|  |  |
| --- | --- |
| A picture of a winding road and trees  Online shop Api DOCUMENTATION  Information, Build, Test | Abstract  In this document you will find how the project is structured, the information about how to build and run tests with coverage reports as well as Sonarqube analyse report. Also assumptions are included while developing the software.  Kemal Baytekin  Tech Lead |

Contents

[Solution Structure 3](#_Toc94618677)

[Domain.Shared Project 4](#_Toc94618678)

[DatabaseLayer - EntityFrameworkCore 4](#_Toc94618679)

[DataLayer - Repository 4](#_Toc94618680)

[Domain.Entity 5](#_Toc94618681)

[Domain 5](#_Toc94618682)

[HttpApi 5](#_Toc94618683)

[How to Run? 5](#_Toc94618684)

[Pre-requirements 5](#_Toc94618685)

[To Run 5](#_Toc94618686)

[To Test 6](#_Toc94618687)

[UML Diagrams for Key Classes 7](#_Toc94618688)

[Communication between microservices 7](#_Toc94618689)

[Assumptions 7](#_Toc94618690)

[Authentication and Authorization 7](#_Toc94618691)

[Mailing 7](#_Toc94618692)

[Update Assumption 7](#_Toc94618693)

[Api List Paging 7](#_Toc94618694)

[Logging 7](#_Toc94618695)

[Sqllite 8](#_Toc94618696)

[Tests 8](#_Toc94618697)

[Stress Tests 8](#_Toc94618698)

[External Packages 8](#_Toc94618699)

[AutoMapper 8](#_Toc94618700)

[MassTransit 8](#_Toc94618701)

[MediatR 8](#_Toc94618702)

[Swagger 8](#_Toc94618703)

[EntityFrameworkCore for SqlLite 8](#_Toc94618704)

[MongoDB 8](#_Toc94618705)

[Moq 8](#_Toc94618706)

[Xunit 8](#_Toc94618707)

General Structure

To use common classes, structures across the solutions for microservices a Shared folder is generated and common solution is builded inside as TheCompany.Shared. Solutions for individual microservices will use these projects using nuget packages. There are 3 nuget packages:

1. TheCompany.Domain.Entity.Common

Common entities and abstract entities like abstract EntityBase class or IaggregateRoot interface can be found in this project.

1. TheCompany.Domain.Shared.Common

Shared classes or interfaces can be found in this project. Iıdgenerator helper class or QueueMessaging POCO classes may be a good candidates of this project.

1. TheCompany.EntityFrameworkCore.Common

The repository pattern is used in the project. So a generic RepositoryBase class may be an example for this project.

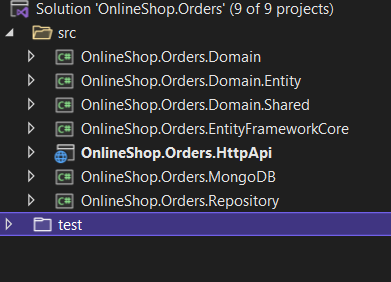
All the projects are registered to nuget and may be found by the given names.

Individual modules/microservices are included in Modules folder. There are 3 modules in the project.

1. OnlineShop.Customers
2. OnlineShop.Products
3. OnlineShop.Orders

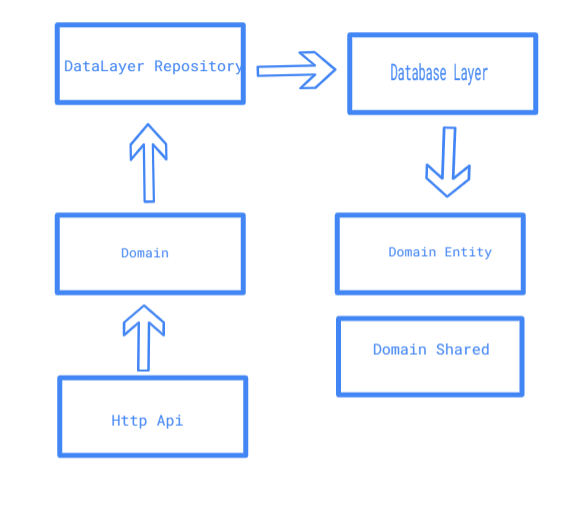
# Solution Structure

Projects are organized in src and test folders. src folder contains the actual application which is tried to be layered based on DDD principles.



Şekil 1 Solution structure

The diagram below shows the layers & project dependencies of the application:



Şekil 2 Project dependencies

## Domain.Shared Project

This project contains constants, enums and other objects these are actually a part of the domain layer, but needed to be used by all layers/projects in the solution.

A CustomerGender enum and a IOrderRepository interface are good candidates for this project.

This project has no dependency to other projects in the solution. All other projects depend on this directly or indirectly.

## DatabaseLayer - EntityFrameworkCore

This is the integration project for the EF Core. It defines the DbContext and configure context objects. SqlLite is used for development purposes.

Depends on the Domain.Entity project to be able to reference to entities.

At Entity Framework data-db configurations being applied in the OnModelCreating method, but it is recommended to separate configurations out to individual files per entity - especially for larger models or ones that require a lot of configuration. On the project in Configurations folder you can see the configuration files for entities. Extension folder has the dbcontext service registration code to services to use later on for dependency injection for repository layer.

## DataLayer - Repository

This is the seperation layer project between DataLayer and Domain/Application Layer. It implements repository interfaces defined in the Domain.Shared project.

Repository Design Pattern is used, which is one of the most popular design patterns to achieve such separation between the actual database, queries, and other data access logic from the rest of the application.

Extension folder has the RepositoryWrapper service registration code to services to use later on for dependency injection for the application layer.

Depends on the Domain.Shared project to be able to reference to entities and repository interfaces. Also depends on the DataLayer to execute db/persistence operations.

Only for Orders repository module MongoDB is also used for querying the data scenarios. So the repository layer (for orders only) also depends on the MongoDB project.

## Domain.Entity

This project contains the entities or aggregate roots used in the domain. Customer entity may be an entity example on the other hand Order class may be a good aggregate root example.

Depends on the Domain.Shared project that Order class contains OrderStatus enum.

## Domain

This project contains the domain service implementations which is used by the httpapi project (or other projects that will consume this domain later). CQRS pattern is used in this project. Using MediatR nuget package, the mediator is simplified for code readability and reusability. Also using Automapper nuget package mapping operations between the db layer objects to view model objects are simplified.

Depends on the DataLayer.Repository project to be able to use domain objects (entities, repository interfaces... etc.) to perform the application logic.

For microservice communication RabbitMQ is used by the help of an enterprise bus MassTransit. I do not mention too much about the enterprise buses but they help us about exception handlings, retries, message routing etc. Or if we want to change out queue message tool from RabbitMQ to another tool like ActiveMQ, Amazon SQS/SNS we do not need to rewrite the code.

## HttpApi

This project is used to define API Controllers for Restful api implementation. OrderController and ProductController are good candidates for this project.

Depends on the Domain project to be able to inject the application service interfaces.

To handle all kind of errors a middleware ExceptionMiddleware is also added to project.

# How to Run?

### Pre-requirements

You must have .net Core 6 is installed on your computer and may have visual studio 2022 or later or you may have visual code as well to run the application. Since the machine that I build has windows 8.1, docker is supported by windows 10. So I can not create a docker container but the docker file is ready in the HttpAPi projects.

### To Run

To build the project with MsBuild, just run the msBuildSln.bat in the solution folder. It will build and publish the project to publish folder in the solution folder.

If you want to run the solutions from Visual Studio:

Set HttpApi as the startup project and run the application. A Swagger UI will be opened. You can do operations like creating new Customers, Products, Orders or getting the data. You can also do update, delete operations for both items. On the other hand you can query and item by id ((GET) /api/orders/{id}) or you can get list of items by filtering the start and end date. System will not return the result with a paging structure, will return the all items in the database. Swagger UI usage is not explained on this document.

### To Test

If you want to run the tests and want to see the test coverage report you can run run-tests.bat in the test folder. It will run the tests and creates a coverage report with a html output. Bat file will also automatically open coverage report in the browser. If we look inside the bat file:

1. dotnet test ..\OnlineShop.Orders.sln --collect:"Xplat Code Coverage"

It will executes the unit tests and exports an output to TestResults folder in the test project with a format of cobertura.xml.

1. tools\reportgenerator.exe ^ "-reports:\*\*\coverage.cobertura.xml" ^ "-targetdir:CoverageReport" ^ -reporttypes:HTML;HTMLSummary

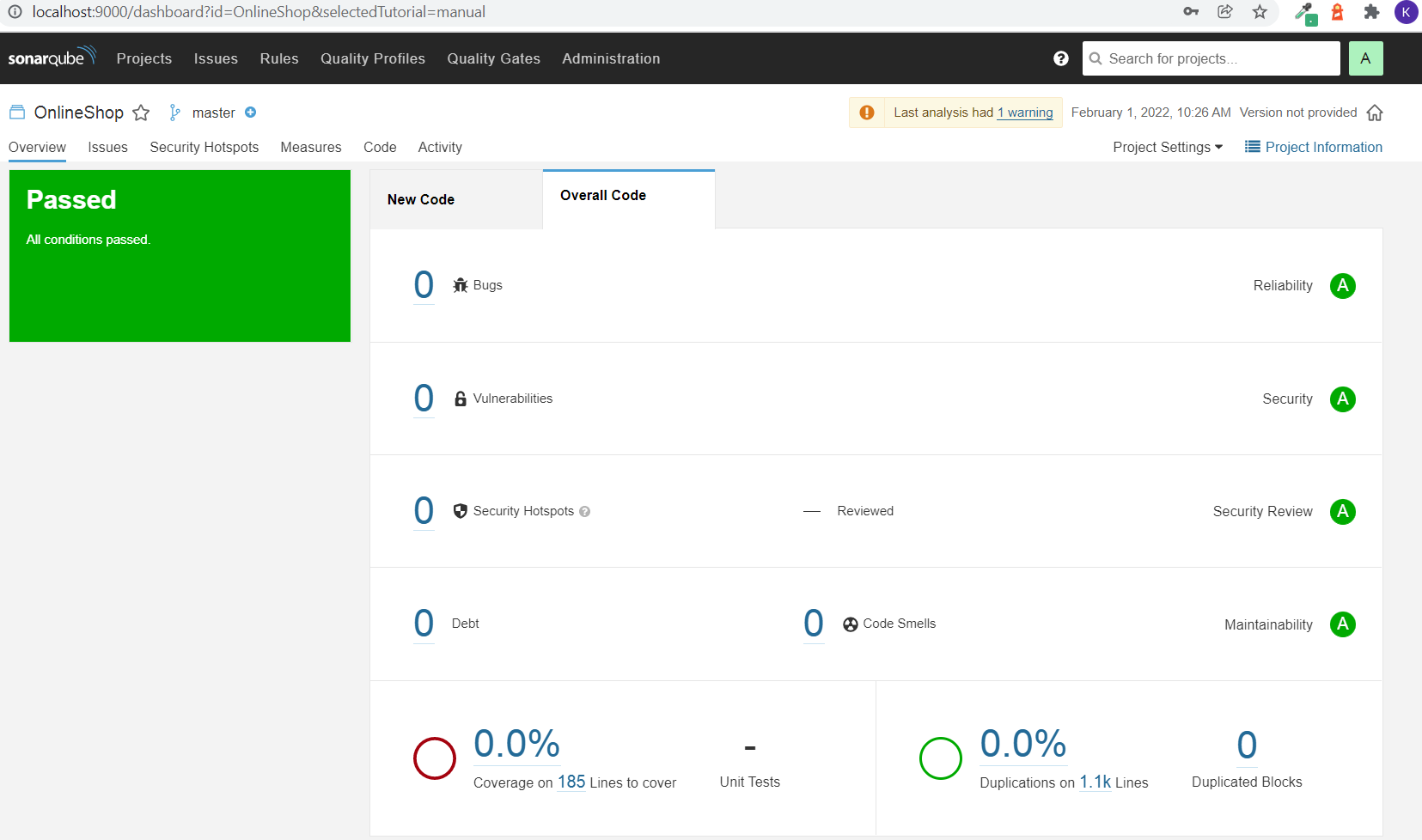
Reportgenerator generates the html output report based on the cobertura xmls in all the test projects to CoverageReport folder. (ReportGenerator may be found on nuget)

1. start CoverageReport\index.htm

Will show the coverage report in the browser.

SonarQube report

Sonarqube report is also generated. It can be shown below.



Şekil 3 Sonarqube report

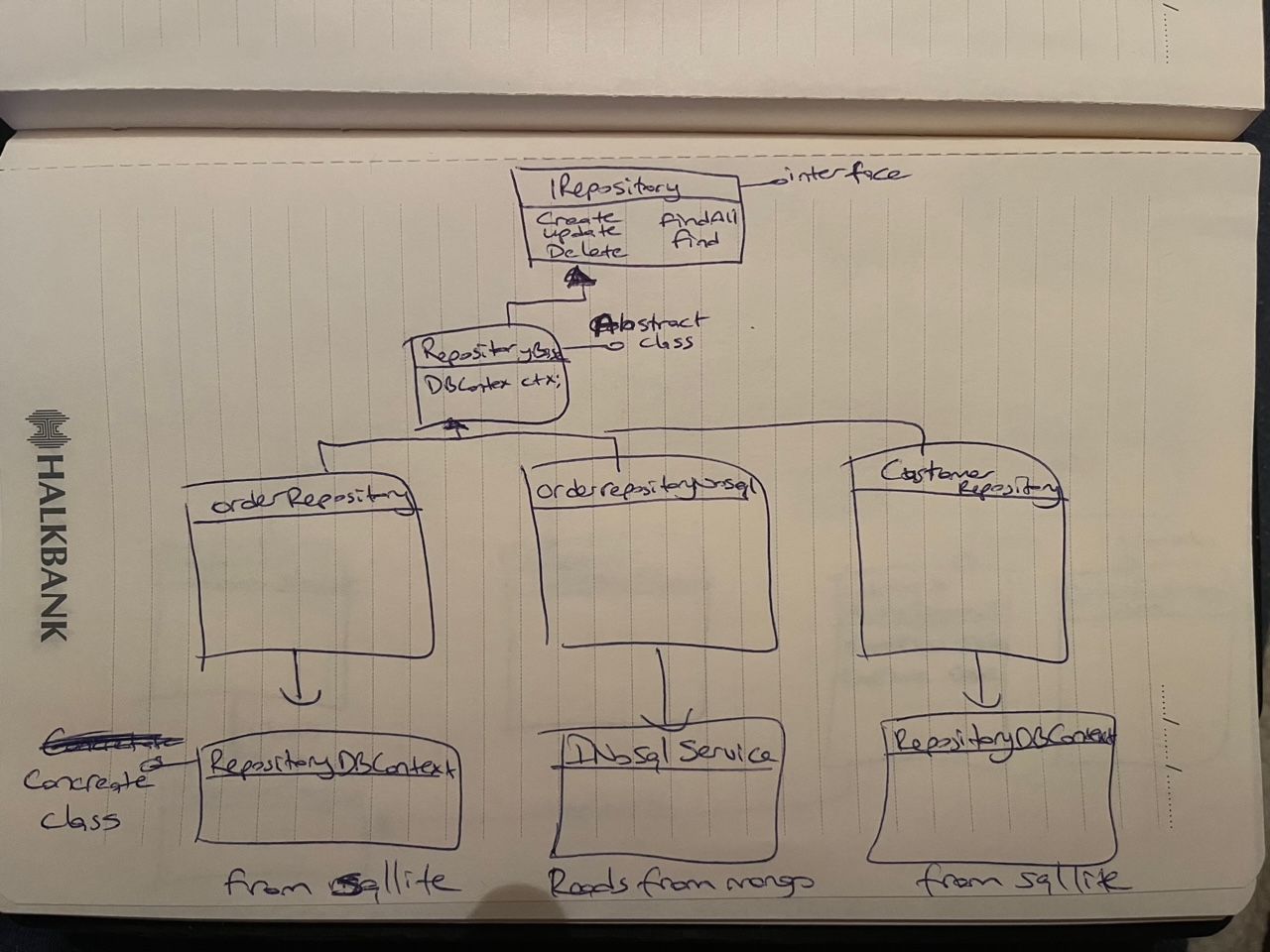
To generate sonarqube report:

1. <https://docs.sonarqube.org/latest/setup/get-started-2-minutes/> follow the commands in this link.
2. For .net core projects, the screen will create a command like these and must be executed in package manager console(sonar.login=<token> will be generated if you follow 1. Option commands):
3. dotnet sonarscanner begin /k:"OnlineShop" /d:sonar.host.url="http://localhost:9000" /d:sonar.login="5424a1f7ac0fcde737e6cc0eed3e5b649fc6c3d7"
4. dotnet build
5. dotnet sonarscanner end /d:sonar.login="5424a1f7ac0fcde737e6cc0eed3e5b649fc6c3d7"

# UML Diagrams for Key Classes

The key classes in the solution as uml diagram as follows:

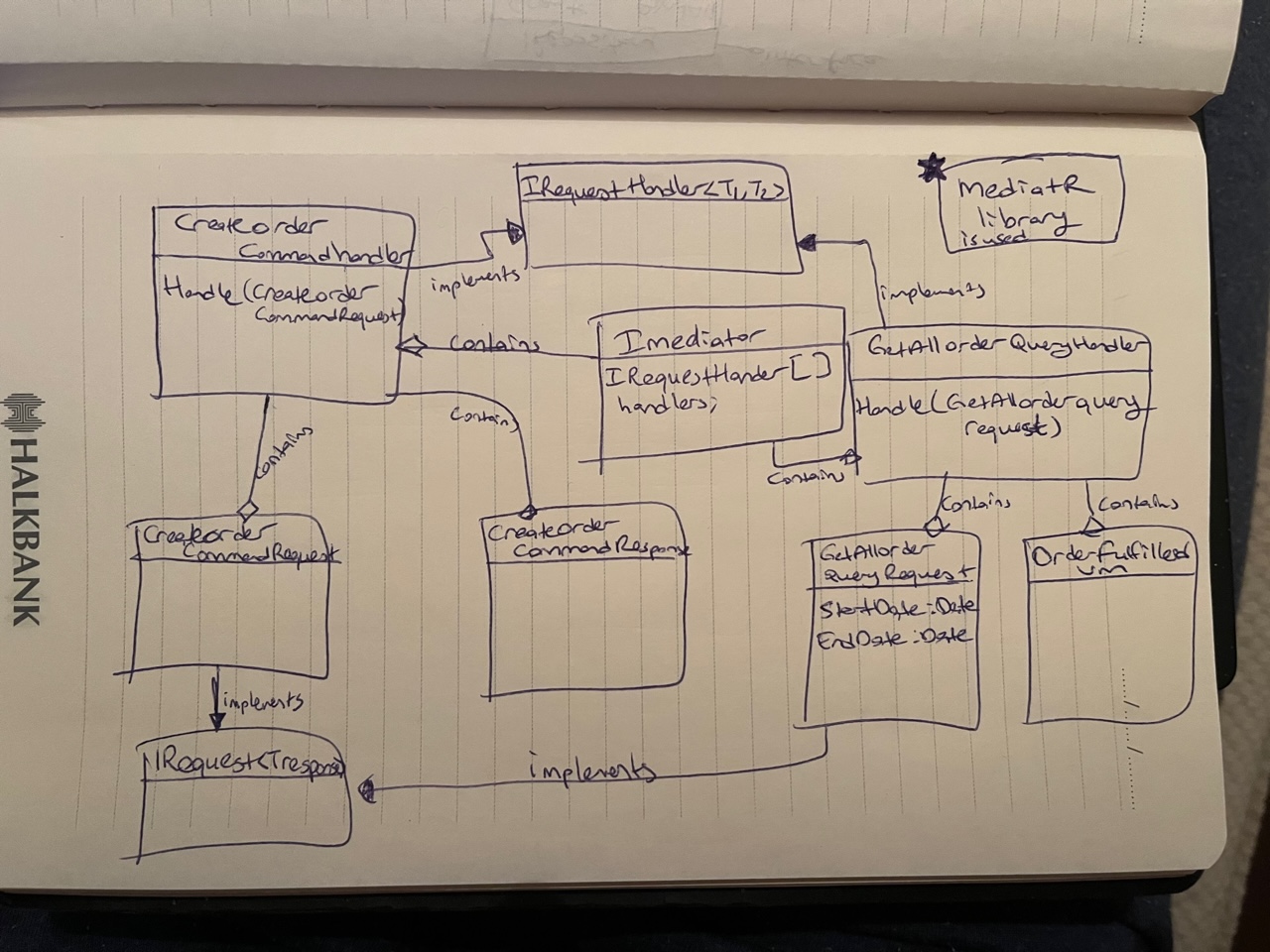
The repository pattern



Şekil 4 Repository Pattern

The CQRS pattern

For orders solution/microservice, commands and queries are seperated by the code and the database infrastructure also. For commands(create, update, delete) sqllite is used but for queries MongoDB is used.



Şekil 5 CQRS Pattern

# Communication between microservices

There are two communication skills in this projects:

1. Customer and product info communication with order microservice

Customer and product microservices creates an event whenever a customer or product is changed and they send the event data to the messaging queue. That messaging queue is listened by the order microservice and creates or updates the customer/product in its boundary context. (customer/product info in the customer/product boundary context may/must be different than the order boundary context. For example customer may have a field of father name in the customer context but it makes no sense in the order context, so that info must not be in order boundary context.)

1. Order created and order query internal communication

Whenever an order is created it produces an order created event and pumps it to messaging queue. Same microservice listens that messaging queue and creates the fulfilled order (with customer main data and products main data) in the mongo db so the prepared fulfilled json order data is ready to be queried by the client.

NOT: Both RabbitMQ and MongoDB are supplied in the cloud enviroment. You do not need to install a tool for that purposes. RabbitMQ is on cloudamqp.com and MongoDB is on mongodb.com.

# Linting

.Net 6 has an internal code analyse after .net 5. But addition to that analyzer SonarLint is used in the project. No warning and no information is considered as a successful build.

# Assumptions

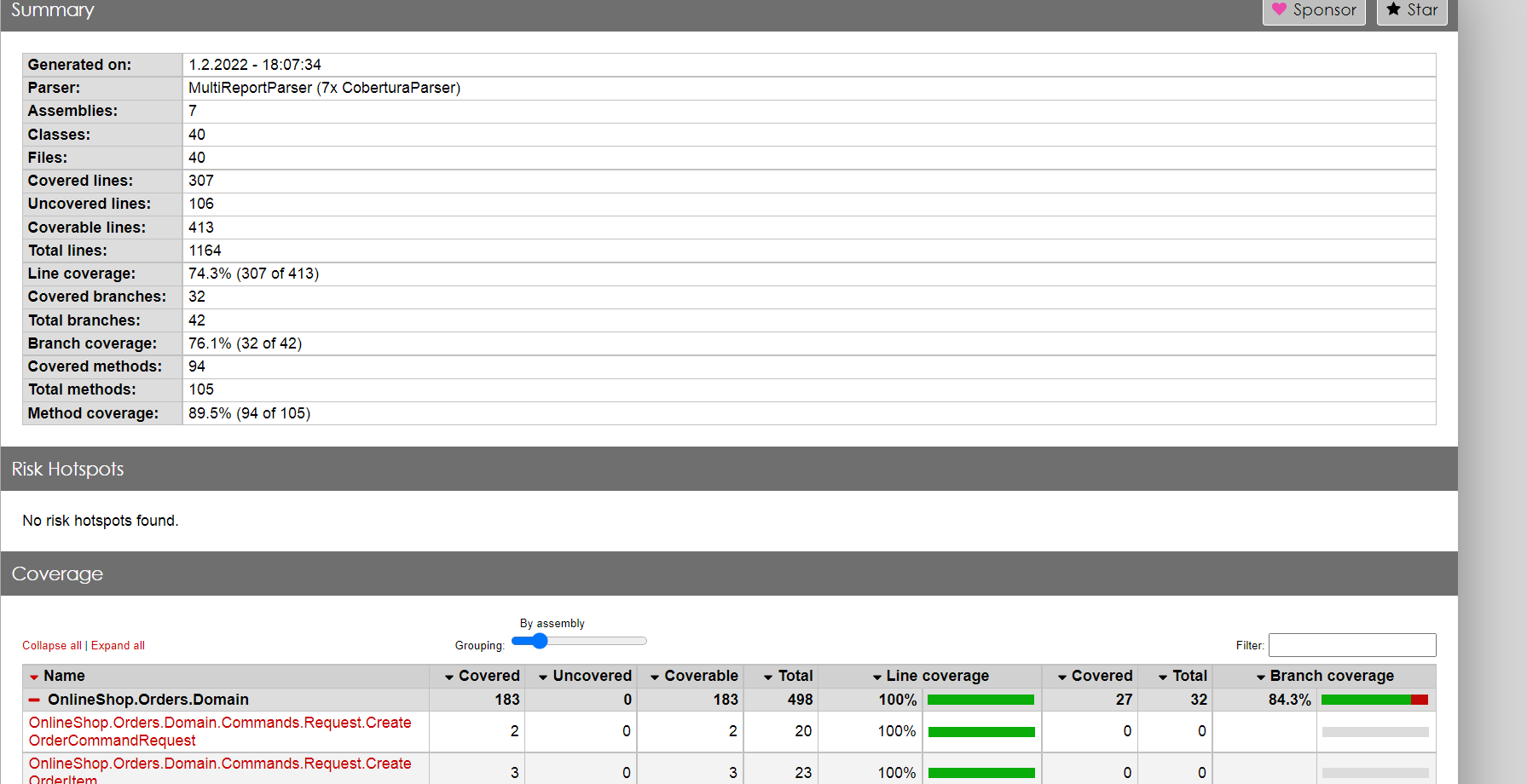
There are a few group of assumptions when developing the project:

## Authentication and Authorization

Authentication and Authorization is not included in the project.

Unit Test

For coverage report %70 is accepted. Orders solution can be analyzed to see the coverage report.



Şekil 6 Coverage report for orders microservice

## Mailing

Mailing is not included in the project.

## Update Assumption

Update operation at rest api used as PUT rest operation, it may be used also as PATCH operation to update some parts of the entities.

## Api List Paging

Paging is not defined for getting all objects.

## Logging

Logging is not used in the project.

## Sqllite

Sqllite db is created on HttpApi project folder as {similarModulename}.db. This db may be at more secure place and may be configured by an external configuration/json file.

## Tests

* 1. HttpApi project has a test project to test CRUD operations. Mocking is used on the project. On every CRUD operation, a success and an error test case is implemented.
  2. Domain has a test project to test CRUD operations on services layer. Mocking is used on the project. On every CRUD operation, a success and an error test case is implemented.
  3. For orders solution repository has a test project.
  4. Also other layers/projects test projects may be added.

## Stress Tests

Jmeter configuration is builded to do a stress test. The parameters are defined easy for the development purposes but they can be changed easily for the production enviroment. The Jmeter test configuration file is in the test\Stress test folder of the Orders solution as TestPlan.jmx.

# External Packages

## AutoMapper

AutoMapper.Extensions.Microsoft.DependencyInjection nuget package is included to map service DTO objects to persistence objects.

## MassTransit

MassTransit, MassTransit.AspNetCore and MassTransit.RabbitMQ is used in project to communicate with other microservices.

## MediatR

MediatR library is used for CQRS simplification.

## Swagger

Swashbuckle.AspNetCore is included to extend the project for open api calls.

## EntityFrameworkCore for SqlLite

Microsoft.EntityFrameworkCore.Sqlite nuget package is included to use Sqllite operations.

## MongoDB

MongoDB is used for orders microservice only. By using this nosql db we boost our query performance as the fulfilled order json is ready for querying (actually we only retreive the data when the client queries it, because we already prepare the data before). If we use a relational database we must do join operations to get the fulfilled order object, so it will be a performance bottleneck for the system.

## Moq

For mocking object moq nuget package is included.

## Xunit

Xunit nuget package is included to do unit tests. For coverage reports coverlet.collector, coverlet.msbuild, microsoft.codecoverage, xunitXml.TestLogger libraries are also included.