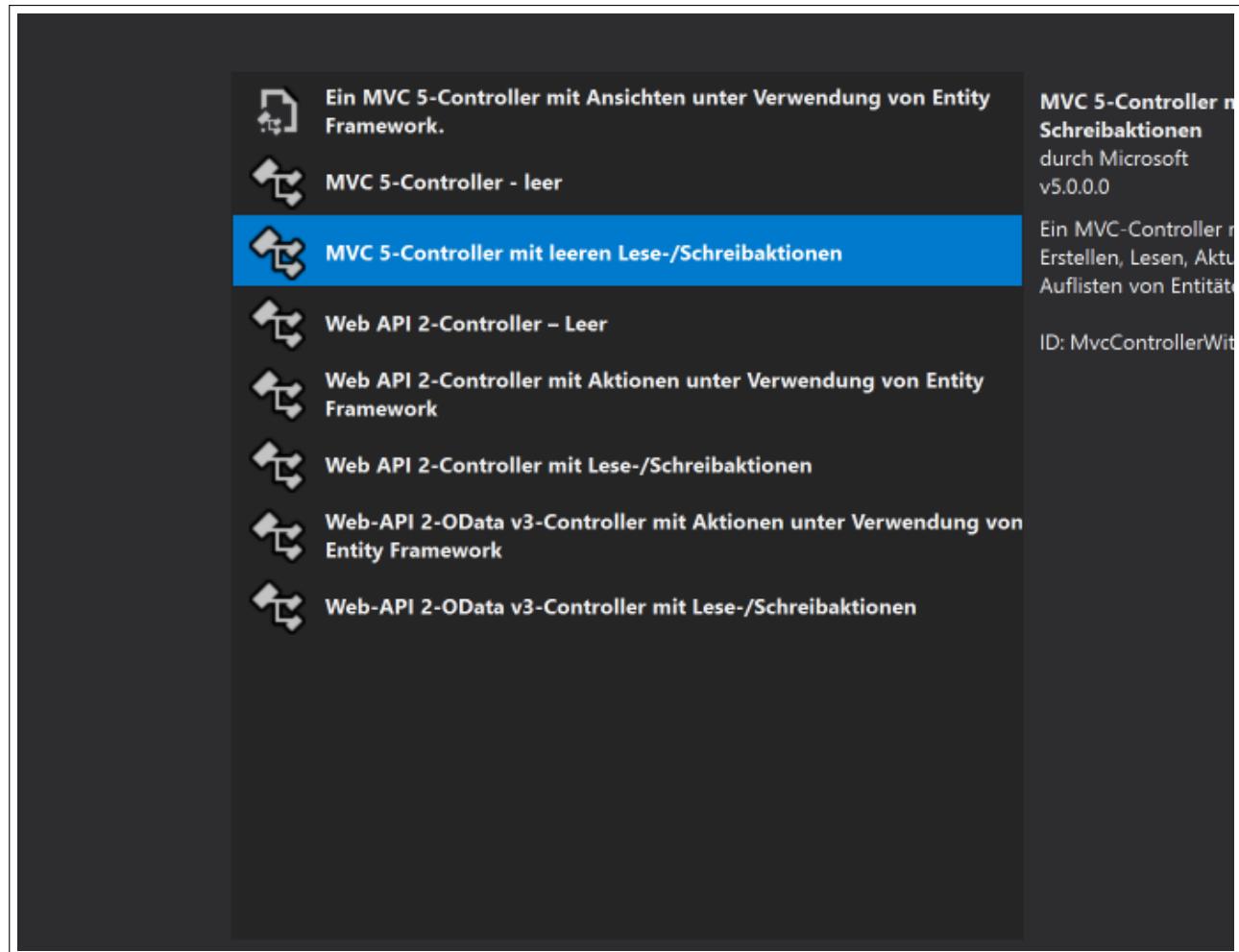
            var p = new { Id = Id }; // An anonymous object (with no named type!)

            var output = sql.LoadData<UserModel, dynamic>(storedProcedure: "dbo.spGetUserById",
                connectionName: "DefaultConnection");

            return output;
        }
    }
}

```

13.1.4 Adding a UserController.cs to RMDataManager.Controllers



Deleting all code in the class, except of this:

```
namespace RMDDataManager.Controllers
{
    [Authorize]
    [RoutePrefix("api/Account")]
    0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    public class UserController : Controller
    {
        // GET: User/Details/5
        0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
        public ActionResult Details(int id)
        {
            return View();
        }
    }
}
```

Implementing a method to get the Id from logged in users:

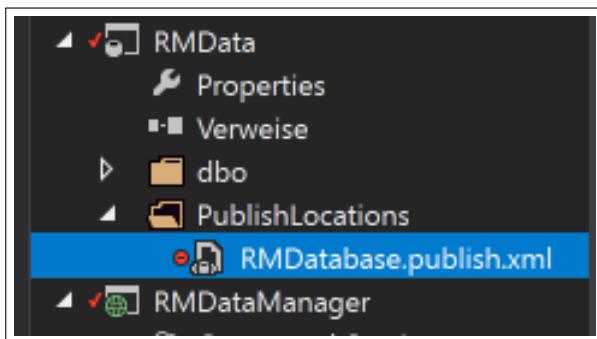
```
namespace RMDDataManager.Controllers
{
    [System.Web.Mvc.Authorize]
    [System.Web.Mvc.RoutePrefix("api/Account")]
    0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    public class UserController : ApiController
    {
        0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
        public string Get(int id)
        {
            return "value";
        }

        // GET: User/Details/5
        0 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
        public List<UserModel> GetById()
        {
            // Getting the Id from logged in users
            string userId = RequestContext.Principal.Identity.GetUserId();

            // Creating a connection to RMDatamanager.Library
            UserData data = new UserData();

            return data.GetUserById(userId);
        }
    }
}
```

Publishing the stored procedure to the database:



Adding a user for testing:

The screenshot shows the Object Explorer in Visual Studio. On the left, it lists several databases and their tables. In the 'aspnet-RMDataManager-20190807034424' database, the 'dbo.AspNetUsers' table is selected and highlighted in blue. On the right, a grid view displays the data for the 'dbo.AspNetUsers' table, showing one row with the following values:

Id	FirstName	LastName	EmailAdress	Creat...
8b9e32c2ea75	Michael	Kaip	michael.kaip@gmx.net	NULL
NULL	NULL	NULL	NULL	NULL

Now that there's an entry in the user database, the application can be started and use SWAGGER to run and test the the GET command in `Values/GET`:

14 Sales Page Creation

15 Event Aggregation in WPF

16 Displaying Product data

17 Wiring up WPF Shopping Cart

18 Modifying SQL, the API and WPF to add Taxes