TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Aleksandr Kožemjakin 185955IADB

# Upgrading Electronic Devices Factory Testing Web Environment in a Private Company

Bachelor's thesis

Supervisor: Nadežda Furs
MBA

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Aleksandr Koženjakin 185955IADB

# Seadmete tehasetestide veebikeskkonna uuendamine elektroonikaettevõttes

Bakalaureusetöö

Juhendaja: Nadežda Furs
MBA

Tallinn 2021

## Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Aleksandr Kožemjakin

17.05.2021

# Abstract

The goal of the thesis is to develop a web environment for factory tests. The solution is made for company inner use to improve quality of analysing electronic devices test results.

The solution has got backend and frontend implemented. Backend has got MongoDB database support and services with API controllers. Frontend was created using Blazor WebAssembly and its features.

Created project is a prototype that is being further developed and will be fully integrated in the company use in the future.

This thesis is written in English and is 49 pages long, including 5 chapters, 31 figures and 9 tables.

# Annotatsioon

Seadmete tehasetestide veebikeskkonna uuendamine elektroonikaettevõttes

Lõputöö eesmärk on areneda veebikeskkond tehasetestide jaoks. Veebikeskkond peab salvestama ja näitama infot seadmete testimise kohta, näiteks seadmete loetelu, nende issue-d ja testitulemused. Lahendus on mõeldud ettevõtte sisekasutuseks, et parandada elektroonikaseadmete testitulemuste analüüsi kvaliteeti.

Lõputöös kirjeldatakse ettevõttes praegu kasutatavat veebikeskkonda koos selle probleemidega, analüüsitakse mitu veebikeskkonda loomiseks sobivat tehnoloogiat, eesmärgiga valida kõige sobivam ja mugavam, luuakse uue veebikeskkonna prototüüpi koos dokumentatsiooniga.

Lahenduses on rakendatud *backend* ja *frontend*. *Backend*-is on implementeeritud MongoDB andmebaasi tugi ja API kontrollerid koos teenustega. *Frontend*-i loodi *Blazor WebAssembly* abil.

Loodud lahendus on prototüüp, mida arendatakse edasi ja mis integreeritakse tulevikus täielikult ettevõtte kasutusse. Veebikeskkond võimaldab näha tooteid, seadmeid ja seadmete issued. Pealegi on *frontend*-i abil võimalik luua uusi seadmeid ja nende issued. *Frontend*-il on mugav routing, seega on võimalik linke parameetritega jagada ja vajalikku infot kohe renderdada.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 49 leheküljel, 5 peatükki, 31 joonist, 9 tabelit.

# List of abbreviations and terms

| | |
|---|---|
| AOP | Aspect Oriented Programming |
| API | Application Programming Interface |
| BSON | Binary JavaScript Object Notation |
| CLR | Common Language Runtime |
| CRUD | Create, Read, Update, Delete |
| DI | Dependency Injection |
| DOM | Document Object Model |
| ERD | Entity Relationship Diagram |
| HTML | HyperText Markup Language |
| HTTP | HyperText Transfer Protocol |
| IDE | Integrated Development Environment |
| IoT | Internet of Things |
| JRE | Java SE Runtime Environment |
| JSON | JavaScript Object Notation |
| PC | Personal Computer |
| REST | Representational State Transfer |
| UI | User Interface |
| URL | Uniform Resource Locator |
| wasm | WebAssembly |
| WPF | Windows Presentation Foundation |

# Table of contents

# List of figures

# List of tables

# 1 Introduction

In the following thesis development of the web environment for factory tests is described and provided analysis and overview of technologies that were used.

## 1.1 Background

Nowadays electronic devices are used everywhere. Since modern world is so dependent on electronics it is essential to provide its correct functioning.

Correct functioning of an electronic product can be achieved with testing. Complete production cycle of an electronic product requires separate tests for each electronic part of a product during each stage of production. It means that circuit board for a product must be tested separately and eventually it must be tested with other components when a device is assembled.

For efficiency and convenience of testing it is necessary to store and display testing information in a clear format for a developer. This way it is easier for the developer to gather and analyse testing information.

This approach requires development of a test environment with modern and comfortable UI (*User Interface*) which meets manufacturer requirements for proper testing.

## 1.2 Problem

Company area of work is electronic equipment production. Test environment that is currently used in equipment manufacture is outdated and needs to be updated.

- The UI is not clear and is inconvenient to use.
- Production standards require more detailed information recording, which is currently not possible.

## 1.3 Goal

The goal of the thesis is to create a web environment for the electronic devices' factory testing. Web environment must store and display information about devices factory testing, such as testing devices list, their issues and test results. Created web environment will be a prototype that will be further developed and integrated in the company use in the future.

Additionally, the environment must meet following requirements:

- Must support existing interfaces between current systems and services.
- The web environment must be accessible to all most common browsers (Chrome, Firefox, Edge Chromium, Safari).

## 1.4 Methodology

Currently used web environment with it`s problems is described.

In order to solve the problem, several technologies suitable for web environment development are analysed, aiming to choose the most suitable and convenient one.

Based on requirements and analysis, technology choice is explained.

New web environment prototype with documentation is created, which implements the most significant functionality and creates basis for further development.

Functionality that will be added in the future is described.

## 1.5 Author`s role

The author of the thesis described technologies used for a project development and provided reasoning for choosing them. The author of the thesis participated in a documentation creation, project structuring and technical implementation.

The project was developed in a team with one developer of the company. The author took part in a development of all parts described in the thesis.

# 2 Analysis

In this part different technologies suitable for solution are described and provided a more detailed view on current solution with it`s problems.

## 2.1 Current solution overview

Backend and frontend of current web environment for factory tests is done in PHP. There are no any documentations provided which makes it hard to implement new features and understand existing code.

Existing code is badly structured and therefore hard to refactor and complement.

Database for the environment is made in MySQL. No ERD (*Entity Relationship Diagram*) model schemas were originally made. Existing ERD model schema was generated automatically based on existing entities and may be inaccurate. Current database structure is too complicated and has got unnecessary tables, which makes it hard to follow data flow and make structure changes if necessary.

UI of the environment is outdated and uncomfortable to use.

Current environment communicates with other application via API (*Application Programming Interface*) in order to get devices and their test results.

Based on the problems it was decided that developing of a new environment is easier and more profitable solution than updating the existing environment.

## 2.2 New solution requirements

Since current solution uses separate program for getting test results, it is essential to provide communication between a new solution and the program that is currently used for running test cases.

Since developers use different browsers, solution must have cross-browser compatibility and work correctly with most popular browsers (i.e. Chrome, Firefox, Edge Chromium, Safari).

Code must be well structured and easy to follow, so it is possible to update web environment in the future.

## 2.3 Database

In this part database technologies suitable for solution are described and compared. In the end it is explained which one was chosen for development and why.

### 2.3.1 MySQL

Current solution uses MySQL database.

MySQL is the most popular Open Source SQL database management system.

MySQL databases are relational. A relational database stores data in separate tables rather than putting all the data in one big storeroom [1].

SQL is a database computer language designed for managing data in relational database management systems [2].

SQL has remained a popular choice for database users over the years mainly due to its ease of use and the highly effective manner in which it queries, manipulates, aggregates data and performs a wide range of other functions to turn massive collections of structured data into usable information.

For this reason, it has been incorporated into numerous commercial database products, such as MySQL, Oracle, Sybase, SQL Server, Postgres and others [3].

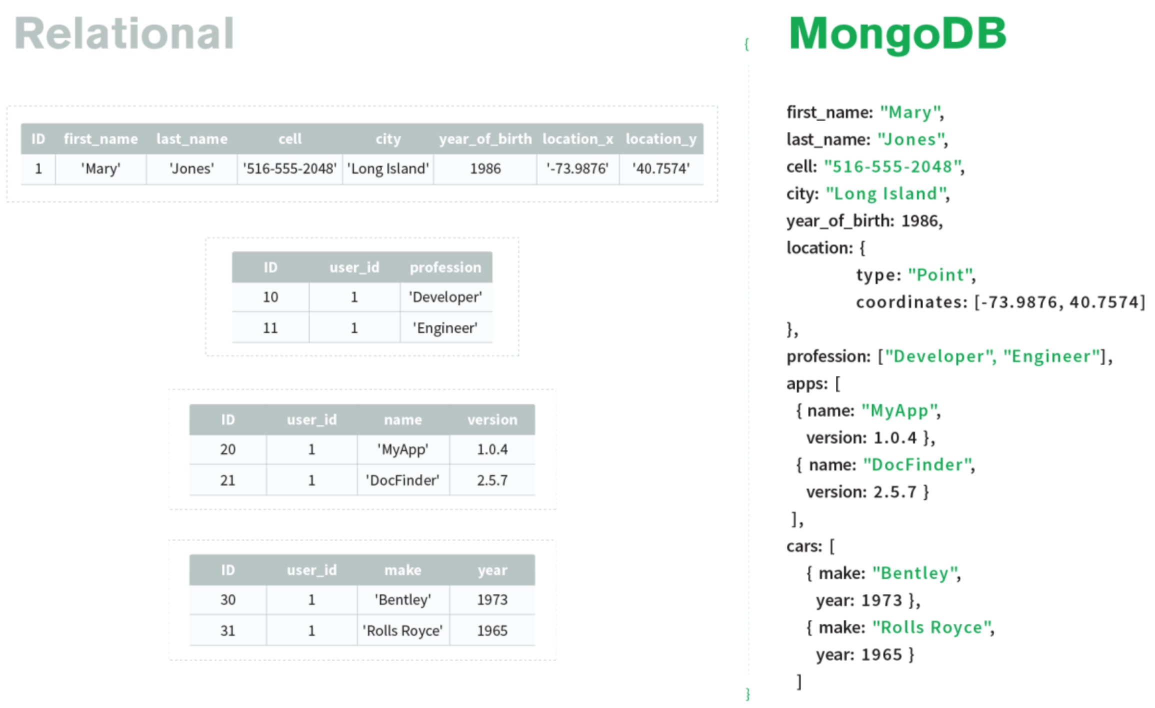
Different SQL iterations usually have same basic commands as select, insert, update and create. This makes it possible for a developer with some knowledge of SQL to work in different SQL products and environments.

### 2.3.2 MongoDB

MongoDB is a NoSQL document-oriented database.

NoSQL is best considered with the acronym “NOSQL” - Not Only SQL - which more accurately represents an approach that combines non-relational databases with the use of relational ones. This approach seeks to leverage both NoSQL and SQL technologies in order to balance the demands of performance, scalability, and schema flexibility with data integrity and consistency.

The primary reason for moving away from the relational model is to make scaling out easier, but there are some other advantages as well. MongoDB provides a more flexible approach for data storing such as embedded documents and arrays. The document-oriented approach makes it possible to represent complex hierarchical relationships with a single record (Figure 1) [4], [5].



| ID | first_name | last_name | cell | city | year_of_birth | location_x | location_y |
|---|---|---|---|---|---|---|---|
| 1 | 'Mary' | 'Jones' | '516-555-2048' | 'Long Island' | 1986 | '-73.9876' | '40.7574' |

| ID | user_id | profession |
|---|---|---|
| 10 | 1 | 'Developer' |
| 11 | 1 | 'Engineer' |

| ID | user_id | name | version |
|---|---|---|---|
| 20 | 1 | 'MyApp' | 1.0.4 |
| 21 | 1 | 'DocFinder' | 2.5.7 |

| ID | user_id | make | year |
|---|---|---|---|
| 30 | 1 | 'Bentley' | 1973 |
| 31 | 1 | 'Rolls Royce' | 1965 |

```
{
first_name: "Mary",
last_name: "Jones",
cell: "516-555-2048",
city: "Long Island",
year_of_birth: 1986,
location: {
          type: "Point",
          coordinates: [-73.9876, 40.7574]
},
profession: ["Developer", "Engineer"],
apps: [
 { name: "MyApp",
   version: 1.0.4 },
 { name: "DocFinder",
   version: 2.5.7 }
 ],
cars: [
   { make: "Bentley",
     year: 1973 },
   { make: "Rolls Royce",
     year: 1965 }
  ]
}
```

Figure 1. Relational DB and document DB example (Picture from the Internet) [6].

A document’s schema is dynamic and self-describing, so it is not needed to first pre-define it in the database. Fields can vary from document to document, and it is possible to modify the structure at any time, allowing to continuously integrate new application functionality, without additional schema migrations. If a new field needs to be added, it can be created without affecting all other documents in the collection, without updating a central system catalogue and without taking the database offline.

When it is needed to make changes to the data model, the document database continues to store the updated objects without the need to perform extra operations. Documents allow multiple versions of the same schema to exist in the same table space.

MongoDB stores data as JSON (*JavaScript Object Notation*) documents in a binary representation called BSON (*Binary JavaScript Object Notation*). Unlike most databases that store JSON data as primitive strings and numbers, the BSON encoding extends the JSON representation to include additional types such as int, long, date, floating point, and decimal128. This makes it much easier for applications using MongoDB to reliably process, sort, and compare data [7].

### 2.3.3 Database choice

It was decided to use MongoDB instead of currently used MySQL database or other relational SQL databases. Data stored in the application has an hierarchical structure without heavily interlinked parts. The layout of the data resembles structured documents and therefore it is more optimal to use MongoDB for the storage.

Since most of the relational SQL databases work by the same principle there is no need to compare each possible option with MongoDB separately.

One of the problems with current database is, that it contains a lot of tables which makes it’s structure too complicated. If database is a relational one, each entity regardless it’s importance requires separate table. Moreover, it is necessary to correctly setup hierarchy between tables. That approach grows into a quite complex database and requires JOIN operations between tables to get necessary information in one query. If there is a need to change database structure these changes adjustment can be time consuming because in relational database, it may be required to correctly merge data.

MongoDB solves these issues. Due to it’s embedded document approach it is possible to hold some minor entities inside bigger ones without making separate collections. This way it is easier to manage data between parent and child entities and it makes data model more compact and comfortable to use. Also, collection structure in MongoDB can be modified at any time without a need of additional database actions such as migrations. Because of that, it is possible to experiment with data structures during development without worrying about database errors. This can save time and make development more efficient.

## 2.4 Backend and frontend stack

In this part technologies that were considered for backend and frontend are described and compared. It is really important to choose the correct technology stack, so it is possible to develop system with good architecture. The architecture of a software system is the shape given to that system by those who build it. The form of that shape is in the division of that system into components, the arrangement of those components, and the ways in which those components communicate with each other. A software system that is hard to develop is not likely to have a long and healthy lifetime. So, the architecture of a system should make that system easy to develop [8].

### 2.4.1 .NET

One of the frameworks that was considered for building backend REST (*Representational State Transfer*) API is .NET using C# language.

.NET is an open-source and cross-platform development platform for building many types of applications. Designed by Microsoft, the platform supports multiple programming languages and libraries to build web, mobile, desktop, IoT (*Internet of Things*) applications, and more.

.NET is supported by Microsoft on Windows, macOS, and Linux. It is updated regularly for security and quality.

Besides C# language, it is possible to use many other languages in .NET. The languages that supported by Microsoft are C#, F#, Visual Basic.

Above the core components, .NET has different application model frameworks, that is, the libraries that offer support for developing different types of applications. The most notable ones are:

- ASP.NET: The framework that allows to build web applications and web APIs.
- WPF (*Windows Presentation Foundation*): A graphical UI for Windows desktop applications.

- Xamarin: The framework for building cross-platform mobile, TV, and desktop applications.
- Blazor: The framework to build client web applications by using C#. It also allows to generate client web apps in WebAssembly code [9], [10].

For package management there is special mechanism in .NET, called NuGet. It defines how packages for .NET are created, hosted, and consumed, and provides the tools for each of those roles.

Within an individual project, NuGet manages the overall dependency graph, which includes resolving multiple references to different versions of the same package. It is quite common that a project takes a dependency on one or more packages that themselves have the same dependencies. In the entire dependency graph, then, it is possible to have ten different references to different versions of the same package. To avoid bringing multiple versions of that package into the application itself, NuGet sorts out which single version can be used by all consumers [11].

### 2.4.2 Spring Framework

Other popular option for building REST API is Spring Framework.

Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications.

The Spring Framework consists of features organized into different modules (Figure 2). These modules are grouped into Core Container, Data Access/Integration, Web, AOP (*Aspect Oriented Programming*), Instrumentation, and Test. These modules provide functionality that defines Spring framework and helps in developing Spring framework applications. For example, Core Container provides Dependency Injection features, Data Access module contains tools for working with database such as Hibernate [12].

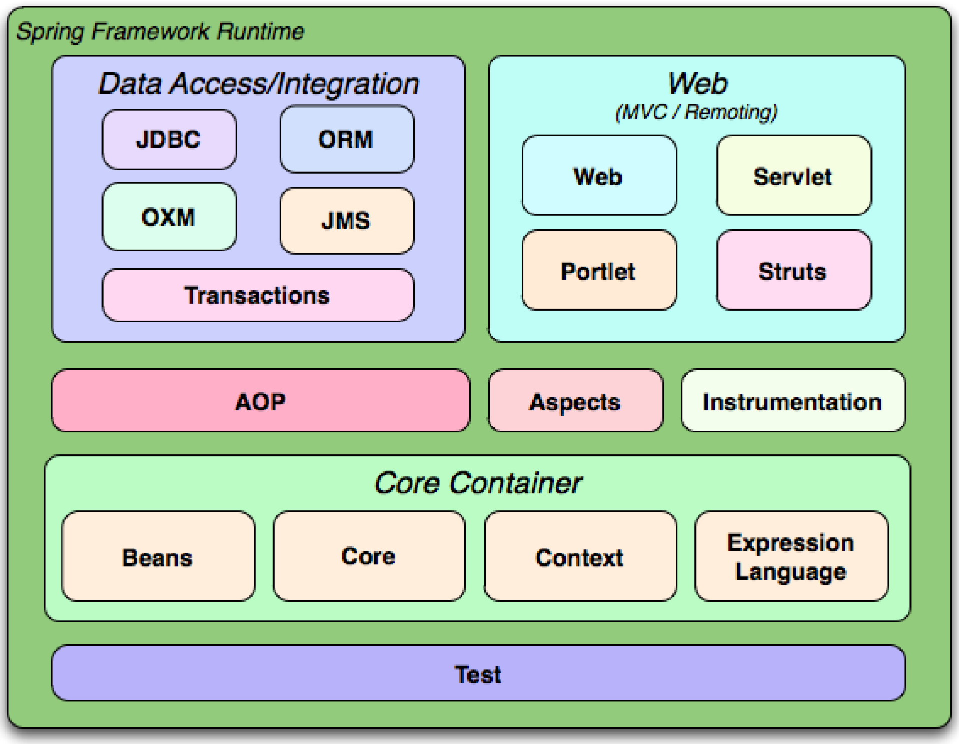


Figure 2. Spring Framework modules overview [12].

With the modules shown above it is possible to use Spring in all sorts of scenarios, from applets up to fully-fledged enterprise applications using Spring's transaction management functionality and web framework integration [13].

It is recommended to install Spring framework using a build tool that supports dependency management (such as Maven or Gradle) [14].

### 2.4.3 JavaScript

JavaScript is a dynamic computer programming language. It is lightweight and most commonly used as a part of web pages, whose implementations allow client-side script to interact with the user and make dynamic pages. It is an interpreted programming language with object-oriented capabilities.

Client-side JavaScript is the most common form of the language. The script should be included in or referenced by an HTML (*HyperText Markup Language*) document for the code to be interpreted by the browser.

It means that a web page need not be a static HTML, but can include programs that interact with the user, control the browser, and dynamically create HTML content [15].

Different browsers use different JavaScript Engines to interpret JavaScript code before rendering it on the web page. The major JavaScript engines are V8 used in Chrome, SpiderMonkey used in Firefox and Nitro used in Safari (Figure 3).

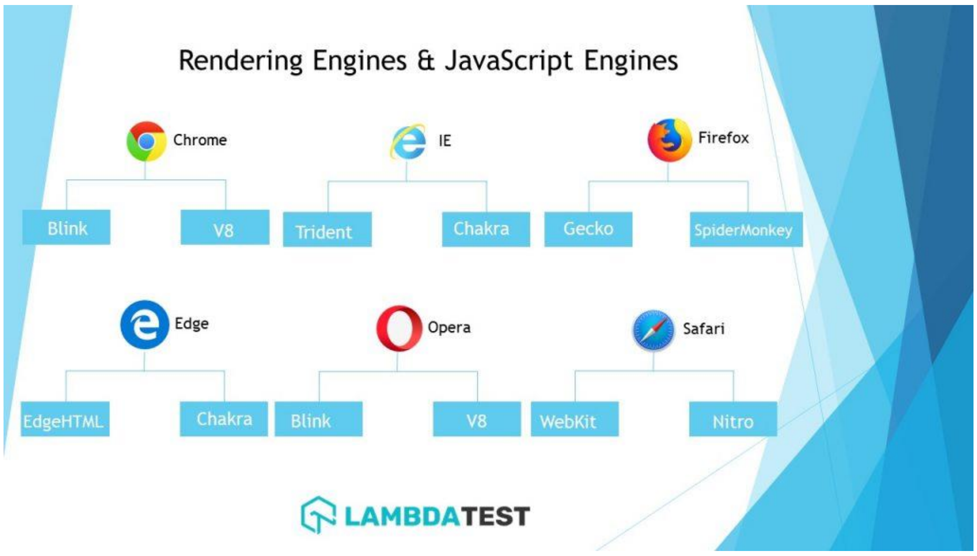


Figure 3. Rendering Engines and JavaScript Engines [16].

Different browsers contain different features. Some browsers show the popup and the perform tag management differently and some contains a few features to attract the audience. The feature that the user sees in the browser works because of the browser engine. Which is why it is possible that JavaScript can work differently in different browsers and it is important to provide correct client-side functioning in the most popular browsers [16].

Nowadays a lot of web applications made with JavaScript use some JavaScript framework. JavaScript frameworks are an essential part of modern front-end web development, providing developers with tried and tested tools for building scalable, interactive web applications [17]. JavaScript frameworks exist to provide a better developer experience. They do not bring brand-new powers to JavaScript; they give easier access to JavaScript's powers [18]. Most popular major JavaScript frameworks are React, Vue and Angular [19]. Each major JavaScript framework has a different approach to updating the DOM (*Document Object Model*), handling browser events, and providing an enjoyable developer experience [20].

### 2.4.4 Blazor WebAssembly

Though being part of the .NET ecosystem, which was briefly covered in section 2.4.1, Blazor WebAssembly is one of the major alternatives to dominant JavaScript frameworks and is therefore considered worth to be covered in detail.

Blazor is a client-side web UI framework similar in nature to JavaScript front-end frameworks like Angular or React but instead of JavaScript Blazor is based on C#. Blazor can run directly in the browser via WebAssembly. No browser plugins are required.

In 2015, the major browser vendors joined forces in a W3C Community Group to create a new open web standard called WebAssembly. WebAssembly is a new type of code that can be run in modern web browsers and provides new features and major gains in performance [21]. WebAssembly has been standardized and implemented by all major browsers.

Blazor has great tooling support in Visual Studio and Visual Studio Code. The framework also includes a full UI component model and has built-in facilities for:

- Forms and validation
- DI (*Dependency Injection*)
- Client-side routing
- Layouts
- In-browser debugging
- JavaScript interop

Blazor apps consist of one or more root components that are rendered on an HTML page (Figure 4).

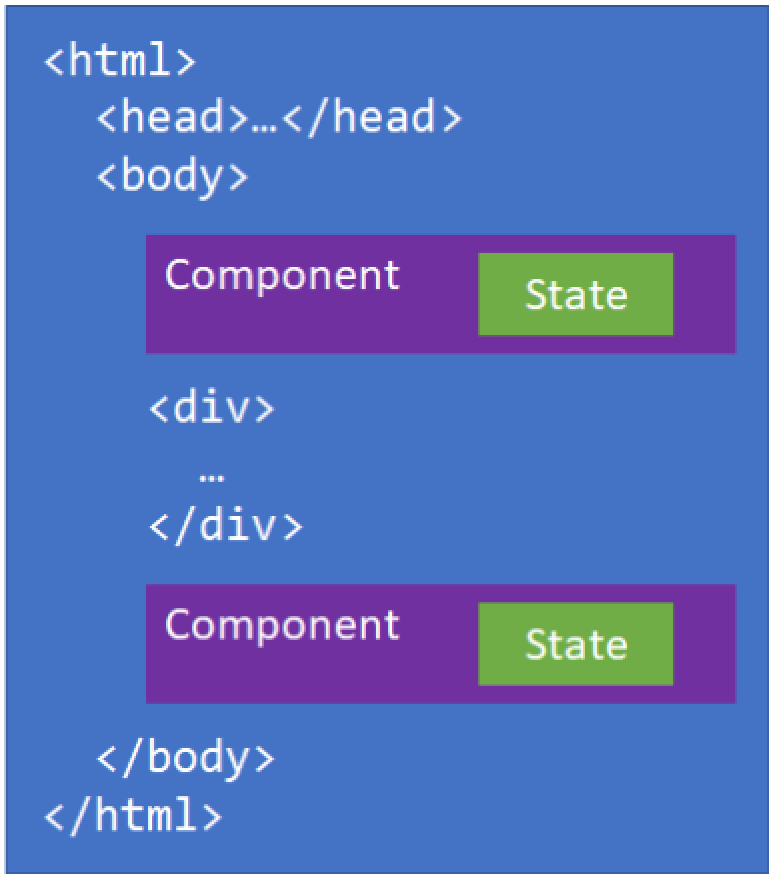


Figure 4. Blazor app in HTML [22].

Blazor components are .NET classes that represent a reusable piece of UI. Each component maintains its own state and specifies its own rendering logic, which can include rendering other components. Components specify event handlers for specific user interactions to update the component’s state [22]. The component class is usually written in the form of a Razor markup page. Razor is a syntax for combining HTML markup with C# code designed for developer productivity. Razor allows to switch between HTML markup and C# in the same file [23].

### 2.4.5 Summary of frontend and backend stack

It was decided to use .NET with Blazor WebAssembly.

One of the main reasons for that decision is that it is possible to write both, backend and client-side using C# language. When using Blazor, .NET is used throughout the whole development process. This brings the benefit of being able to reuse code from the back to the front end and vice versa. In addition, the ability for a developer to work in both areas without needing to know two different technologies.

If there appears a necessity to use JavaScript, Blazor solves this issue as well due to its JavaScript interoperability. With JS Interop, a Blazor app can invoke JavaScript functions from .NET methods and .NET methods from JavaScript functions [24].

Alternatively, it is possible to develop frontend and backend using JavaScript. Main disadvantage of that approach compared to .NET development is a necessity of installing and optimazing additional extensions and frameworks, while .NET provides all necessary tools out of the box. Moreover, company prefers to use strongly typed compiled languages instead of dynamically typed interpreted languages. It helps to reduce amount of mistakes and find errors early.

If JavaScript was used as a solution for client-side application, then backend should have been done in a separate language and framework. Moreover, JavaScript would have required an additional client-side framework. This would have been a big technology stack and constant changes between used technologies would be uncomfortable during development. .NET with Blazor solves this issue. Moreover, WebAssembly loads fast inside the browser, because pre-compiled wasm (*WebAssembly*) files have to be transported to the Internet, compared to JavaScript, which uses just in time compilation. Also, it is not needed to use polyfills in WebAssembly, because all browsers support it following same instructions [25].

## 2.5 Tools

Author used following developing tools:

- Visual Studio 2019 – Author used Windows 10 operational system during development, therefore it was decided to use Visual Studio 2019 as an IDE (*Integrated Development Environment*). Visual Studio includes compilers, code completion tools, graphical designers, and many more features to ease the software development process [26].

- Postman – Was used for testing backend system API. Postman is an API development tool. It has the ability to make various types of HTTP (*HyperText Transfer Protocol*) requests (GET, POST, PUT, PATCH) [27].

- Git – Git was used as a version control tool. Git is a fast, scalable, distributed revision control system with an unusually rich command set that provides both high-level operations and full access to internals [28].
- .NET 5.0 – Is a developer platform that was used for developing web environment.
- MongoDB – Database that was used for storing data.
- NetBeans – IDE that is used for developing Java applications. It was used to adjust another company software, which was developed, using this IDE.
- JDK - Includes a complete JRE (*Java SE Runtime Environment*) plus tools for developing, debugging, and monitoring Java applications [29]. Was used with NetBeans to run company software.

# 3 Implementation

In this part project structure is described and provided a detailed view on backend and frontend implementation of the project. Considering that this solution is a prototype in the future changes are inevitable.

## 3.1 Entities and API endpoints

Project had a documentation created beforehand. In the documentation were defined entities that project must use and API endpoints for communication with those entities. Based on these instructions a major part of the backend was developed.

Customer is the highest entity level in the system (Table 1). It groups together all products that can be manufactured for the customer. It also holds some generic information that associates the customer and product with other information systems inside the company.

Table 1. Customer object.

| Name | Type | Required | Description |
|---|---|---|---|
| Id | Guid | Yes | Unique identifier. |
| Name | string | Yes | Customer name. |
| Code | string | Yes | Customer identifier as defined in quality management system. |

Every manufactured product must be associated with a customer. Every product has a name that must be unique in the customer scope (Table 2).

Table 2. Product object.

| Name | Type | Required | Description |
|---|---|---|---|
| Id | Guid | Yes | Unique identifier. |
| Name | string | Yes | Product marketing name. |
| Code | string | Yes | Product identifier as defined in quality management system. |
| CustomerId | Guid | Yes | Reference to customer object the project is associated with. |
| Suites | Suite[] | No | List of test suites associated with this product. |

Each manufactured or maintained device must be part of one product (Table 3). Device inherits the name of the product. For unique identification, every device must have a number that is visible on a device.

Table 3. Device object.

| Name | Type | Required | Description |
|---|---|---|---|
| Id | Guid | Yes | Unique identifier. |
| Name | string | Yes | Device name (inherited from product). |
| Number | string | Yes | Human readable unique identification. |
| ProductId | Guid | Yes | Reference to product object the device is associated with. |

Issue is used to track all device lifecycle events and associate them with executed test suites (Table 4).

Table 4. Issue object.

| Name | Type | Required | Description |
|---|---|---|---|
| Id | Guid | Yes | Unique identifier. |
| Name | string | Yes | Issue name. |
| Description | string | No | Issue description. |
| DeviceId | Guid | Yes | Reference to device object the issue is associated with. |
| Closed | boolean | Yes | Indicates whether the issue is resolved or not. |
| Comment | Comment[] | Yes | List of comments associated with issue. |
| Sessions | Session[] | Yes | List of conducted test suites. |

Session is used to get set of test results (Table 5).

Table 5. Session object.

| Name | Type | Required | Description |
|---|---|---|---|
| Start | DateTime | Yes | When testing was started. |
| Stop | DateTime | Yes | When testing ended. |
| Results | Result[] | No | List of test results. Can be omitted if test session was aborted. |

| Name | Type | Required | Description |
|---|---|---|---|
| Verdict | enum | Yes | Final results for this test session:<br>PASSED – All checks were completed successfully<br>FAILED – Some checks failed<br>ABORTED – Test session was cancelled by user |
| Comment | string | No | Additional information associated with this test session. |

In a result object information about test results is stored (Table 6).

Table 6. Result object.

| Name | Type | Required | Description |
|---|---|---|---|
| Name | string | Yes | Test name (inherited from test declaration). |
| Verdict | enum | Yes | Final result of the test:<br>PASSED – Test was passed successfully.<br>FAILED – Test failed.<br>TIMEOUT – Test timed out<br>ABORTED – Test aborted |
| Name | Type | Required | Description |
| Comment | string | No | Additional information about the result. |
| Criteria | string | Yes | Describes the criteria required to pass the test. |
| Value | string | Yes | Actual test value. |

All tests in one suite must be run in the same fixed order as they are defined (Table 7). Failure of any one of the test cases will trigger the failure of the whole test suite.

Table 7. Test Suite object.

| Name | Type | Required | Description |
|---|---|---|---|
| Name | string | Yes | Suit name. |
| Description | string | No | Suit description. |
| Tests | Test[] | Yes | List of tests to be carried out in this suite. |

Every test case must have a clearly defined pass criteria that will be checked when the test case is run (Table 8).

Table 8. Test object.

| Name | Type | Required | Description |
|---|---|---|---|
| Name | string | Yes | Test name. |
| Description | string | No | Test description. |
| Criteria | string | Yes | Result required to pass for this test to be considered successful. |

With described entities, project has got API endpoints shown in Table 9.

Table 9. Project API.

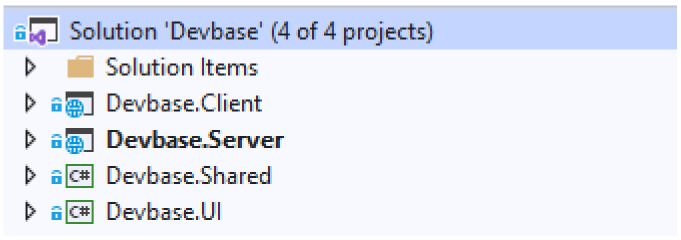
| Endpoint name | Request type | Description |
|---|---|---|
| /customers | GET | Lists all defined customers. |
| /customers | POST | Create new customer. |
| /customers/{id} | GET | Get customer identified by *id*. |
| /customers/{id}/products | GET | Get all products for a specific customer. |
| /products | GET | Lists all defined products. |
| /products | POST | Create a new product. |
| /products/{id} | GET | Get product identified by *id*. |
| /products/{id}/devices | GET | List all devices associated with the product identified by *id*. |
| /devices | POST | Create a new device. |
| /devices | GET | Lists all defined devices. |
| /devices/{id} | GET | Get device identified by *id*. |
| /devices/{id}/issues | GET | List all issues associated with the device identified by *id*. |
| /issues | POST | Create a new issue. |
| /issues/{id} | GET | Get issue identified by id. |
| /issues/{id}/sessions | GET | All test sessions associated with the issue identified by *id*. |

| **Endpoint name** | **Request type** | **Description** |
|---|---|---|
| /issues/{id}/sessions | POST | Create new test session under selected issue. |
| /issues/{id}/sessions/{i} | GET | Get information about the selected session under the issue *id* at position *i* in the issue list. |
| /issues/{id}/sessions/{i} | POST | Post test result for the test session. |
| /product/{id}/suites/ | POST | Create a new test suite. |
| /product/{id}/suites/{i} | GET | Get test suite for product *id* located at position *i*. |

Described API was implemented in the project backend.

## 3.2 Project structure

Project structure is shown in Figure 5.



```
Solution 'Devbase' (4 of 4 projects)
  Solution Items
  Devbase.Client
  Devbase.Server
  Devbase.Shared
  Devbase.UI
```

Figure 5. Project structure (screenshot by author).

Client part is a frontend of the application. It contains UI components of the application.

Server part is a backend of the application. It hosts client part of the application and contains controllers and services.

Shared part contains parts of the app, that client and server parts use, such as entities.
UI part contains widgets and UI layout that are used to build UI components from the client part and reduce code repetition.

## 3.3 MongoDB support implementation

To add MongoDB support in application, MongoDB was previously installed on authors PC (*Personal Computer*).

First, it was necessary to install MongoDB driver for .NET. It was done via NuGet package manager.

In appsettings.json file was added a configuration model (Figure 6).

```
    "FTDatabaseSettings": {
          "DatabaseName": "FTDatabase",
          "CustomersCollectionName": "Customers",
          "ProductsCollectionName": "Products",
          "DevicesCollectionName": "Devices",
          "IssuesCollectionName": "Issues",
          "ConnectionString": "mongodb://localhost:27017"
    }
```

Figure 6. Configuration model (part of author`s code).

Configuration model contains database name, connection string to the database and names of collections, that will be used for storing data.

FTDatabaseSettings class and it`s interface were created (Figure 7).

```
public interface IFTDatabaseSettings {
            string DatabaseName { get; set; }
            string CustomersCollectionName { get; set; }
            string ProductsCollectionName { get; set; }
            string DevicesCollectionName { get; set; }
            string IssuesCollectionName { get; set; }
            string ConnectionString { get; set; }
      }
public class FTDatabaseSettings : IFTDatabaseSettings {
            public string DatabaseName { get; set; }
            public string CustomersCollectionName { get; set; }
            public string ProductsCollectionName { get; set; }
            public string DevicesCollectionName { get; set; }
            public string IssuesCollectionName { get; set; }
            public string ConnectionString { get; set; }
      }
```

Figure 7. Database settings class (part of author`s code).

FTDatabaseSettings class is used to store appsettings.json file`s FTDatabaseSettings property values.

In a startup class the configuration instance to which the appsettings.json file's FTDatabaseSettings section binds is registered in the DI container (Figure 8).

```
        services.Configure<FTDatabaseSettings>(Configuration.GetSection(nameof
(FTDatabaseSettings)));
        services.AddSingleton<IFTDatabaseSettings>(sp=>sp.GetRequiredService<I
Options<FTDatabaseSettings>>().Value);
```

Figure 8. Database settings configuration in the startup class (part of author`s code).

This way MongoDB was implemented in the solution and now it is possible to create API.

## 3.4 API implementation

First of all, entity model classes with necessary attributes were created (Figure 9).

```
        public class Device {
                [BsonId]
                public Guid Id { get; set; }
                [Required]
                public string Name { get; set; }
                [Required]
                public string Number { get; set; }
                [Required]
                public Guid ProductId { get; set; }
        }
```

Figure 9. Device entity (part of author`s code).

Id property is required for mapping the CLR (*Common Language Runtime*) object to the MongoDB collection and it is annotated with [BsonID] to mark property Id as the document's primary key.

Rest of the models were created following similar pattern (Figure 10).

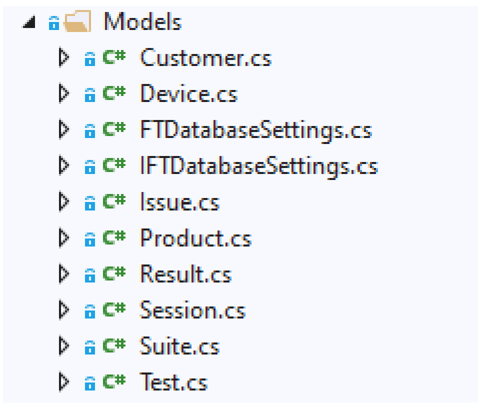
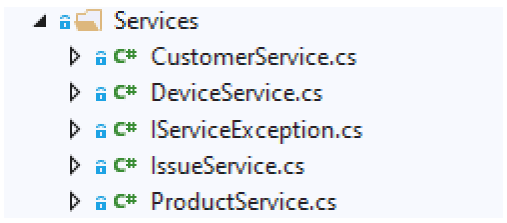


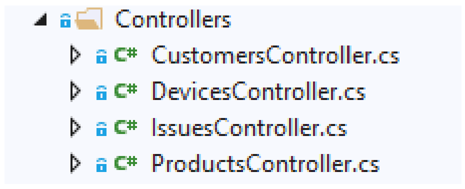
Figure 10. Entity models and database settings (screenshot by author).

For every model with it`s own collection was created separate service and controller (Figures 11, 12). Services implement functionality that is used by controllers to perform communication between API and database (Table 9).



Services
CustomerService.cs
DeviceService.cs
IServiceException.cs
IssueService.cs
ProductService.cs

Figure 11 Services (screenshot by author)



Controllers
CustomersController.cs
DevicesController.cs
IssuesController.cs
ProductsController.cs

Figure 12. Controllers (screenshot by author).

It was decided to create a few custom exceptions, so it is easier to understand what happens in the application if error occurs (Figure 13).

```
        public interface IServiceException {
                public abstract class NotFoundException : Exception {
                        public NotFoundException(string message) : base(message)
{ }
                }
                public abstract class DuplicateEntryException : Exception {
public DuplicateEntryException(string message) : base(message) { }
                }
        }
```

Figure 13. Custom exceptions interface (code created by project team).

NotFoundException is thrown when empty list is returned, or object is not found in database and DublicateEntryException is thrown if some conflict with duplicate entries based on business logic occurs. For example, two devices with same ProductId cannot have similar Number property.

To describe services implementation, device entity service will be shown and explained (Figure 14).

```
            private readonly IMongoCollection<Device> _devices;
            private readonly IMongoCollection<Issue> _issues;

            public DeviceService(IFTDatabaseSettings settings) {
                  var client = new MongoClient(settings.ConnectionString);
                  var database = client.GetDatabase(settings.DatabaseName);

                  _devices = 
database.GetCollection<Device>(settings.DevicesCollectionName);
                  _issues = 
database.GetCollection<Issue>(settings.IssuesCollectionName);

                  _devices.Indexes.CreateOne(new 
CreateIndexModel<Device>(new 
IndexKeysDefinitionBuilder<Device>().Ascending(device => device.ProductId)
                        .Ascending(device => device.Number), new 
CreateIndexOptions { Unique = true }));
            }
```

Figure 14. DeviceService constructor (part of author`s code).

In the preceding code, an IFTDatabaseSettings  instance is retrieved from DI via constructor injection. Also, unique MongoDB index is created. It is needed to avoid creation of device with same Numbers, if they have same ProductId. Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field. The unique property for an index causes MongoDB to reject duplicate values for the indexed field [30].

To use custom exceptions, it is needed to initialize them inside service class (Figure 15).

```
            public class NotFoundException : 
IServiceException.NotFoundException {
                  public NotFoundException(string message) : base(message) 
{ }
            }

            public class DuplicateEntryException : 
IServiceException.DuplicateEntryException {
                  public DuplicateEntryException(string message) : 
base(message) { }
            }
```

Figure 15. Exceptions initialization in service (code created by project team).

After these steps CRUD (*Create, Read, Update, Delete*) operations described in Table 9 were created (Figure 16).

```
		public async Task<Device> Get(Guid id) {
			var device = new Device();
			try {
				device = await _devices.Find(device => device.Id ==
id).FirstAsync();
			}
			catch (Exception) {
				throw new NotFoundException($"Device with ID {id}
not found.");
			}
			return device;
		}
```

Figure 16. Get function (part of author`s code).

Function in Figure 16 tries to get device from database based on given id and return it. If device is not found, then custom exception is thrown (Figure 17).

```
		public async Task<Device> CreateDevice(Device device) {
			try {
				await _devices.InsertOneAsync(device);
			}
			catch (MongoWriteException ex) {
				throw new DuplicateEntryException($"{ex.Message}");
			}
			return device;
		}
```

Figure 17. Create function (part of author`s code).

Function shown in Figure 17 tries to create a new device in database. If unique index conflict occurs, than MongoDB exception is caught and custom exception is thrown. Other services and CRUD operations in them are created following a similar pattern.

In the startup class all service classes are registered with DI to support constructor injection in consuming classes (Figure 18).

```
			services.AddSingleton<ProductService>();
			services.AddSingleton<DeviceService>();
			services.AddSingleton<IssueService>();
			services.AddSingleton<CustomerService>();
```

Figure 18. Services singleton implementation (part of author`s code).

Then entity controllers were created (Figure 19).

```
[Route("api/devices")]
[ApiController]
      public class DevicesController : ControllerBase {
            private readonly DeviceService _deviceService;
            public DevicesController(DeviceService deviceService) {
                  _deviceService = deviceService;
            }
```

Figure 19. Device API controller (part of author`s code).

API controller uses the DeviceService class to perform CRUD operations and contains action methods to support GET, POST, PUT, and DELETE HTTP requests (Figure 20).

```
            [HttpGet("{id}")]
            public async Task<ActionResult<Device>> Get(Guid id) {
                  try {
                        return Ok(await _deviceService.Get(id));
                  }
                  catch (DeviceService.NotFoundException ex) {
                        return StatusCode(StatusCodes.Status404NotFound,
$"{ex}");
                  }
            }
```

Figure 20. API Get function (part of author`s code).

Method shown in Figure 20 uses Get method from DeviceService to get device from database by id, if it catches custom exception, then it responses with code 404 and exception message.

Other controllers and their functions are made using similar pattern.

## 3.5 Frontend implementation

Frontend of the project is divided into two parts. First part is in the Client project. It contains main pages of the application user communicates with. Second part is located in the UI project. UI project contains widgets and components that are used by the first part of the frontend. Widgets are lightweight entities that are mainly used to enforce consistent styling and rendering. Components are stand-alone entities that encapsulate business logic and more complex rendering use cases.

Since frontend uses a lot of endpoints for web navigation and API requests it was decided to create a separate class with endpoints (Figure 21).

```
public class Endpoints {
            public class API {
                  public const string Devices = "/api/devices";
                  public const string Products = "/api/products";
                  public const string Issues = "/api/issues";
                  public const string Customers = "/api/customers";
            }

            public class Web {
                  public const string Devices = "/devices";
                  public const string Products = "/products";
            }

            public class Action {
                  public const string New = "new";
                  public const string Edit = "edit";
                  public const string Delete = "delete";
            }
      }
```

Figure 21. Endpoints class (part of author`s code).

This way all main endpoints are stored in one place and in case change is needed, it can be made in one place rather than changing through the whole frontend implementation.

### 3.5.1 Widgets

ContentTable widget is used to render lists of elements (Figure 22).

```
@typeparam TModel
@if(Model == null) {
      <p>Loading...</p>
      return;
}
<table>
      <thead>
            <tr> @TableHeader </tr>
      </thead>
      <tbody>
            @foreach(var entry in Model) {
                  <tr>@RowTemplate(entry)</tr>
            }
      </tbody>
</table>
@code {
      [Parameter]
      public RenderFragment TableHeader { get; set; }
      [Parameter]
      public RenderFragment<TModel> RowTemplate { get; set; }
      [Parameter]
      public IReadOnlyList<TModel> Model { get; set; }
}
```

Figure 22. ContentTable widget (code created by project team).

TableHeader and RowTemplate are templated components that can be specified using child elements that match the names of the parameters. TModel is a generic type, that is used to render RowTemplate.

Header widget is used in components to display header information (Figure 23).

```
<header>
	@ChildContent
</header>

@code {
	[Parameter]
	public RenderFragment ChildContent { get; set; }
}
```

Figure 23. Header widget (code created by project team).

These widgets help to reduce code repetitions and differences in styles, when lists or headers are added to the page.

### 3.5.2 UI components

DockedEditor is a component majority of frontend pages are built with (Figure 24).

```
@typeparam TModel
@inject NavigationManager NavigationManager
@implements IDisposable
@if (Model == null || !(Visible ?? true)) { return; }
@if (Action != null && !NavigationManager.HasAction(Action)) { return; }
<div class="@Class">
	<nav>
		@EditorHeader
		<a @onclick="HandleClose"><i class="far fa-window-
close"></i></a>
	</nav>
	<main>
		<aside>
			@EditorSidebar
		</aside>

		<section>
			@ChildContent
		</section>
	</main>
</div>
@code {
	[Parameter]
	public RenderFragment ChildContent { get; set; }
	[Parameter]
	public EventCallback<TModel> OnClose { get; set; }

	[Parameter]
	public RenderFragment EditorHeader { get; set; }
	[Parameter]
	public RenderFragment EditorSidebar { get; set; }
	[Parameter]
	public TModel Model { get; set; }
	[Parameter]
	public string Action { get; set; }
	[Parameter]
	public string Class { get; set; }
	[Parameter]
	public bool? Visible { get; set; }
	protected override void OnInitialized() {
		base.OnInitialized();
		NavigationManager.LocationChanged += LocationChanged;
	}
	private void LocationChanged(object sender, LocationChangedEventArgs
e) {
		StateHasChanged();
	}
	void IDisposable.Dispose() {
		NavigationManager.LocationChanged -= LocationChanged;
	}
	private void HandleClose(MouseEventArgs args) {
		OnClose.InvokeAsync();
	}
}
```

Figure 24. DockedEditor component (code created by project team).

In HTML part it has got a few if clauses. They define conditions when component must not render. OnInitialized function is invoked when the component is initialized. LocationChanged is an event that is triggered whenever the URL (*Uniform Resource Locator*) in the browser is altered. It passes an instance of LocationChangedEventArgs which provides information about new URL and if navigation was initiated via code or via an HTML navigation [31]. Dispose is executed when component is removed from its parent tree.

Section component is used to add sections to DockedEditor component (Figure 25).

```
<details open="open">
	<summary>
		@SectionHeader
		<i class="fa fa-angle-double-up"></i>
	</summary>
	@ChildContent
</details>
@code {
	[Parameter]
	public RenderFragment SectionHeader { get; set; }
	[Parameter]
	public RenderFragment ChildContent { get; set; }
}
```

Figure 25. Section component (code created by project team).

Other components can be put inside the templated components as well, which makes it possible to build the frontend using universal components, developed in advance. This can reduce amount of HTML markup and make frontend code cleaner.

### 3.5.3 Main components

DeviceList component is used to display all created devices (Figure 26).

```
@attribute [Route(Endpoints.Web.Devices)]
@attribute [Route(Endpoints.Web.Devices + "/{deviceId:guid}")]
@attribute [Route(Endpoints.Web.Devices +
"/{deviceId:guid}/issues/{issueId:guid}/{*sectionId}")]
@layout DeviceLayout
@if (_devices == null) {
	<p>Loading...</p>
	return;
}
<NavLink href=@($"{Endpoints.Web.Devices}?action=new")>Add</NavLink>
<ContentTable Model="_devices" Context="D">
	<TableHeader>
		<th>ID</th>
		<th>Device Number</th>
		<th>Product</th>
	</TableHeader>
	<RowTemplate>
		<td><NavLink
href=@($"{Endpoints.Web.Devices}/{D.Id}")>@D.Id</NavLink></td>
		<td>@D.Number</td>
		<td>@D.Name</td>
	</RowTemplate>
</ContentTable>
<aside>
	<DeviceDetails DeviceId="DeviceId" IssueId="IssueId"
SectionId="@SectionId" OnClose="HandleSave" />
	<AddDevice OnClose="HandleAdd" DeviceId="DeviceId" />
</aside>
@code {
	[Parameter]
	public Guid? DeviceId { get; set; }
	[Parameter]
	public Guid? IssueId { get; set; }
	[Parameter]
	public string SectionId { get; set; }
	private List<Device> _devices;
	protected void HandleAdd(Device device) {
		if (device != null) _devices.Add(device);
		NavigationManager.NavigateTo(Endpoints.Web.Devices);
	}
	protected void HandleSave(Device device) {
		NavigationManager.NavigateTo(Endpoints.Web.Devices);
	}
	protected override async Task OnInitializedAsync() {
		try {
			_devices = await
HttpClient.GetFromJsonAsync<List<Device>>($"{Endpoints.API.Devices}");
```

```
		}
		catch (System.Net.Http.HttpRequestException) {
			_devices = new();
		}
	}
}
```

Figure 26. DeviceList component (code created by project team).

Attribute part determines which URL links can lead to this page. It uses endpoints defined in the Endpoints class. In addition, route also has got such parts as deviceId:guid or issueId:guid. These are parameters passed in a URL link and type they must match. These parameters are defined as DeviceId, IssueId and SectionId in a Code part and are passed to other components, so these components can be rendered. In the DeviceList component it is possible to see how ContentTable widget with templated components is used. Templated components have info passed into them and due to logic written in the ContentTable widget, given information is being rendered. In addition, it is possible to see how functions and parameters are passed to other components in a AddDevice component. OnClose is the name of the function, that AddDevice uses, HandleAdd is the function that is being passed from DeviceList component.

In the AddDevice component DockedEditor is used as a component everything is built around (Figure 27).

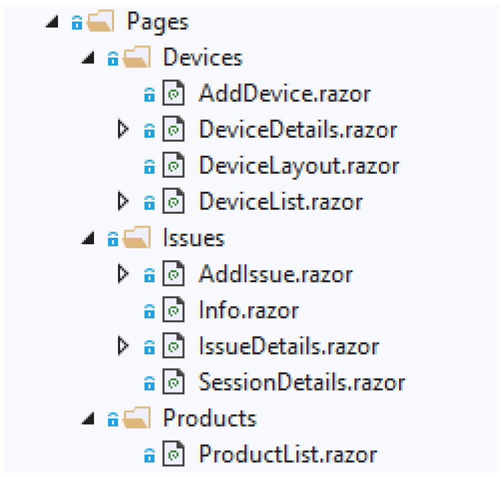
```
<DockedEditor Model="_device" Action="@Endpoints.Action.New"
OnClose="OnClose" Class="device">
	<EditorHeader>
		<h1>Add Device</h1>
	</EditorHeader>
	<ChildContent>
		<EditForm Model="_device" OnValidSubmit="CreateDevice">
			<label>
				Device Number
				<InputText @bind-Value="_device.Number"
DisplayName="Device number" />
			</label>
			<label>
				Product Family
				<InputSelect @bind-Value="_device.ProductId"
DisplayName="Product">
					<option>Select product</option>
					@foreach (var product in _products) {
						<option
value="@product.Id">@product.Name</option>
					}
				</InputSelect>
			</label>
			<button type="submit">Submit</button>
		</EditForm>
```

```
        </ChildContent>
</DockedEditor>
```

Figure 27. AddDevice HTML part (code created by project team).

AddDevice component uses EditForm component that is provided with Blazor. EditForm component allows to manage forms, validations, and form submission events. Editform has two attributes specified Model and OnValidSubmit. Model - Specifies the top-level model object for the form. OnValidSubmit is a callback/method that will be invoked when the form is submitted, and attributes used in form are determined to be valid [32].

Frontend part has got other components as well that are build using similar pattern and logic (Figure 28).



```
Pages
  Devices
    AddDevice.razor
    DeviceDetails.razor
    DeviceLayout.razor
    DeviceList.razor
  Issues
    AddIssue.razor
    Info.razor
    IssueDetails.razor
    SessionDetails.razor
  Products
    ProductList.razor
```

Figure 28. Client project components (screenshot by author).

Current list of components is not final, and it will increase as the project is being further developed.

## 3.6 Existing software adjustment

Current web environment communicates with another program built in Java for posting devices and test results. Author`s task was to find out what is needed to do to post trivial information via this program, so it is possible to adjust the program completely in the future.

Since program was built using NetBeans IDE, author installed it on his PC.

First of all, Device model was changed so it has got same structure as a model from .NET app. Then API link for posting devices was changed to „api/devices“. Since .NET app does not have authentication yet, it was necessary to avoid authentication need in a class that is used for posting. For that separate constructor was created (Figure 29).

```
        public DevbaseClient(String path, boolean skipLogin) {
            OkHttpClient.Builder builder = new OkHttpClient.Builder();
            okHttpClient = builder.build();

            retrofit = new Retrofit.Builder()
                .callFactory(okHttpClient)
                .baseUrl(path)
                .addConverterFactory(GsonConverterFactory.create())
                .build();
            devbase = retrofit.create(DevbaseService.class);
            authKey = "UNAUTHENTICATED";
            userAgent = "User-Agent: FTApp3/";
            httpAcceptHeader =  "Accept: application/vnd.ft3.v1+json";
            loggedIn = skipLogin;
        }
```

Figure 29. Constructor that skips authentication (part of author`s code).

This constructor imitates state when user has already logged in.

Finally, this class was used in a main method, device object was created and sent to .NET backend (Figure 30).

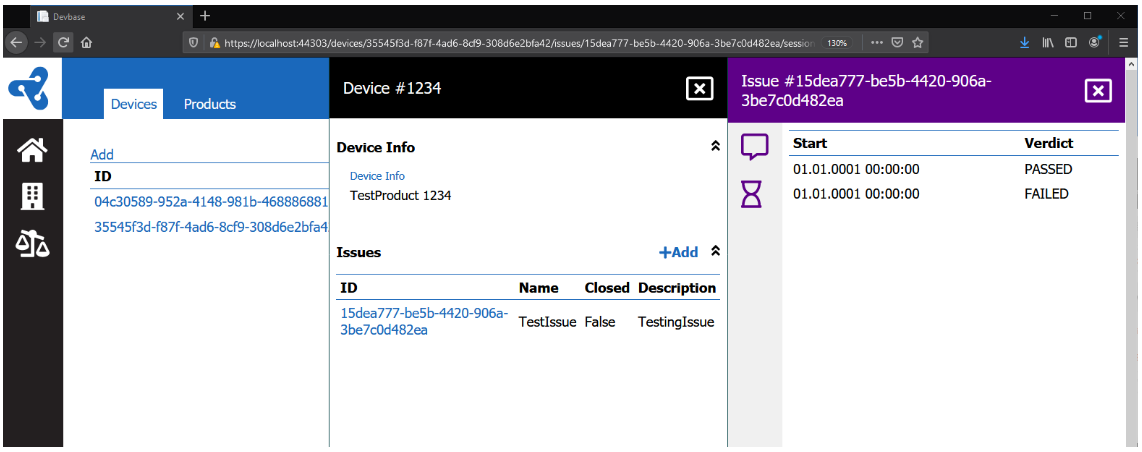
```
public static void main (String[] args) throws IOException {
        DevbaseClient devbaseClient = new DevbaseClient("http://localhost:307
9/", true);
        Random random = new Random();
        Device device = new Device();
        device.Id = UUID.randomUUID();
        device.Name = "TestingDeviceName";
        device.Number =Integer.toString(random.nextInt(100000));
        device.ProductId = UUID.fromString("284ed37f-c8b6-4761-9fc1-
e2760bb25373");
        devbaseClient.saveDevice(device);
}
```

Figure 30. Main function (part of author`s code).

Device object was successfully added to the database, using company software.

## 3.7 Result

During the development was created backend and frontend of the new web environment. Backend implements all logic described in Table 9 and in addition has got custom exceptions handling. Frontend development still continues, but it already allows, to see list of products, devices and device`s issues (Figure 31). Moreover, it is possible to create new devices and issues using frontend. Frontend has got comfortable routing, so it is possible to share full links with parameters and get necessary information rendered right away.


Devbase

https://localhost:44303/devices/35545f3d-f87f-4ad6-8cf9-308d6e2bfa42/issues/15dea777-be5b-4420-906a-3be7c0d482ea/session

Devices Products

Add

| ID |
|---|
| 04c30589-952a-4148-981b-468886881 |
| 35545f3d-f87f-4ad6-8cf9-308d6e2bfa4 |

Device #1234

Device Info

Device Info
TestProduct 1234

Issues +Add

| ID | Name | Closed | Description |
|---|---|---|---|
| 15dea777-be5b-4420-906a-3be7c0d482ea | TestIssue | False | TestingIssue |

Issue #15dea777-be5b-4420-906a-3be7c0d482ea

| Start | Verdict |
|---|---|
| 01.01.0001 00:00:00 | PASSED |
| 01.01.0001 00:00:00 | FAILED |

Figure 31. Web environment UI (screenshot by author).

New functionality implementation will be easier, because UI components, main pages are built around have been developed. UI is modern and comfortable to use. Due to the shared components and widgets, styling of new pages will be easier.

# 4 Further development

The current version of the solution is not final, and project is being further developed. Features that will be added in the near future, are authentication, commenting of the issues, detailed tests view and complete integration with another company software.

As the project is meant for developers to follow device state throughout testing, it will provide a detailed information about made tests. For example, it will be shown which tests are passed or failed, expected and received results and time it was consumed to pass all needed tests. It will be possible to comment on issues so developers can make notes and share them with each other.

When all necessary functionalities will be developed, old web environment will be replaced.

# 5 Summary

The goal of the thesis was to develop a web environment that stores and displays information about factory testing. Old web environment had problems that made it uncomfortable to use and hard to improve: lack of documentation, poorly structured code, complex ERD model with odd tables, outdated UI. In addition, web environment communicates with company software. This communication should have been saved in a new web environment.

The thesis has described technologies that were used for a development and provided reasoning for choosing them. Was developed a prototype version of the web environment. New web environment has got documentation, improved UI, well-structured code and a simplified database structure. Was created basic communication between company software and developed environment.

New environment still lacks functionality, such as authentication, test results display and full integration with company software. Web environment development continues, and it will replace old version in the future.

## References

[1] "What is MySQL?," Oracle Corporation, [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html. [Accessed 13 04 2021].

[2] H.-P. Halvorsen, "Structured Query Language," [Online]. Available: https://www.halvorsen.blog/documents/tutorials/resources/Structured%20Query%20Language.pdf. [Accessed 13 04 2021].

[3] R. Leach, "SQL Database Management," [Online]. Available: https://employthisguy.com/sql-database-management/. [Accessed 13 04 2021].

[4] K. Chodorow, MongoDB: The Definitive Guide, Second Edition, O'Reilly Media, Inc., 2013.

[5] M. Madison, "NoSQL Database Technologies," *Journal of International Technology and InformationManagement.*

[6] "What is a Document Database?," MongoDB, Inc., [Online]. Available: https://www.mongodb.com/document-databases. [Accessed 14 04 2021].

[7] M. W. Paper, "MongoDB Architecture Guide:Overview," [Online]. Available: http://s3.amazonaws.com/info-mongodb-com/MongoDB_Architecture_Guide.pdf. [Accessed 12 4 2021].

[8] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, Pearson Education, Inc., 2018.

[9] A. Chiarelli, "What is .NET? An Overview of the Platform," [Online]. Available: https://auth0.com/blog/what-is-dotnet-platform-overview/. [Accessed 18 04 2021].

[10] "Introduction to .NET," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/introduction. [Accessed 18 04 2021].

[11] "An introduction to NuGet," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/nuget/what-is-nuget. [Accessed 18 04 2021].

[12] "Introduction to Spring Framework," [Online]. Available: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html. [Accessed 18 04 2021].

[13] R. Johnson, "Spring java/j2ee Application Framework," [Online]. Available: https://docs.spring.io/spring-framework/docs/2.0.x/spring-reference.pdf. [Accessed 18 04 2021].

[14] "Installing Spring Boot," [Online]. Available: https://docs.spring.io/spring-boot/docs/1.3.0.M1/reference/html/getting-started-installing-spring-boot.html. [Accessed 18 04 2021].

[15] "JavaScript - Overview," [Online]. Available: https://www.tutorialspoint.com/javascript/javascript_overview.htm. [Accessed 19 04 2021].

[16] A. Zlatkov, “How JavaScript works: the rendering engine and tips to optimize its performance,” [Online]. Available: https://blog.sessionstack.com/how-javascript-works-the-rendering-engine-and-tips-to-optimize-its-performance-7b95553baeda. [Accessed 19 04 2021].

[17] “Understanding client-side JavaScript frameworks,” Mozilla, [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks. [Accessed 19 04 2021].

[18] “Introduction to client-side frameworks,” Mozilla, [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction. [Accessed 19 04 2021].

[19] “Front-end frameworks popularity,” [Online]. Available: https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190. [Accessed 19 04 2021].

[20] “Framework main features,” Mozilla, [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Main_features. [Accessed 19 04 2021].

[21] “WebAssembly Concepts,” Mozilla, [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts. [Accessed 19 04 2021].

[22] D. Roth, Blazor for ASP NET Web Forms Developers, Redmond, Washington: Microsoft Developer Division, .NET, and Visual Studio product teams, 2020 .

[23] “Introduction to ASP.NET Core Blazor,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-5.0. [Accessed 19 04 2021].

[24] C. Fuentes, “Why to choose blazor for your web development & why not,” [Online]. Available: https://www.zartis.com/why-to-choose-blazor-for-your-web-development-why-not/. [Accessed 19 04 2021].

[25] “Introduction,” WebAssembly Community Group, [Online]. Available: https://webassembly.github.io/spec/core/intro/introduction.html. [Accessed 29 04 2021].

[26] “Welcome to the Visual Studio IDE,” [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2019. [Accessed 20 04 2021].

[27] “Introduction to Postman for API Development,” [Online]. Available: https://www.geeksforgeeks.org/introduction-postman-api-development/. [Accessed 20 04 2021].

[28] “git - the stupid content tracker,” [Online]. Available: https://git-scm.com/docs/git. [Accessed 20 04 2021].

[29] “Java SE Downloads,” Oracle, [Online]. Available: https://www.oracle.com/java/technologies/javase-downloads.html. [Accessed 29 04 2021].

[30] “Indexes,” [Online]. Available: https://docs.mongodb.com/manual/indexes/. [Accessed 26 04 2021].

[31] “Detecting navigation events,” [Online]. Available: https://blazor-university.com/routing/detecting-navigation-events/. [Accessed 26 04 2021].

[32] “EditForm - forms and validation in Blazor,” [Online]. Available: https://dev.to/rineshpk/editform-forms-and-validation-in-blazor-54h7. [Accessed 26 04 2021].

## Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Aleksandr Koženjakin

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Upgrading Electronic Devices Factory Testing Web Environment in a Private Company”, supervised by Nadežda Furs
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

17.05.2021

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.