Contents

[1. General Information 1](#_Toc179441502)

[2. Setup 1](#_Toc179441503)

[3. Sprint 1 1](#_Toc179441504)

[3.1 IDE Setup 1](#_Toc179441505)

[3.2 Docker Containerization 2](#_Toc179441506)

[4. Sprint 2 2](#_Toc179441507)

[4.2 Nginx 2](#_Toc179441508)

[4.3 Project Structure Adjustments 2](#_Toc179441509)

[5. Sprint 3 3](#_Toc179441510)

[5.1 Entities 3](#_Toc179441511)

[5.2 Postgres 3](#_Toc179441512)

[5.3 Postgres Container Issues and Solutions 3](#_Toc179441513)

[5.4 DTOs and Mapping 3](#_Toc179441514)

[5.5 Unit Tests for DTOs and Mapping 3](#_Toc179441515)

# 1. General Information

# 2. Setup

Before starting with the first Sprint, following things had to be checked:

* IDE like Visual Studio or JetBrains Rider installed?
* Docker Installed?
* Git installed?

We both decided to use JetBrains Rider, since we are very familiar with their software and simply prefer using it over Visual Studio.

# 3. Sprint 1

## 3.1 IDE Setup

When creating the project, we opted for using the APS.NET Core Web API template in Rider, which provides a swagger UI out of the box. We ditched the swagger UI and just printed sample documents for Sprint 1.

## 3.2 Docker Containerization

After creating the Dockerfile for this application, the command was used to build the docker image for the container:

*docker build -t document-management-system .*

‘document-management-system’ is the name of the image.

This command is used to run the docker container:

*docker run -d -p 8081:8081 --name dms\_container document-management-system*

*‘-p 8081:8081’*  lets the docker container run on port 8081 and also exposes it to port 8081, where it is being accessible in the browser. ‘dms\_container’ is the name of the container.

Using a ‘*docker-compose.yml’*  file makes coding with Docker a lot easier. Since you can just type ‘*docker-compose build’* and ‘*docker-compose up’* to practically build and run the containerized application.

# 4. Sprint 2

## 4.2 Nginx

## 4.3 Project Structure Adjustments

After taking a look at the sample code from our lecturer, we decided to apply their project structure to ours to some degree. We added following folders:

* **Controllers**: MVC architecture, Controllers are basically the URL
* **DTOs**: Data Transfer Objects, carries data between ‘processes’
* **Entities**: Model-classes for the data
* **Repositories** (later in DAL): Defines queriers following a pattern for the API

# 5. Sprint 3

## 5.1 Entities

Entities define the data structure / model and properties of objects used in the application. For example, ‘Document’ consists of:

* (Id)
* Name
* Path
* File\_Type

## 5.2 Postgres

In our *.yml* file we added the postgres container with following parameters:

* **User**: dms\_user
* **Password**: dms\_password
* **Database**: dms\_db

The connection string for the postgres connection with the application is stored in the *appsettings.json* and *appsettings.Development.json* files on the DAL. We created a DocumentContext file, which is used to create the database and tables. In order to register the database with the table ‘Document’ we needed to execute 2 commands (after docker compose up):

* dotnet ef migrations add InitialCreate
* dotnet ef database update

These commands create a directory called ‘Migrations’ with 2 files in it and creates the database and table ‘Document’ in the postgres container’s database.

## 5.3 Postgres Container Issues and Solutions

Upon starting the container the first time, everything went well. After the initialization we changed the parameters to the ones specified above. We didn’t delete the volume for postgres, that’s why the initial parameters were still stored in postgres and our new user wasn’t created.

## 5.4 DTOs and Mapping

The DTO ‘DocumentDTO’ was created in the REST project. Validation rules for the properties of the the dto classes were defined afterwards. In case of the ‘Document’ entity, name, path and file\_path are all required in order to pass the validation. Additionally, a mapping class was also created. This is used to create the appropriate mapping between the entity ‘Document’ (DAL) and the dto ‘DocumentDTO’ on the REST side.

## 5.5 Unit Tests for DTOs and Mapping

On new project for Unit Test was added and 2 test files were created for testing the dtos and mapping.

## 5.6 Adjustment to project structure

Previously we had a documentcontroller class which called an HTTP Client on DAL. This approach was changed after asking our lecturer for their expertise.  
Currently we have:

* REST:
  + DocumentController (API)
* DAL:
  + DocumentService (Business Logic / Layer)
  + DocumentRepository (Communication with postgres)
  + RabbitMQPublisher (used to happened in REST)

With this approach the entry point for users (rest api) does not expose code / services directly, since they are in DAL.

# 6. Sprint 4 / Mid-Term

## 6.1 RabbitMQ

The container was configured and tested. The web interface can be accessed under port 9093. After researching how to implement the queue in .NET, the actual coding was straightforward and did not present any obstacles.

We implemented a RabbitMQPublisher class, which allows us to pre configure the queue and easily push data into the queue by using the Publish() method with the payload (in this case the path of the document) and the routing\_key for the queue. The Configuration data is in appsettings.json and is being fetched and setup in Program.cs, where the queue currently runs in a Singleton.

## 6.2 Logging

We used log4net and downloaded the NuGet package for almost all projects. Implementing log4net was also very easy and we also started to implement some error and exception handling.

## 6.2 Unit Tests test coverage

Since we are required to reach a test coverage of at least 70%, we tried to implement unit tests accordingly. Recently implemented unit tests are:

* DocumentServiceTests
* DocumentControllerTests (adjustments and additional tests)

Currently we achieved a test coverage of 82%.

# 7. Sprint 5