Lab: Automated Testing

Lab for the "Software Engineering and DevOps" module @ SoftUni

We are given a simple JS app that lets the user add tasks, mark them as completed, delete them and filter them according to their status.

We should use Playwright and write some code to test the UI of this app.

1. Bug Tracking Tools

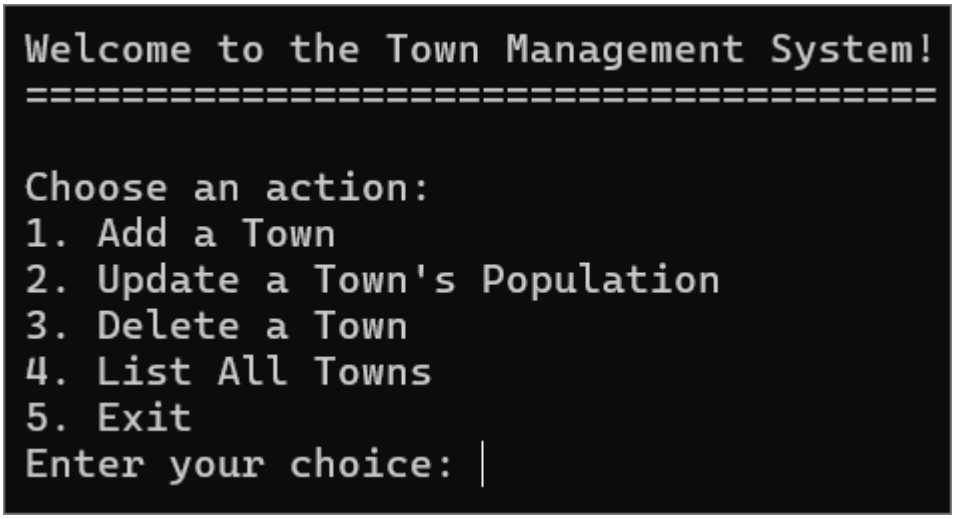
You can checkout the links below and take a look at how bugs are tracked in real projects:

- https://github.com/twbs/bootstrap/issues
- https://github.com/twbs/bootstrap/issues/31392
- https://daosio.atlassian.net/jira/software/c/projects/CART/issues

2. Integration Testing

Open **Visual Studio** and load the **TownsApplication** from the resources to inspect it. It is a simple application which let's you input data about towns and validate them in the process.

The architecture of the application can be defined as a basic MVC-like structure, although it is a console application:

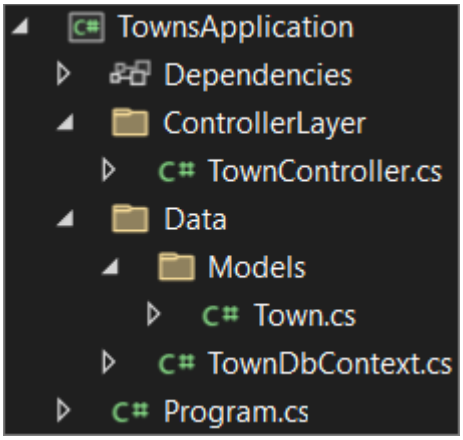


The application has a controller layer – **`TownController`** which is responsible for the business logic of the application and handling the CRUD operations. It has four methods:

- **`AddTown`** – add a town
- **`UpdateTown`** – update a town's population
- **`DeleteTown`** – delete a town by **`id`**
- **`ListTowns`** – list all of the input towns.

The data access layer is represented by the **`TownDbContext`**. It also manages the interaction with the in-memory database. The table of towns in the database is represented by the **`DbSet<Town>`** property. The **`OnConfiguring`** method ensures that the context uses in-memory database.

The domain model for storing towns is represented by the **`Town`** data model.



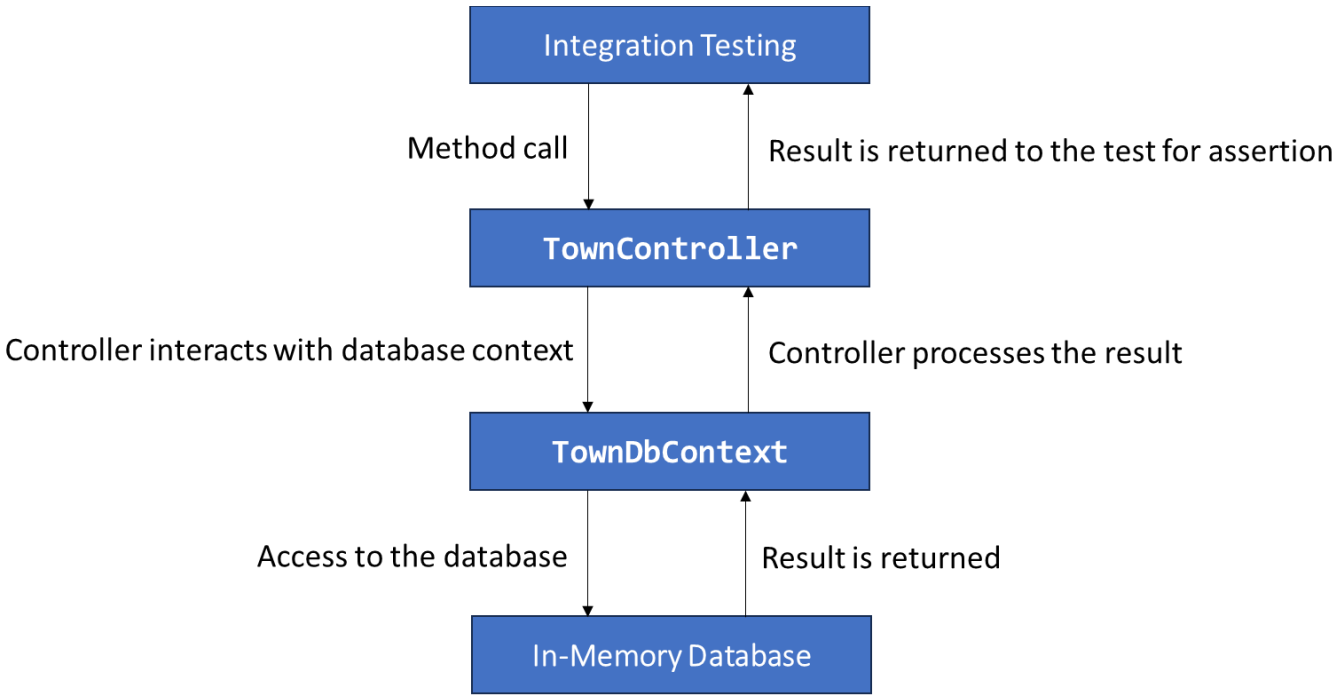
How does the application work? The **TownController** calls methods that interact with the **TownDbContext**.

	For the purpose of this demo, the **TownDbContext** uses an in-memory database. Remember that you won't be working with apps that have in-memory database in the real world. It is considered bad practice using in-memory databases in production environments and for performance testing. The common uses for in-memory database are for unit and integration testing (but for the tests themselves, not for the application that is being tested).

And how does the integration testing work on this app? As you can see, each test uses a fresh in-memory database to ensure that tests are independent. This is done by the **ResetDatabase** method in the controller.

After that, we have tests that are executed on these newly initialized databases.

You can see the whole process for each test described in the diagram below:



Finally, let's see how to execute the tests. Usually, to run xUnit tests, you need the packages described below:

```
xunit
xunit.runner.visualstudio
```

You can either download them using the built-in NuGet Package Manager from Visual Studio or use the CLI to download and install them.

For you, as a developer, it is a must-know how to use CLIs, so open the Package Manager Console in Visual Studio, chose the **`TownApplication.IntegrationTests`** project and use the following commands to download and install the required packages:

```
Install-Package xunit
Install-Package xunit.runner.visualstudio
```

We will also be using the CLI to run the tests using the command below:

```
dotnet test
```

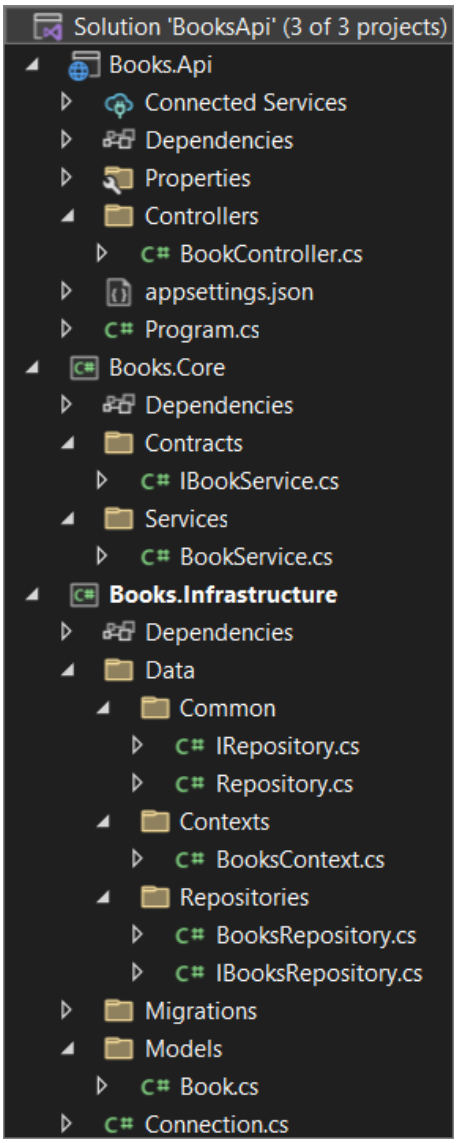
Now you know how to run **integration** tests written with **xUnit** on a **C#** application. Don't worry, in the next session, we will see more examples.

3. Swagger

In the resources, you have a **BooksAPI** application, written with **C#**.

This is a simple REST API for displaying, creating, editing and deleting products. The **Swagger** tool **is used** to document the **API** and try it out directly from the browser.

The **BooksApi** application follows a layered architecture that separates concerns into distinct projects, which is a common and best practice in large-scale applications.



The solution is composed of three distinct projects:

- **`Books.Api`**
- **`Books.Core`**
- **`Books.Infrastructure`**

Books.Api (Presentation Layer)

This project represents the API layer of the application.

The **`BookController`** in it exposes **HTTP API endpoints** that clients (such as web or mobile apps) interact with. The controller interacts with the **underlying services** to handle **requests** related to books.

In this layer there is also the entry point of the API – **`Program.cs`**

This layer depends on `Books.Core` to handle business logic and **`Books.Infrastructure`** for data access.

Books.Core (Business Logic Layer)

This layer is responsible for handling the business rules, data validation, and operations that do not directly involve database access. Here we have the contracts and the services.

Contracts are represented by interfaces that define the business logic operations, which are related to the books. In other word, these are the abstract definitions of the methods for CRUD operations.

Services implement the business logic defined in the interfaces. In this case, the services interact with the repositories to handle operations that manage books in the database. It enforces business rules and validations before data is passed to the repository layer for persistence.

Books.Infrastructure (Data Access Layer)

This project contains the data access logic, interacting directly with the database and models.

Swagger Integration

We have Swagger integrated into the **`Books.Api`** project, allowing developers and users to explore and interact with the API via a user-friendly web interface.

Swagger automatically generates API documentation based on the controller and model definitions, and it provides an interactive interface where users can execute API operations (GET, POST, PUT, DELETE) directly from the browser.

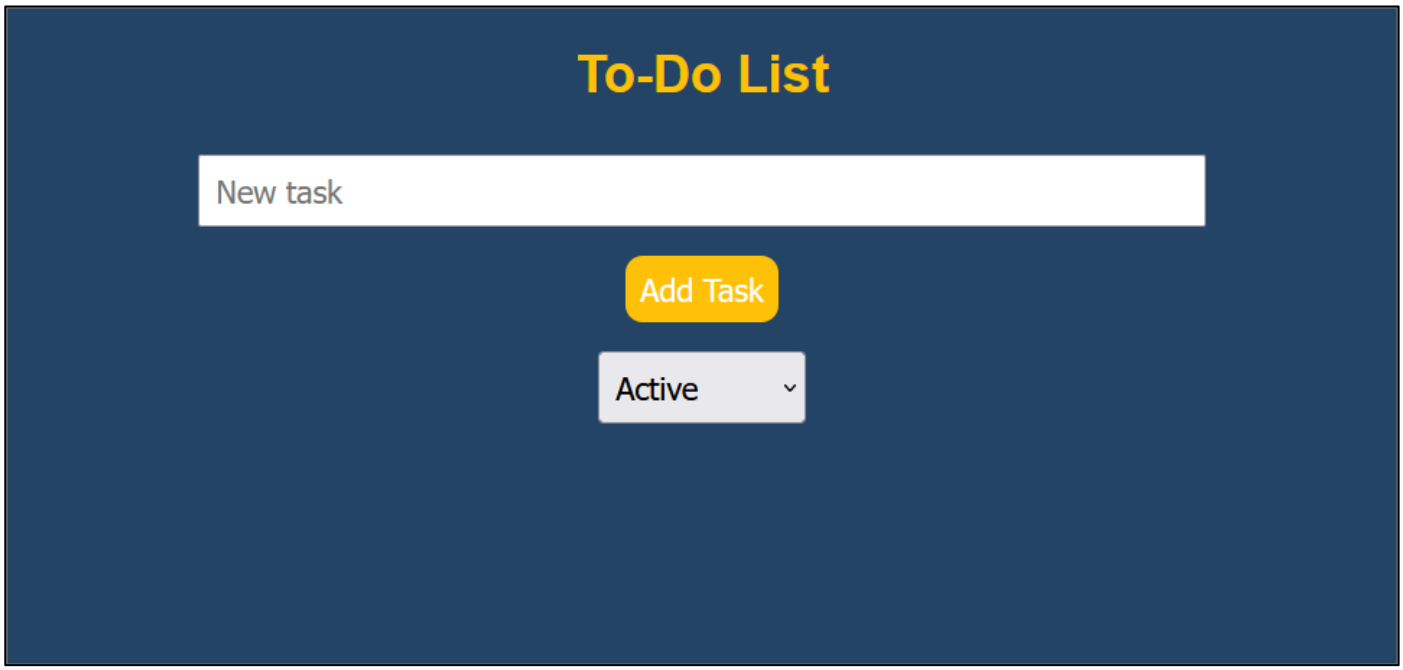
You can run the app and explore the API interface in the browser.

Now that you have acknowledged yourself with the architecture of this API application, you will be able to perform some testing using it in the next session.

4. Web UI Testing

In the resources, there is a **To-Do** app. This is a simple **JS** app that lets the user add tasks, mark them as completed, delete them and filter them according to their status. It has some Playwright tests written. In order for you to be able to run them, you need to open Visual Studio Code and install Playwright using this command:

```
npm install
npm install @playwright/test
npx playwright install
```



npm install @playwright/test

This command installs Playwright's test package (**`@playwright/test`**) into your project using **`npm`**. The package provides utilities for writing and running tests, specifically built for Playwright. It adds Playwright as a dependency to our project, enabling us to write tests for browser automation.

npx playwright install

This command installs the necessary browser binaries (such as Chromium, Firefox, and WebKit) for Playwright. As you know, Playwright automates browser actions, and this command ensures that the required **browsers** are **downloaded** and available **locally** for testing.

Now, let's start the server using

```
npm start
```

Finally, we can run the tests, using this command:

```
npx playwright test
```

We should see that all of the tests pass:

```
PS D:\SoftUni\to-do-app> npx playwright test

Running 4 tests using 1 worker

  ✓  1 tests\todo.test.js:3:1 › user can add a task (2.0s)
  ✓  2 tests\todo.test.js:11:1 › user can delete a task (510ms)
  ✓  3 tests\todo.test.js:21:1 › user can mark a task as complete (493ms)
  ✓  4 tests\todo.test.js:31:1 › user can filter tasks (518ms)

  4 passed (5.5s)
```