API: Report RegistrationManager

|  |  |
| --- | --- |
| **COVER AND CONTROL PAGE OF DOCUMENT** | |
| Project Acronym | Panna: RegistrationManager API |
| Project Full Name | Report about the functionality of the RegistrationManager API |
| Storage | Google Drive Document |
| Format | PDF, docx |
| Dissemination | All Working Members |
| Start Date | 01/August/2016 |
| Iteration | 1 |
| Version | 0.01 |
| Responsible | Vitali Dettling |
| Email | vitali.dettling@gmail.com |

Table of Content

[1. Document History 4](#_Toc466379854)

[2. Glossary 5](#_Toc466379855)

[3. Introduction 6](#_Toc466379856)

[1. Motivation and Vision 6](#_Toc466379857)

[2. Problem Statement 6](#_Toc466379858)

[4. Technical Details 8](#_Toc466379859)

[1. Architectural Design and Frameworks 8](#_Toc466379860)

[2. External Licenses 8](#_Toc466379861)

[3. Implementation Information 9](#_Toc466379862)

[4. Test setup and Scenarios 10](#_Toc466379863)

[5. Setting up and using the Service 10](#_Toc466379864)

[6. References 11](#_Toc466379865)

# Document History

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Changes** | **Author(s)** |
| 0.01 | 16/September/16 | Initialization of the document: RegistrationManger API | Vitali Dettling |
| 0.01 | 16/September/16 | Init: Introduction | Vitali Dettling |
| 0.01 | 16/September/16 | Init: Technical Definition | Vitali Dettling |
| 0.01 | 16/September/16 | Init: Setting up and using the Service | Vitali Dettling |
| 0.01 | 08/November/16 | Adding the xUnit framework license | Vitali Dettling |
| 0.01 | 08/November/16 | Init: Test setup | Vitali Dettling |
|  |  |  |  |

# Glossary

| **Term** | **Explanation** |
| --- | --- |
| API | Application Programming Interface |
| REST | Representational State Transfer is a stateless service (loosely coupled system) which means it does not store information about any user. |
| App | Application |
| CDN | Contend Delivery Network |
| Login | Resource of the API, in order to verify that a user exist in the DB. |
| Logout | Resource of the API, in order to add a new user to the DB. |
| DB | Database |
| CMD | Command Prompt |
| CD | Change Directory |
|  |  |
|  |  |

# Introduction

There are two main objections in this report:

* Firstly, it shows the functionality of the RegistrationManager service.
* Secondly, the report points out the most important parts of the implementations, as well as the technologies which were used.

Moreover one should look into the source code and to read other reports about the service. The comments within the source code are important as well and thus, this repost will not cover all details again, if they are already mentioned in the source code or at another place.

## Motivation and Vision

The motivation behind the RegistrationManager REST API is that it should be possible to register a user. Moreover, it should be possible to do so with different application, e.g. webpage, mobile app, client application, etc. Therefore, and because of the nature of REST APIs, the RegistrationManager API should not store any state of a current user. Last but not least, the HTTP requests are limited to: GET, POST, PUT, and DELETE.

The API itself should be deployed in the Azure cloud environment. The environment should provide abilities, namely:

* Monitoring the API
* Load balancer, in case an error occurs in one of the services
* Authorization possibilities, in order that only authorized user can use it
* Access control for resources
* Server discovery mechanism
* The cloud provider should detect when a service is down

## Problem Statement

A list about the problems the service has to face:

|  |  |
| --- | --- |
| **Security** | User Authentication with at least email and password.  Authorization of an app, in order that the service can be used. |
| **Performance** | Of course the login should be as fast as possible. Thus multiple instances have to be deployed across the globe. |
| **Availability** | The service should be as robust as possible, so that it runs without interruption, even if the multiple user try to register. |
| **Reliability** | Even if a service fails more instances of the servicer should exist, in order to catch the further registrations. |
| **Scalability** | The service must be run in multiple instances. Moreover, it should be possible to have multiple instances at pick times. |
| **Maintainability** | The service is at any times easy to adjust to certain circumstances, as well as to be extended with further registrations options. |
| **Changeable** | In case major changes occur, such as a new framework is introduced, the service needs to be adjusted accordingly and fast. |
| **Efficiency** | The resources of the API should response within 2 second at most, with appropriate response messages. |
| **Usability** | In this context the usability means that any kind of application, written in any language, can use the service, as easy as possible. |

# Technical Details

This section is about the technologies, middleware and frameworks which had been used in order to develop the service.

## Architectural Design and Frameworks

The architectural design is based mostly on the MVC pattern. Because it is not supposed to be highly visible, like a homepage; hence the View folder only contains the index file, in order to show the API resources via the Swashbuckle (Swagger) framework. The Model folder on the other hand contain all required data contains which are required from the service. The most important folder is the controller folder. It contains all the API resources.

The main framework which was used is the ASP.NET Core v1.0 framework. This is the case because the RegistrationManager is a REST based service. The Core statement express that it is a redesign and reimplementation of the original ASP.NET framework. Hence, not all information and tutorials will work for the ASP.NET Core framework, as it may had done for ASP.NET. Furthermore, it is a lightweight framework in comparison to the original one. Not to mention, it is cross platform and open source.

Finally, all used frameworks can be found at the project.json file under “dependencies”, within the RegistrationManager implementation.

## External Licenses

The section lists all external tools and software which were used in order to implement and document the API. They can be included directly within the RegistrationManager API project in the project.json file, but they don’t have to be. Furthermore, software which they were used additionally are also committed (External folder) with the rest of the code. Finally, Microsoft license have not been included because they are supposed to be free of charge.

List of licenses:

|  |  |  |
| --- | --- | --- |
| **Name** | **License** | **Link** |
| Swagger | Apache License | http://swagger.io/license/ |
| Swashbuckle | Copyright (c) 2013, Richard Morris | https://github.com/domaindrivendev/Swashbuckle/blob/master/LICENSE |
| Doxygen | GNU General Public License | https://github.com/doxygen/doxygen/blob/master/LICENSE |
| dotnet ef (Entity Framework) | Apache License, Version 2.0 | https://github.com/aspnet/EntityFramework/blob/dev/LICENSE.txt |
| TexLive | GNU General Public License, LGPL, BSD license, X license | https://www.google.de/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&ved=0ahUKEwiPjrmGv5jPAhUQz2MKHRZ3D7UQFggkMAE&url=https%3A%2F%2Ftug.org%2Fmactex%2Fsrc%2FLicense.rtf&usg=AFQjCNGF8wzfNnnDQiqlq937LP\_4QjhU1Q&sig2=ByG9eRywanA3e2RXA6KccA&cad=rja |
| xUnit | Apache 2 | https://opensource.org/licenses/Apache-2.0 |
| Moq | Unknown | ??? |

## Implementation Information

The implementation is mostly done via the MCV pattern and via the ASP.NET Core framework. Nevertheless, the DB migration was done via the dotnet ef (Entity Framework). It is a tool to generate DB classes (Migrations folder) which will then generate a DB schema at the Azure cloud. This, of course, implies that a SQL server is already running within the Azure cloud in order to apply the DB schema. It goes without saying that the generated migration classes should not be changed by hand, only with the dotnet ef framework tool.

Nevertheless, the dotnet ef framework has three main operations:

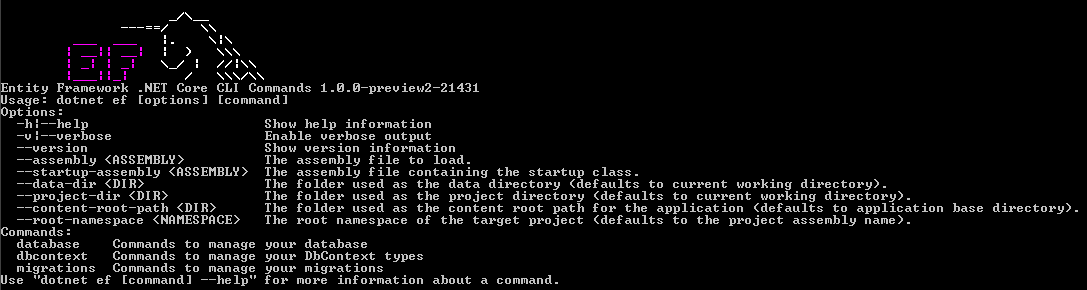
dotnet ef migrations add {MigrationName}

dotnet ef migrations remove

dotnet ef database update

Further, information about the framework can be found here [1] and here [2]:

Anyway, the directory to user the tool is the src folder within the project. In order to do so, simply open and CMD in the src folder and type “dotnet ef”, a window should occur as follows:



As mentioned above all migration classes will be generated into the Migrations folder, and the Migrations folder is part of the Data folder. On the other hand, the Data folder contains all classes which are required for the DB. Therefore, one should read the comments within the code, as well as all generated reports. This is the case, because the classes and methods are already covert there and it won’t be done here again.

## Test setup

In order to create a test suit, one has two possibilities. One of the possibilities can be found at the official XUnit framework website [3]. The other possibility is shown next which was also used by the RegistrationManager project.

First of all one has to create a test folder like this:

RegistrationManager

|

|- src

|

|- test

Now, CD into the test directory and open up a CMD and type the following commands:

dotnet new -t xunittest

dotnet restore

These two commands will, first of all create a preconfigured project.json file with all required dependencies, and a “test.cs” class. The second command updates the xUnit framework, to the newest version. Of course, one has to add the new existing method to the already existing solution. The best way to do that is like this:

src -> add -> Existing Project…

Moreover, the project uses a lot of constructor dependency injections. Hence, it is indispensable to use a mockup framework. In this case the Moq framework is being used. How to setup the framework can be found [4] and also in the next steps. Although, keep in mind it is a beta version and not approved by Microsoft as a default mockup framework.

The image shows the steps for the configuration of the Nugget package manager, in order that it will find the dependencies:



The steps are quite straight forward. Open the window by means of the setting symbol at the right hand side corner. Further, click the plus symbol and fill out the name and source:

Name: aspnet-contrib

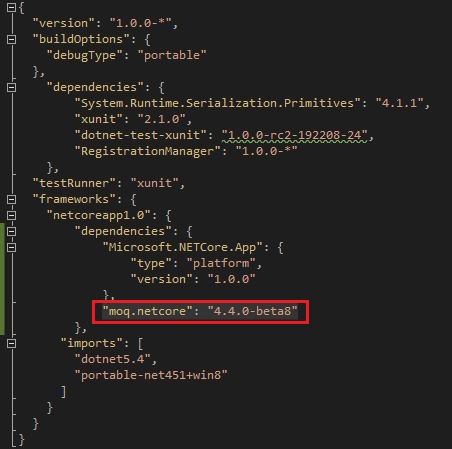
Source: https://www.myget.org/F/aspnet-contrib/api/v3/index.json

Last but not least press the Upload button. The result should look like as the image suggests.

One more step needs to be done, namely the project.json file itself. Add the required dependency within the framework field:

"moq.netcore": "4.4.0-beta8"

Also shown in the following image:



# Setting up and using the Service

ToDo!

# References

|  |  |
| --- | --- |
| [1] | Jérémie Bertrand (April, 2016): “Dotnet EF Migrations for ASP.NET Core”  Publisher: BenCull’sBlog; Accessed: 18/September/2016  <http://benjii.me/2016/05/dotnet-ef-migrations-for-asp-net-core/> |
| [2] | Microsoft (April, 2016): “Entity Framework Core”  Publisher: Microsoft; Accessed: 18/September/2016  <https://docs.efproject.net/en/latest/intro.html> |
| [3] | xUnit.net (2016): “Getting started with xUnit.net (.NET Core / ASP.NET Core)”  Publisher: .NET Foundation; Accessed: 08/November/2016  <https://xunit.github.io/docs/getting-started-dotnet-core.html> |
| [4] | .Net LIBERTY (2016): “Se/November/tting up Moq on .NET Core”  Published: Armen Shimoon; Access: 08/Novermber/2016  <http://dotnetliberty.com/index.php/2016/02/22/moq-on-net-core/> |