FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Open-Source Legal Process Designer in .NET Blazor
Student:	Martin Drozdík
Supervisor:	Ing. Marek Skotnica
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2020/21

Instructions

An open-source smart contract designer is an ongoing research project at the CCMi research group. The main goal of this project is to provide a visual language to design legal contracts between parties which can be used to generate executable blockchain smart contracts. Blazor is a technology that allows running .NET code directly in the browser using a WebAssembly standard.
A goal of this thesis is to investigate the possibilities of the Blazor technology to design an open-source smart contract designer and create a proof of concept prototype.

Steps to take:
- Review the Blazor framework.
- Review an existing open-source designer for the Das Contract project.
- Design and create a proof-of-concept application in Blazor.
- Summarize the benefits and potential of the Blazor compared to other approaches.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 20, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Open-Source Legal Process Designer in .NET Blazor

Martin Drozdík

Faculty of Information Technology
Supervisor: Ing. Marek Skotnica

May 6, 2020

Acknowledgements

Thank you to my supervisor, Ing. Marek Skotnica, for providing guidance and feedback throughout this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for nonprofit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 6, 2020 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Drozdík, Martin. *Open-Source Legal Process Designer in .NET Blazor*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstract

This thesis focuses on an exploration of a single page application framework called Blazor and a proof of concept implementation of a Blockchain Smart Contract editor.

The Smart Contract editor has the potential to create decentralized, autonomous and secure electronic contracts, using a user–friendly visual language DasContract. Such contracts could significantly reduce various administration tasks and in some cases eliminate the need for central authorities, such as banks.

In this thesis, a proof of concept implementation of a Smart Contract editor is analyzed, designed and implemented. The functionality of the editor was demonstrated in a case study of a mortgage contract. The editors' source code is publicly available for further research. This thesis also reviewed and summarised the benefits of Blazor and compared it to other single page application frameworks.

Keywords Proof of concept DasContract editor, Blazor summary, Blazor comparison, Smart Contract editor, C# and JavaScript interoperability, Blazor Web Assembly, client-side Blazor

Abstrakt

Tato práce se zabývá průzkumem frameworku Blazor a tvorbou prototypu editoru chytrých blockchain smluv.

Editor chytrých smluv by mohl být schopen tvořit decentralizované, autonomní a zabezpečené elektronické smlouvy, a to za použití vizuálního a uživatelsky přívětivého jazyka DasContract. Chytré smlouvy mohou významně snížit množství administrativní práce a v některých případech i eliminovat potřebu centrálních autorit, například bank.

V rámci této práce byl zanalyzován, navržen a implementován editor chytrých smluv. Funkčnost editoru byla demonstrována vytvořením hypoteční smlouvy. Zdrojový kód editoru je veřejně dostupný pro další potenciální výzkum. Tato práce také posoudila a shrnula přínosy frameworku Blazor a porovnala jej s ostatními SPA frameworky.

Klíčová slova Prototyp DasContract editoru, souhrn Blazoru, porovnání Blazoru, Smart Contract editor, interoperabilita C# a JavaScript, Blazor Web Assembly, client-side Blazor

Contents

List of Figures

List of Tables

List of Listings

Introduction

DasContract is a visual language capable of defining contracts [1]. This thesis aims to improve the usage of DasContract language by creating a proof of concept editor.

Motivation

Current society closes deals and arrangments using contracts. These contracts commonly take the form of signed papers with clearly stated conditions of the arrangement. If conditions of a paper contract are broken by an interested party, other parties must rely on a juridical system to enforce the conditions of the contract, which can take years and cost a significant amount of resources.

Blockchain Smart Contracts are a form of secure code, that has the potential to replace common paper contracts and eliminate the need of central authorities, because is it capable of enforcing certain conditions on its own [1].

Making Smart Contracts using a software code can be difficult, especially for non-technical people. Additionally, a code can contain issues and bugs. Errors in a Smart Contract, that is capable of handling money, can cause significant financial losses [2]. This issue can be potentially solved by replacing the contract creation process with a visual-driven and user-friendly language called DasContract [1].

Potentially beneficial technology, used for the editors' front-end development, would be the Blazor framework. Blazor is a SPA front-end framework developed using C# and Razor languages.

If Blazor would prove to be a functioning framework with rich and reliable features, it would be the first SPA framework embedded directly into the .NET ecosystem, developed by C# and capable of utilizing existing JavaScript packages and libraries [3]. This would make the Blazor framework a strong competitor against vast amount of existing JavaScript SPA frameworks.

Problem statements

Currently, there are no DasContract editors, that are capable of creating Smart Contracts using the newest version of the DasContract language. This thesis aims to create a proof of concept editor of Smart Contracts using the DasContract language.

The DasContract editor could enable lawyers and other non-programmers to create Smart Contracts with easy to use interface and without the need to learn complex programming languages.

Blazor is a SPA framework used in this thesis, but it is uncertain if it currently offers all the required features on high enough standards. By making a project entirely based on the Blazor framework, that has high demands on the User Interface (UI), it would thoroughly test the framework and suggest, if Blazor is ready for production development or not.

Objectives

The main objective of this thesis is to investigate the possibilities of the Blazor technology, analyze, design and implement an open-source DasContract designer and create a proof of concept prototype.

The output of the Smart Contract designer must be data, structured in a universal format, such as Extensible Markup Language (XML) or JavaScript Object Notation (JSON), so it can be easily understood by diverse DasContract interpreters.

Blazor possibilities investigation must cover JavaScript interoperability and evaluate if the Blazor framework is capable of using existing JavaScript packages and technologies, that are not available for Blazor. Good practice approaches must be thought out and summarized.

Objectives of this thesis do not include developing the DasContract language, creating DasContract interpreters or stores, working with Blockchain or Smart Contracts, interpreting or simulating process models besides extracting activities and their information.

Structure and Methodology

The thesis is organized as follows:

- In chapter 1, the Blazor framework and some of its libraries, such as JavaScript interoperability, are investigated.
- In chapter 2, Smart Contract technology is briefly mentioned, the existing editor is briefly analyzed and DasContract language foundations are defined.

- In chapter 3, analysis and design of the new DasContract editor are described.
- In chapter 4, implementation of the new DasContract editor is summarized.
- In chapter 5, advantages of using Blazor are summarized and a comparison of Blazor to other frameworks is presented.

Chapter 1

Review of the Blazor framework

Blazor is a single page application front-end framework directly embedded in the .NET ecosystem and recently partially released in the ASP.NET Core MVC 3.1 framework [4], [5]. Blazor is capable of compiling into the Web Assembly (WASM) format, making it capable of running in browsers and being developed using .NET languages. This makes it fundamentally different from frameworks such as Angular [6] or Vue.js [7]. Having a variety of .NET libraries and shared language between front-end and back-end can make a positive impact in the front-end development process.

1.1 Web Assembly

WASM is a standard that defines an executable binary instruction format for a stack-based virtual machine and interfaces for communication between a program and its hosting environment. WASM is currently shipped in Firefox, Google Chrome, Safari, Edge and other less mainstream browsers [8], [9].

The power of Web Assembly is the universality. Any code can be compiled into the WASM format, given that the language of the code provides a WASM compiler. Any browser that understands WASM can run any language that can compile into WASM. Such languages include C# (and .NET standard in general), C++, C or Rust [9], [4].

1.2 Blazor

Thanks to the Web Assembly standard, any language that provides a WASM compiler can be compiled and executed in a web browser. This opens the possibility of creating new front-end frameworks, which are not entirely based on JavaScript or TypeScript. One of these new frameworks is Blazor.

Blazor is a single page application front-end framework developed by Microsoft [4]. Blazor is currently partially released in the ASP.NET Core frame-

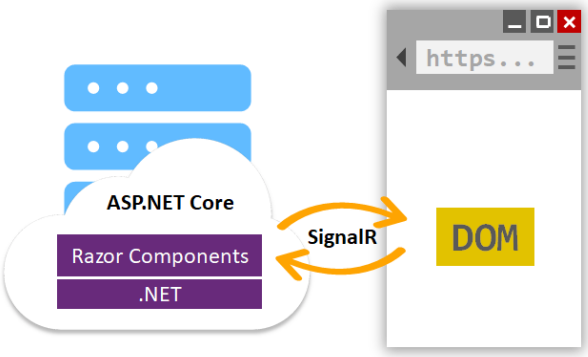

Figure 1.1: Server-side communication diagram [4]

work (version 3.1) [5].

Blazor applications are primarily developed using Razor language. Razor language can be described as a mixture of dynamically generated Hypertext Markup Language (HTML) using C# language and directives. Razor within the Visual Studio IDE offers strongly typed syntax, which makes it an excellent language for big–scaled and long–time projects.

Blazor currently supports two modes:

- Client-side
- Server-side

1.2.1 Server-side Blazor

Server-side Blazor, as shown at figure 1.1, computes all Document Object Model (DOM) related changes on the server-side and the client receives only DOM change commands. The communication between the server and the client is handled using SingnalR and uses a blazorpack protocol. This approach does not require Web Assembly support and offloads the DOM and virtual DOM difference computations to the server [4].

Server-side approach can bring better performance to the client. Unfortunately, it works well only if the latency is not slowing down the SignalR communication [4].

The major property of the server-side Blazor is that it does not require to be compiled into WASM, making is perfectly capable of utilizing any .NET Core library. In other words, the whole front-end can be in the same project as the back-end. Server-side Blazor can for example directly invoke a database query or safely handle authorization [4].

The capability of server-side Blazor being and executing all while within a .NET Core project also means, that existing ASP.NET Core applications, that use Razor views or Razor pages can load and easily use Blazor components in

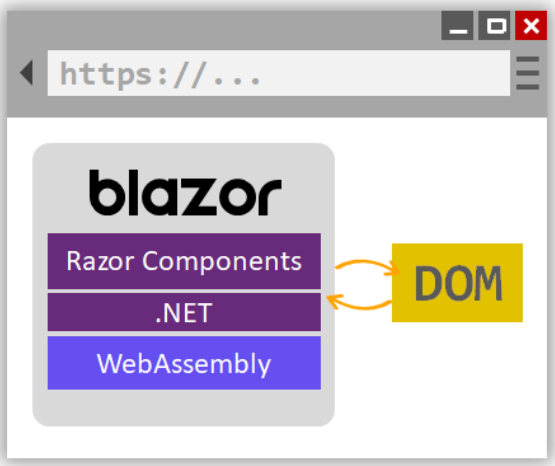

Figure 1.2: Client-side communication diagram [4]

the server-side mode. This means extending older applications using server-side Blazor is very approachable.

1.2.2 Client-side Blazor

Client-side Blazor, as shown at figure 1.2, is a more "traditional" Hypertext Transfer Protocol (HTTP) approach, where the entire page and resources are sent to the user. This requires browser Web Assembly support because the C# libraries and Razor language are compiled into WASM to be able to be executed within the clients' browser [4].

Client-side Blazor must be in a project that targets .NET Standard 2.0 or .NET Standard 2.1 framework. This project is being compiled into WASM [4]. This means that it cannot be in the same project as a server, that is, for example, an ASP.NET Core. Because of that, the client-side Blazor cannot directly invoke the back-ends' methods or safely handle validation, like for example the server-side Blazor. Communication between those two layers can be established for example with an Representational State Transfer (REST) Application Programming Interface (API) interface and JSON.

1.3 Java Script and C# interoperability

Blazor is unable to do everything JavaScript can. Furthermore, existing powerful JavaScript libraries cannot be compiled to or understood by C#. That is why .NET provides interoperability for JavaScript (JS interop).

It is possible to invoke JavaScript function by name, all within the Blazor framework and Blazor components. Those functions must be available in the

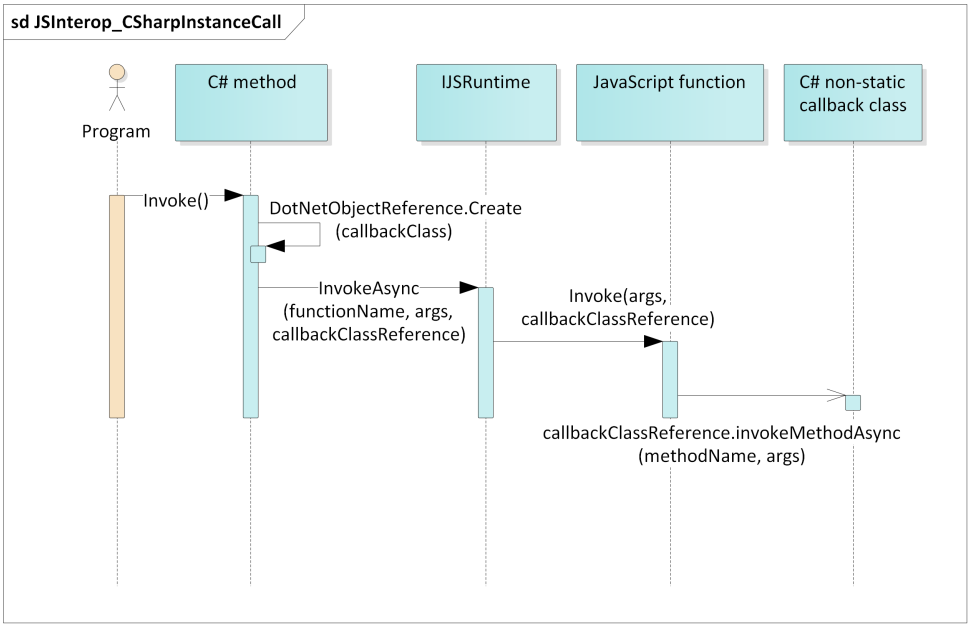

Figure 1.3: Sequence diagram of invoking a JavaScript function with the intention to callback an instance-specific C# method

global JavaScript scope. The JavaScript function call is weakly typed [4].

It is also possible to invoke C# methods from within JavaScript, but it requires extra steps since C# is not an interpreted language (like for example JavaScript). First of all, C# methods to be invoked from JavaScript must be provided with the `JSInvokable` attribute. Secondly, if it is required to call a non-static method, C# must firstly provide a class callback reference, meaning it is required to call the JavaScript method with callback reference as a parameter first and only then can JavaScript invoke the method earlier referenced by C# (as shown at figure 1.3) [4].

1.4 Summary

Blazor is a single page application front-end framework directly embedded in the .NET ecosystem and recently partially released in the ASP.NET Core MVC 3.1 framework.

Blazor applications are written in Razor and C# languages. It can compute all operations on the server-side or it can be compiled into the WASM format and executed directly in a browser.

Blazor provides two-way interoperability with JavaScript to support JavaScript libraries and not (yet) implemented functionalities.

Chapter 2

Smart Contract designer

The DasContract designer is the first step in creating fully-fledged automated model-driven Smart Contracts. The designer can enable lawyers, amongst other users, to design a Smart Contract [1].

2.1 Blockchain

Blockchain technology introduced by Satoshi Nakamoto is a type of trustless network. The technology and its underlying properties offer a simplistic, yet robust peer-to-peer network structure [10], [11].

A research article "A visual domain-specific language for modeling Blockchain Smart Contracts", that introduces the first public version of DasContract, summarized Blockchain as:

"*Blockchain is a technology introduced by Satoshi Nakamoto. It is mostly known for its use with Bitcoin as it is its underlying technology. It is a new way of looking at transactions, assets exchange or even whole organizations. It introduces a decentralized, autonomous, replicated and secure database. Based on cryptography offers a trustless network with no need for an intermediary, resulting in major resource and also time-saving. The possibilities of applying this technology are very broad and it could be effectively used in most of the parts of our world.*" [1]

As the research article states, Blockchain provides "a decentralized, autonomous, replicated and secure database", which are qualities especially required when dealing with electronically operated legal contracts.

2.2 Smart Contract

Smart Contracts, as described by Nick Szabo, are a new form of contract, that is smarter and more functional than commonly used paper contracts [12].

A research article "A visual domain-specific language for modeling Blockchain Smart Contracts", that introduces the first public version of DasContract, summarized Smart Contracts as:

"*The idea of Smart Contracts is to offer more complex solutions than just sell/buy transactions. A Smart Contract is a transaction embedded in the Blockchain that contains enhanced logic – an executable contract, has its data storage and can access other resources to evaluate its current state and perform actions – a contract made of code.*" [1]

Smart contracts are – simply put – an extension of Blockchain, that enables programmers to write and run more complex code. Smart Contracts are an essential tool for creating electronic legal contracts, that are decentralized, autonomous and secured.

Blockchain and Smart Contract properties implicate almost impossible tempering with already deployed legal contracts. This means deployed contracts are secured from harmful or vicious intentions from any entities, even the ones that deployed the contract in the first place.

One of the major Smart Contract disadvantages is difficult bug handling on deployed contracts, which can be hard to fix without precautions (such as kill-switches). These bugs can cause significant financial losses [2].

2.3 The existing Smart Contract designer (1.0)

By the time of making this thesis, a 1.0 version of a Smart Contract designer exists. The newly developed editors' version is 2.0.

The 1.0 editor uses:

- Design and Engineering Methodology for Organizations (DEMO) modeling to define contract processes,
- Object Fact Diagram to define contract data models,
- Blockly editor with Solidity support for defining action rules.

The primary reason the 2.0 version of a DasContract designer being developed is due to the fact, that the 1.0 version implements an older DasContract language, which heavily depends on DEMO models to define contract processes [13]. The current DasContract model replaced the DEMO models for a subset of Business Process Model and Notation (BPMN) models, which makes the 1.0 version outdated [14].

The second reason for upgrading the 1.0 DasContract designer is the architecture and framework changes. The 1.0 DasContract designer follows the Flux design pattern, which enables it to let multiple users work on the same contract, with Angular as its SPA front-end framework [13]. The new DasContract editor will be designed more towards the Model-View-Controller

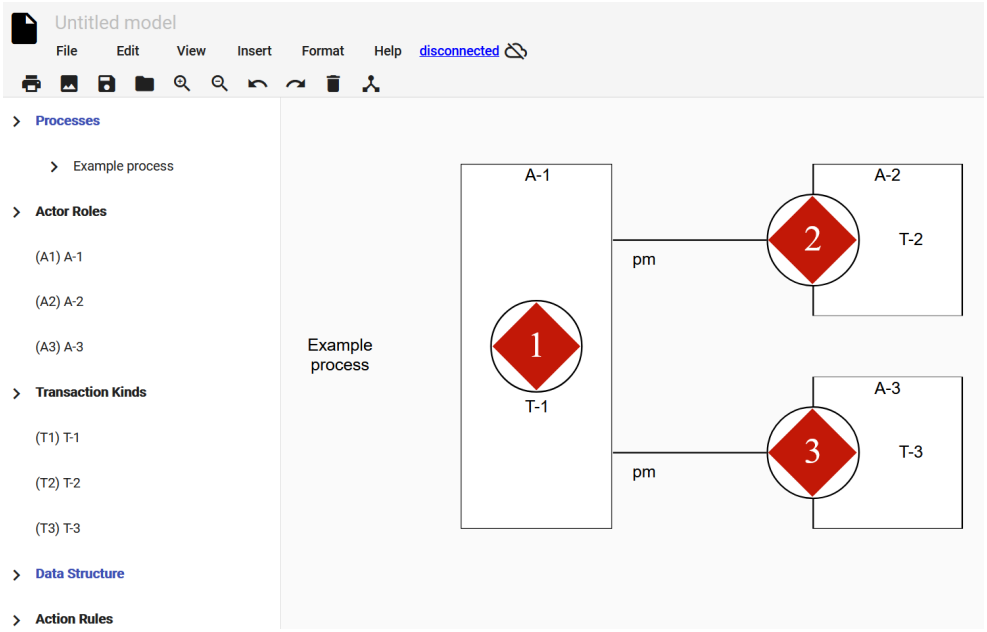

Figure 2.1: Screenshot of the 1.0 editors' DEMO modeler

(MVC) design pattern to maintain the approach as simple as possible. The front-end SPA framework will be swapped for the Blazor framework. The Blazor framework with ASP.NET Core as the back-end will enable shared models between those two layers and exploration of the Blazor framework properties and possibilities.

2.4 The new Smart Contract designer (2.0)

The new Smart Contract designer (version 2.0) uses the latest DasContract language to defined Smart Contracts. DasContract is a visual model-driven language that dictates, how contracts should be structured and what diagrams should be used [1].

The 2.0 Smart Contract designer can be separated into two main editors:

- Contract processes and activities
- Contract data model

2.4.1 Process and activities

A contract can be described by a process. The process is composed of activities and their relations.

To describe a process and its activities, the 2.0 Smart Contract editor uses a subset of a BPMN language [14]. The BPMN offers a great modeling language for business processes, which can be translated into useful and universal code structures [15].

The selected subset of BPMN contains the following types of modified activities:

- User
- Business rule
- Script [14]

The User activity is defined by a form, which needs to be filled and submitted, to continue to the next activity. The form fields must be bound to data model properties. The field–property binding determines the field data type and provides easy and intuitive access to the form data [14].

The Business rule activity is defined by a Decision Model and Notation (DMN) diagram. The DMN diagram is simply a table of inputs and their corresponding outputs [16].

The Script activity is defined by a manually written script.

2.4.2 Data model

The 2.0 Smart Contract editor handles data access, persistence, and manipulation using data models and their properties [14]. Data models can be also known as classes or entities.

Data models contain one or more properties. These properties can be one of the following data types:

- Integer
- Unsigned Integer
- Boolean
- String
- Date and time
- Address
- Billable address
- Data
- Reference to other model [14]

These models can be instantiated within the contract process and their properties can be manipulated using scripts or bound to a form field.

2.5 Summary

The new Smart Contract editor is a 2.0 version of an existing 1.0 version Smart Contract editor.

There are two main reasons for making a completely new editor. The first reason is that the DasContract language, used to define Smart Contracts, changed. It swapped model to defined processes from DEMO to a subset of BPMN. The second main reason is changing the application design from Flux to MVC and replacing the front-end framework with Blazor.

The 2.0 editor uses a subset of BPMN to describe processes. A process can be described as activities with relations between them. The BPMN subset defines three types of activities:

- User activity is a form that users must fill out.
- Business activity is defined by the DMN model.
- Script activity is defined by a code.

The 2.0 editor uses data models (classes/entities) to handle data persistence, data binding, and data handling. A data model editor is also a part of the 2.0 Smart Contract editor.

Chapter 3

Analysis and design of the new editor

This chapter describes an analysis and design of the new DasContract editor. The analysis includes functional and non-functional requirements, business processes and use cases. The design focuses on high-level planning, layout, layering and class diagrams.

3.1 Analysis

The analysis of the proof of concept implementation consists of Unified Modeling Language (UML) activity diagrams, use case diagrams and functional and non-functional requirement diagrams.

3.1.1 Business processes

The DasContract editor can be described with only one business process, which is creating and editing a contract. This is due to the fact that the editor is only a proof of concept implementation and is supposed to offer only essential functionalities required for it to work, which is simply editing contracts.

The contract creating and editing process is described in figure 3.1 using UML activity diagram.

3.1.2 Use cases

Based on the business process from figure 3.1, a UML use case diagram could be thought out and drawn. Use cases are:

- Create contract
- Publish contract

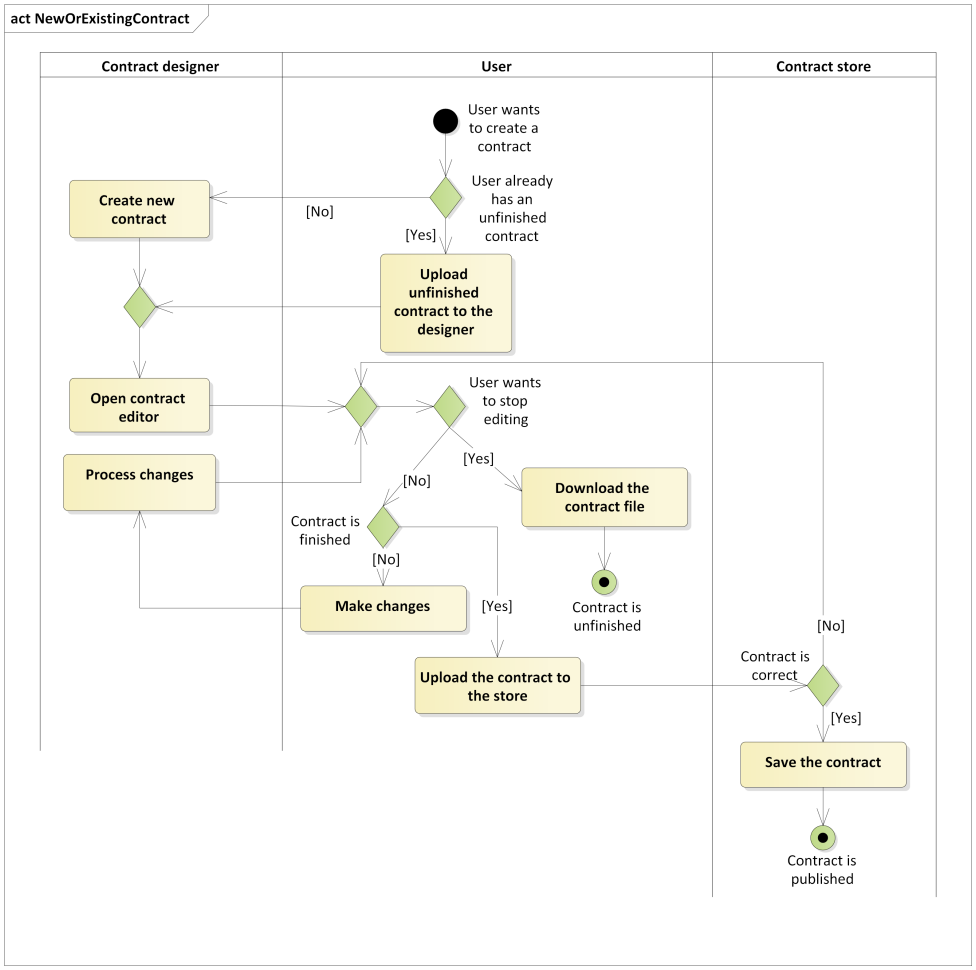

Figure 3.1: Process diagram that shows the contract creation process

- Upload existing contract
- Edit contract
  - Edit contract data models
  - Edit contract process
  - Edit contract activities
- Save contract

Relationships between use cases are visualized using the use case diagram in figure 3.2.

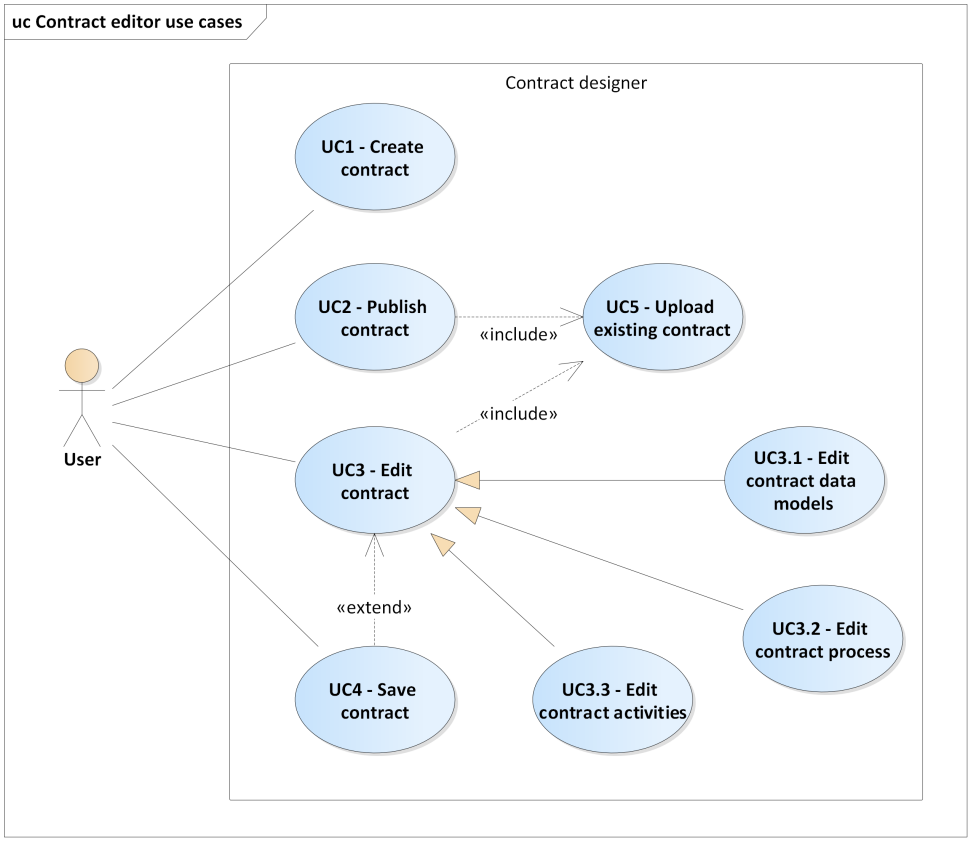

Figure 3.2: Editors' use case diagram

3.1.3 Functional and nonfunctional requirements

Requirements are categorized into business requirements, functional requirements, usability requirements, reliability requirements, performance requirements, and supportability requirements.

Most of the requirements are business requirements, since the result of this thesis is supposed to be a proof of concept implementation, and it does not require any properties expected in a commercial environment.

A requirements diagram can be seen at figure 3.3.

3.2 Design

The application is designed as an MVC application. It creates several layers to eliminate as much potential refactoring and spaghetti code as possible. The front-end uses components to simplify complicated tasks, remove redundancies and maximalize reusability.

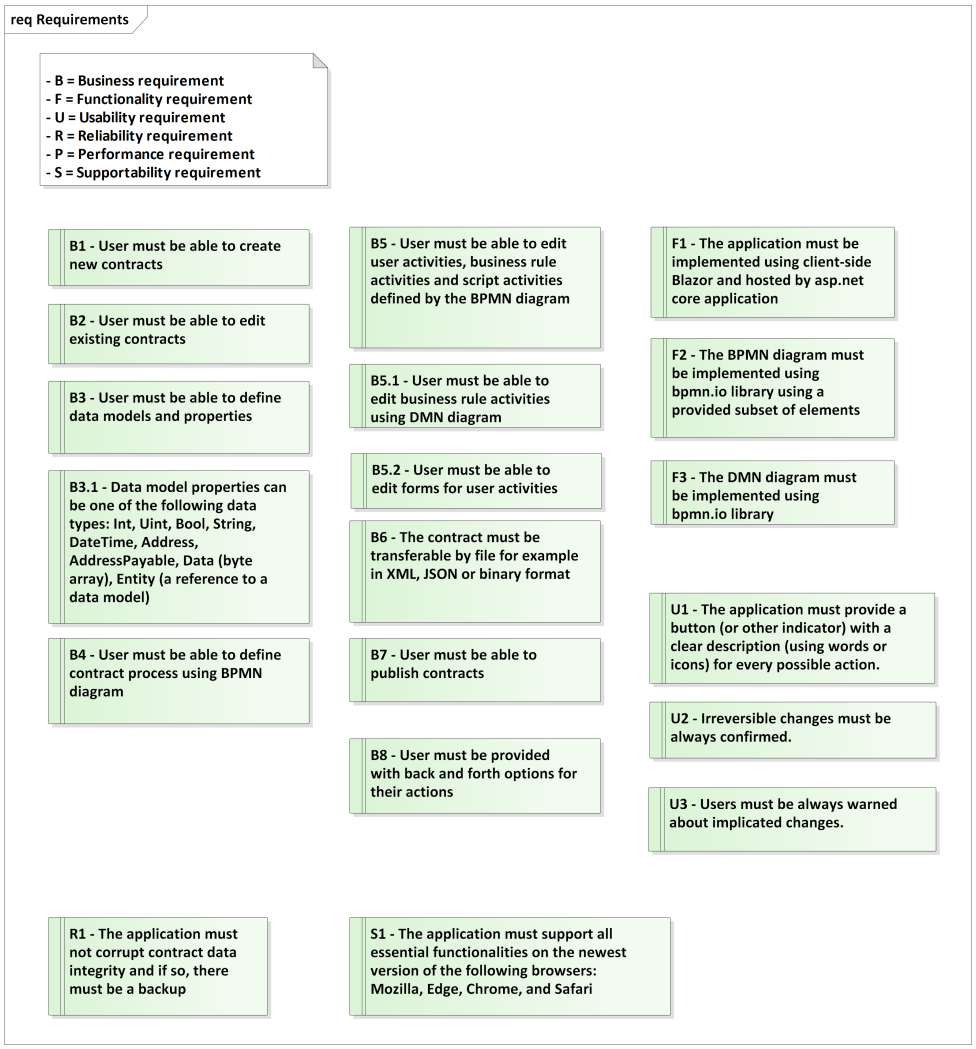

Figure 3.3: Diagram of editors' requirements

3.2.1 Package diagram

The editor is designed as an MVC application. The editor contains 6 essential packages:

- DataPersistence
- AppLogic (business logic)
- Server
- Pages
- Components
- Entities

The *DataPersistence* package contains repositories, that provide access to persistent data storage. This layer ensures that in case of a persistent data storage change, the rest of the application would not have to be refactored.

The *AppLogic* package contains applications' business logic.

The *Server* package uses business logic to provide an API. The API is consumed by various pages in the Pages package.

The *Pages* package contains services for communication with the Server API and page components, that are served to the user.

The *Components* package contains razor components used by various pages, such as a component "ContractEditor".

The *Entities* package contains all entity models shared across the editor. This package imports the *Migrator* package, which enables entities to be versioned.

Packages are visualized in figure 3.4.

3.2.2 Contract models

Figures 3.5 and 3.6 describe contract entities and their relations. The model is based on the newest DasContract specifications on the official repository [17]. The contract consists of two major parts: a data model and processes.

The data model is a collection of entities. Entities consist of properties, which can be primitive (such as numbers, strings, ...) or reference (pointing to another contract entity).

The processes currently support only one main process described by a subset of a BPMN diagram. The process consists of process elements and their connections – sequence flows. Process elements can be events, gateway or activities. Activities are an important part of the editor since their behavior must be further edited. Activities are either default, script, business or user activities. Their meaning and details are described in chapter 2.

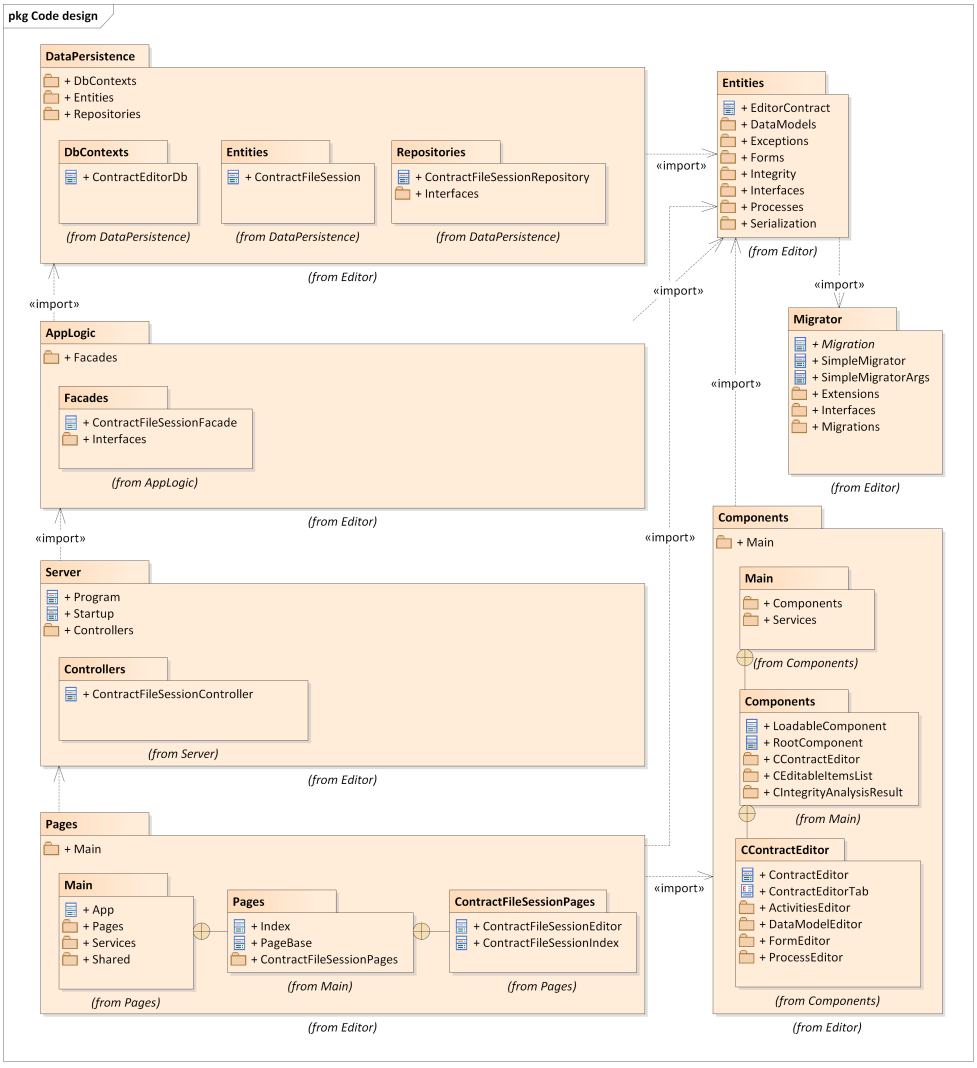

Figure 3.4: Diagram of editors' packages

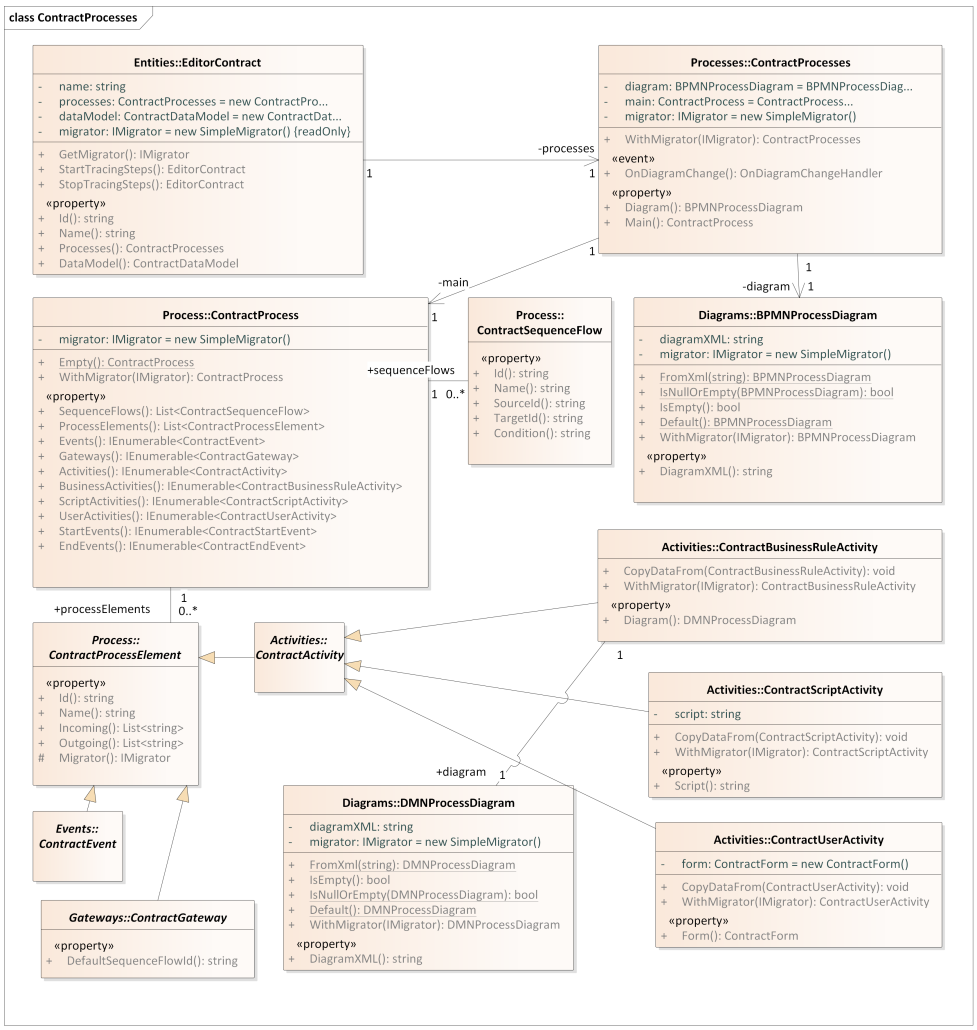

Figure 3.5: A class diagram that describes contract processes

3.2.3 Back-end

The back-end of the editor offers mainly persistent manipulation of file sessions. When a user wants to edit their contract, they will be able to upload the contracts' file and start editing. The upload invokes the creation of a new file session, that the server keeps track of. Sessions expire over time, which is handled by the DataPersistence layer.

File sessions exist so that when users accidentally refresh or close their browser window, the last saved contract from their session can be loaded and they do not lose all their progress. Furthermore, file sessions can offer features such as sharing contracts via a link, but that is not the intended nor safe to use functionality.

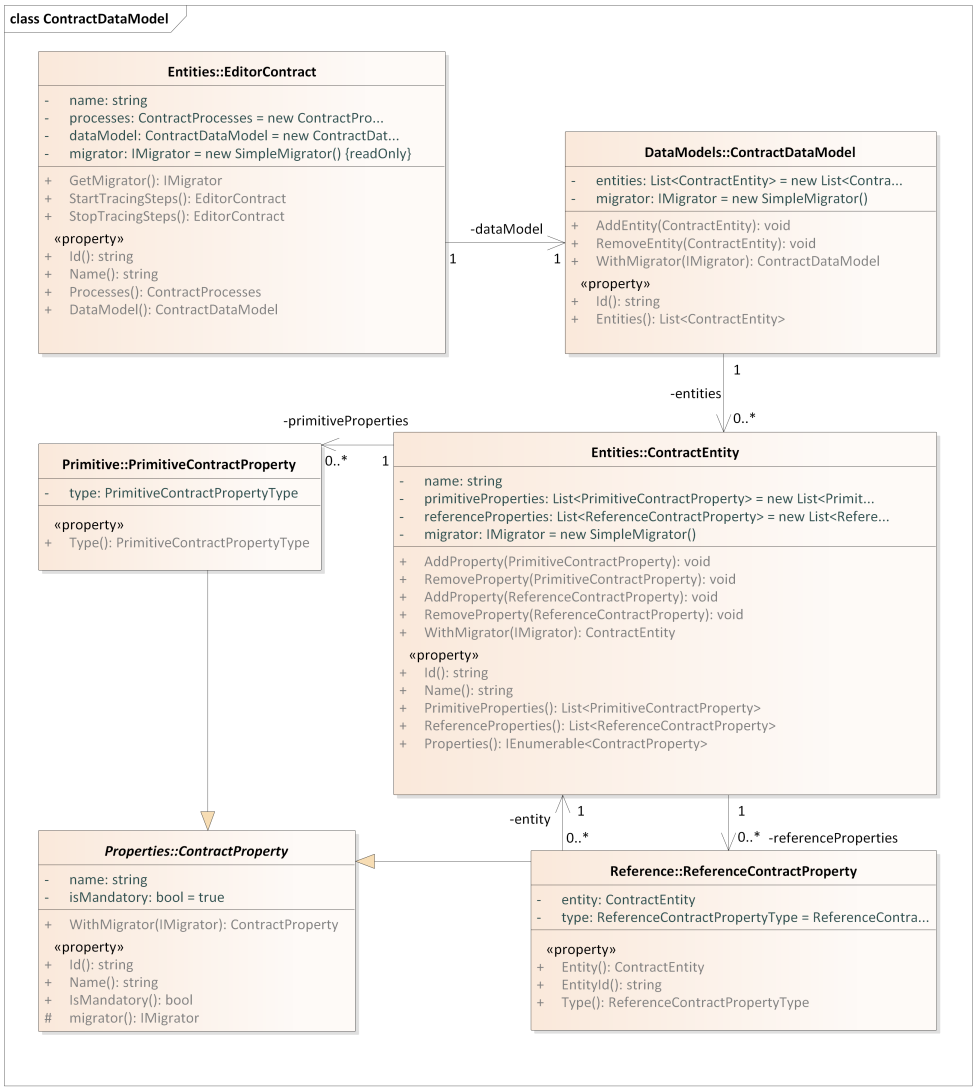

Figure 3.6: A class diagram that describes contract entities

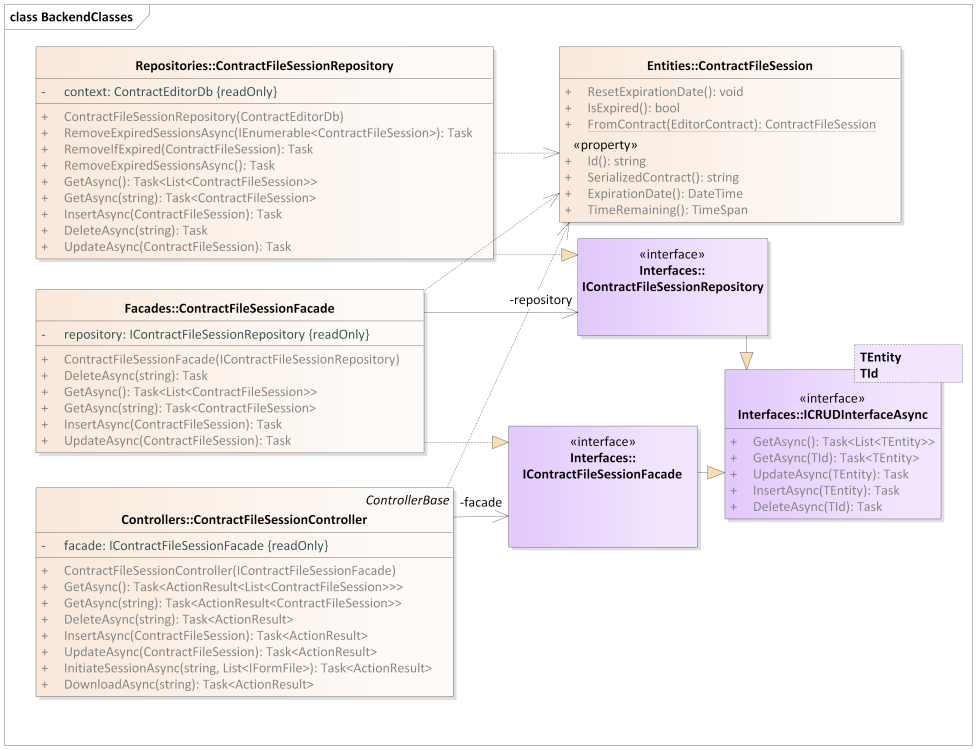

Figure 3.7: A class diagram that describes the back-end

Figure 3.7 shows a class diagram of the back-end. The repository handles communication with a persistent data storage and provides an interface. This interface is consumed by a facade, that provides access to entities within the AppLogic package via an interface. This interface is further consumed by a controller, that provides a REST API to be consumed by the front-end.

Figure 3.8 show an example of creating a new file session.

3.2.4 Front-end

Editors' front-end consists of components. The most complicated and important component is the contract editor. This editor accepts a contract instance as a parameter and renders the whole UI necessary to edit the contract.

The front-end uses injectable services for communication with the server API. In this case, there is only one service for handling file sessions, as shown in figure 3.10.

3.3 Summary

In this chapter, basic analysis and design of the proof of concept editor have been done.

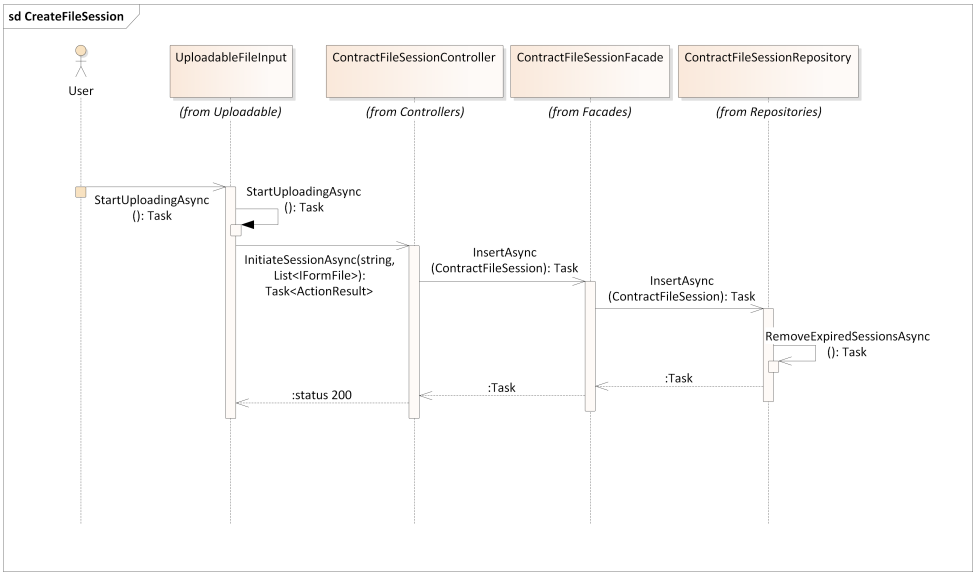

Figure 3.8: A sequence diagram that describes how a file session is created

The analysis includes functional and non-functional requirements, business processes and use cases. Business process is only one – editing a contract. This is since the implementation is only a proof of concept. For the same reason, the requirements consist mostly of business requirements, that state how the contract must be edited and structured.

The design established applications' layering, back-end functionalities and front-end structure, services, and main components' decomposition. The application consists of 5 major packages – DataPersistence, AppLogic (business logic), Server, Pages, Components and Entities (models). Data persistence allows access to persistent data storage/storages, application logic provides business methods and operations, server provides a REST API and Components are consumed by Pages, that are served to the user. Entities are shared across all layers.

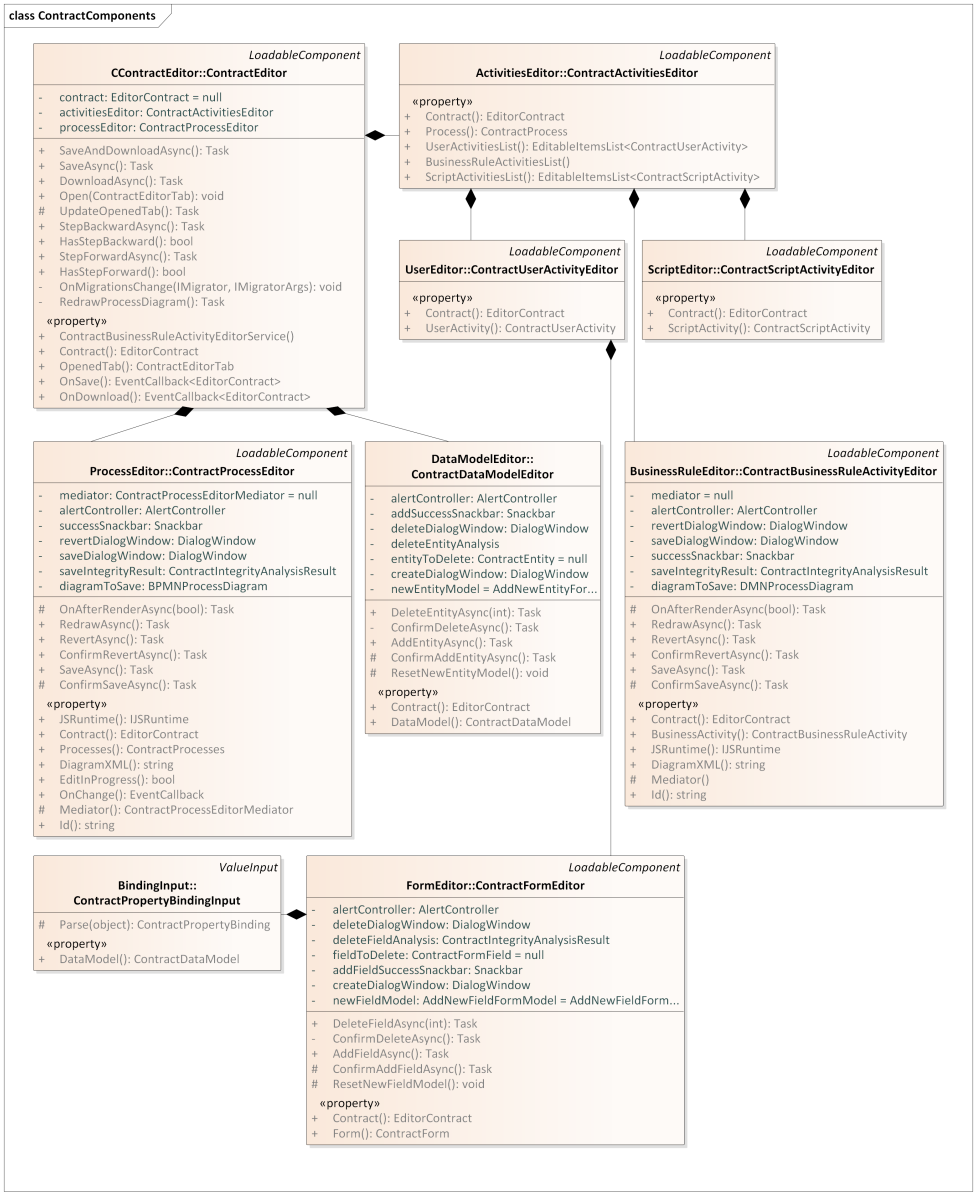

Figure 3.9: Diagram of important front-end components

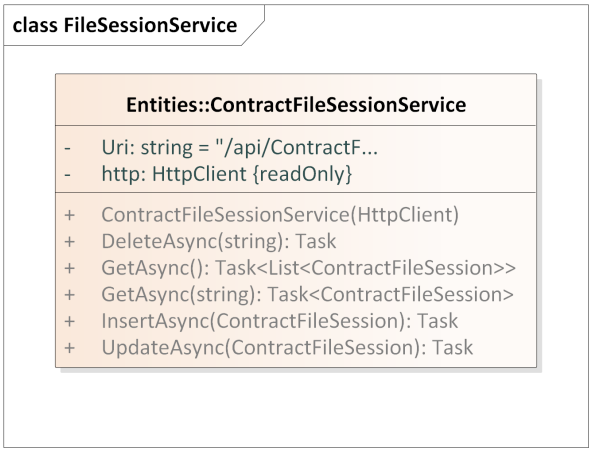

Figure 3.10: A service for communication with the back-end

Chapter 4

Proof of concept implementation

In this chapter, the implementation of the proof of concept editor is discussed and some unique details and facts that are not describable in the design from chapter 3 are presented.

Firstly, used technologies and important libraries are mentioned. Then, the development process and underlining details are described, including the JavaScript interoperability. After that, the integration between the front-end and the back-end is described. Testing the application is also mentioned, including selected testing details. At last, one contract case study is presented.

4.1 Used technologies

The back-end of the application is implemented using the ASP.NET Core MVC 3.1, which is a "*rich framework for building web apps and APIs using the MVC design pattern*" [18]. This version is currently the latest and it fully integrates the server-side Blazor. The client-side Blazor is currently in the preview, but since the server-side Blazor shares Razor syntax and service interfaces with the client-side Blazor, it is safe to say, that there is little to no risk of complicated refactoring with the first stable release of the client-side Blazor [5].

Persistent data in the development environment is stored in a local Microsoft Structured Query Language (SQL) database and the production environment uses an SQLite database. "*SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine.*" [19]

The front-end of the application is implemented using a SPA framework Blazor. This framework is described in chapter 1.

Blazor components are accompanied by stylesheets and scripts. Stylesheets

are developed using Syntactically Awesome Style Sheets (SASS). SASS is a Cascading Style Sheets (CSS) preprocessor, that extends the CSS syntax [20]. Scripts are developed using TypeScript. "*TypeScript is a language for application-scale JavaScript. TypeScript adds optional types to JavaScript that support tools for large-scale JavaScript applications for any browser, for any host, on any OS. TypeScript compiles to readable, standards-based JavaScript.*" [21]

Front-end resources, such as stylesheets and scripts are compiled and packed using webpack. "*At its core, Webpack is a static module bundler for modern JavaScript applications. When Webpack processes an application, it internally builds a dependency graph that maps every module a project needs and generates one or more bundles.*" [22]

The front-end UI is build on a Material Bootstrap framework. This framework expands the Bootstrap framework by restyling it as material design and adding components iconic for material design, such as the ripple effect [23].

For BPMN and DMN editors, *bpmn-js* and *dmn-js* libraries were used. These libraries are Open Source libraries developed by Camunda [24].

4.2 Development process

The development process of the back-end went smoothly. Using already established patterns and design, no unexpected errors, bugs or design flaws were found. The back-end of the application is implemented according to the design created in chapter 3.

The front-end is implemented according to the design created in chapter 3, but during the development, some issues were encountered.

The biggest and still existing issue is with the BPMN and DMN editors. The packages work fine, without any known bugs. The problem is the lack of documentation of the editors. Since the contract process is defined by a subset of BPMN, some features need to be cut from the BPMN editor. This is however very difficult without any documentation or existing examples. Hotfix of this problem, currently implemented in the contract editor, is to hide extra features using stylesheets. Other development issues caused by the lack of documentation and typings include unknown configuration settings, unknown event loops, unknown list of files that need to be included and much more.

No issues or bugs were encountered with the Blazor framework.

4.2.1 JavaScript interoperability patterns

Since the Blazor is unable to do everything JavaScript can, as described in chapter 1, interoperability invokes between Blazor and JavaScript must be utilized. This consumes extra development time, if proper and strongly typed development process needs to be maintained. For this thesis, best practice approaches for several use cases were thought out and summarized.

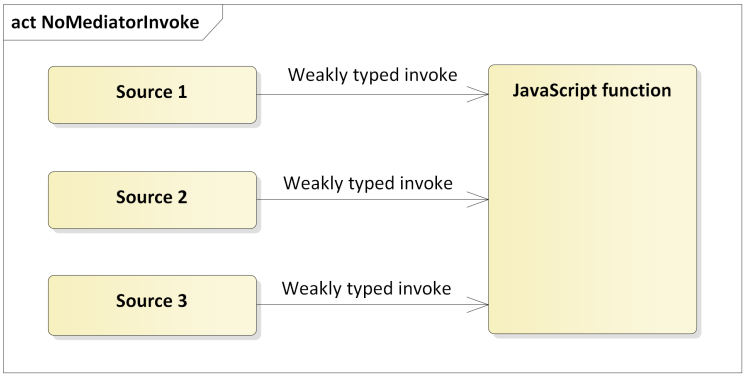

Figure 4.1: A diagram illustrating interoperability invokes without a mediator

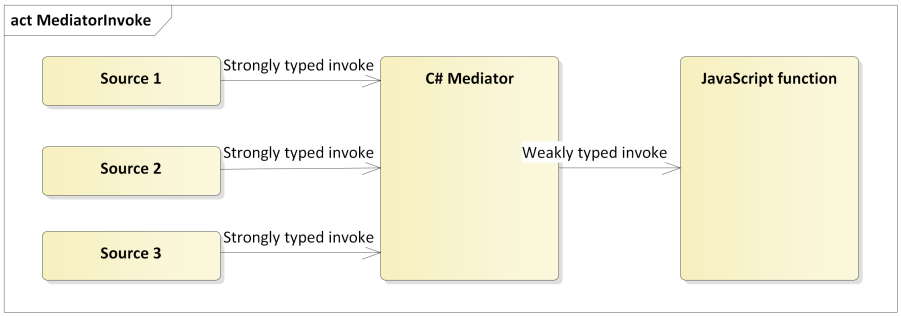

Figure 4.2: A diagram illustrating interoperability invokes with a mediator

JavaScript functions can be invoked using an implementation of the IJSRuntime interface. This implementation is provided by the Blazor framework and injected if required.

JavaScript functions can be invoked by simply invoking a method on the IJSRuntime interface, but the invoke is weakly typed. This means potential problems with using, maintaining and refactoring, such as typos, data type mismatches and more. This issue can be partially resolved by adding a "mediator" layer between C# and JavaScript.

The mediator layer simply streamlines the JavaScript invokes through a C# interface. This prevents multiple weak invokes, as shown in figure 4.1. The only weakly typed invoke is done inside the mediator, as shown in figure 4.2.

Situations with JavaScript interoperability, encountered while developing the Smart Contract editor, can be categorized into two groups:

- Calling a JavaScript function with parameters,
- Calling a JavaScript function with parameters and requesting a callback.

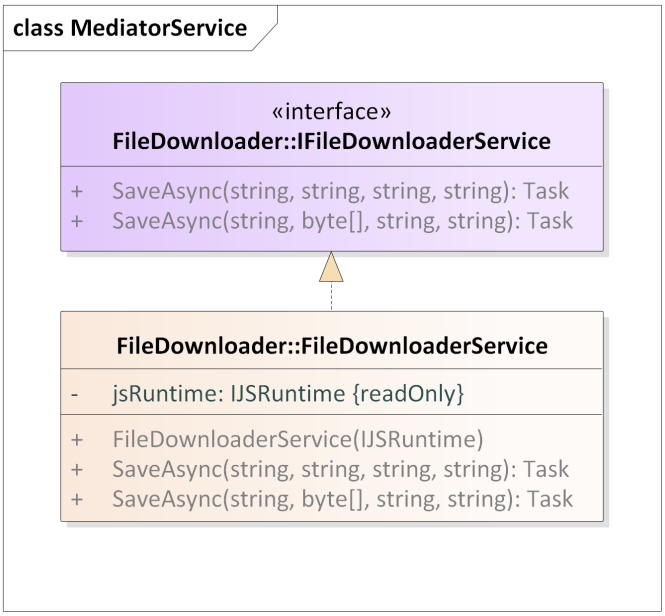

Figure 4.3: A class diagram of a mediator service

To simply call a JavaScript function, one possible approach is to create an injectable singleton service, that acts as a mediator. The service mediator ensures strongly typed streamlined invokes, easy extendability, shareability, and refactoring. An example can be seen in figure 4.3.

To invoke a JavaScript function with expected callback back to C#, one possible approach is to create a new transient mediator class, with a lifespan of each invoke and callback (or callbacks) with the following methods:

- A method that invokes the JavaScript function and requests a callback.
- A method with "JSInvokable" attribute that is invoked from JavaScript as a callback and further invokes C# events or actions to inform observers.

An example of a transient mediator class that provides an event, that is invoked by a JavaScript callback, can be seen in figure 4.4. Figure 4.5 shows how to register an event and call a JavaScript method, which starts sending callbacks. Figure 4.6 shows how a callback invoked by a JavaScript is propagated all the way back to C# and mediator observers. An example of a code for such mediator can be seen in listing 4.1.

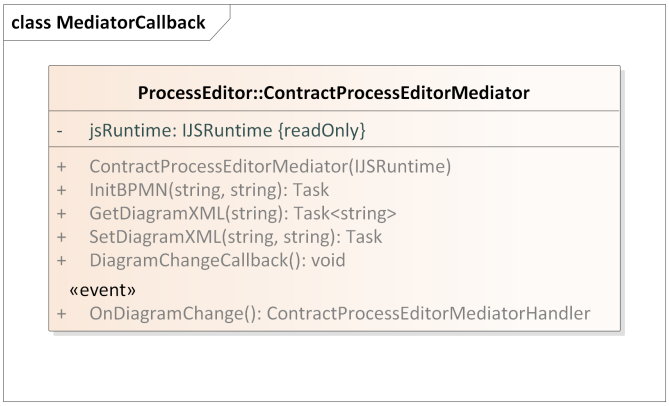

Figure 4.4: A class diagram of a mediator class capable of handling JavaScript callbacks

Listing 4.1 Example of a C# mediator

```
public class ContractProcessEditorMediator
{
    public event ContractProcessEditorMediatorHandler
        OnDiagramChange;
    readonly IJSRuntime jsRuntime;

    // ...

    public async Task InitBPMN(string id, string xml = "")
    {
        await jsRuntime.InvokeVoidAsync("InitBPMN", id, xml,
            DotNetObjectReference.Create(this));
    }

    // ...

    [JSInvokable]
    public void DiagramChangeCallback()
    {
        OnDiagramChange?.Invoke(/* ... */);
    }
}
```

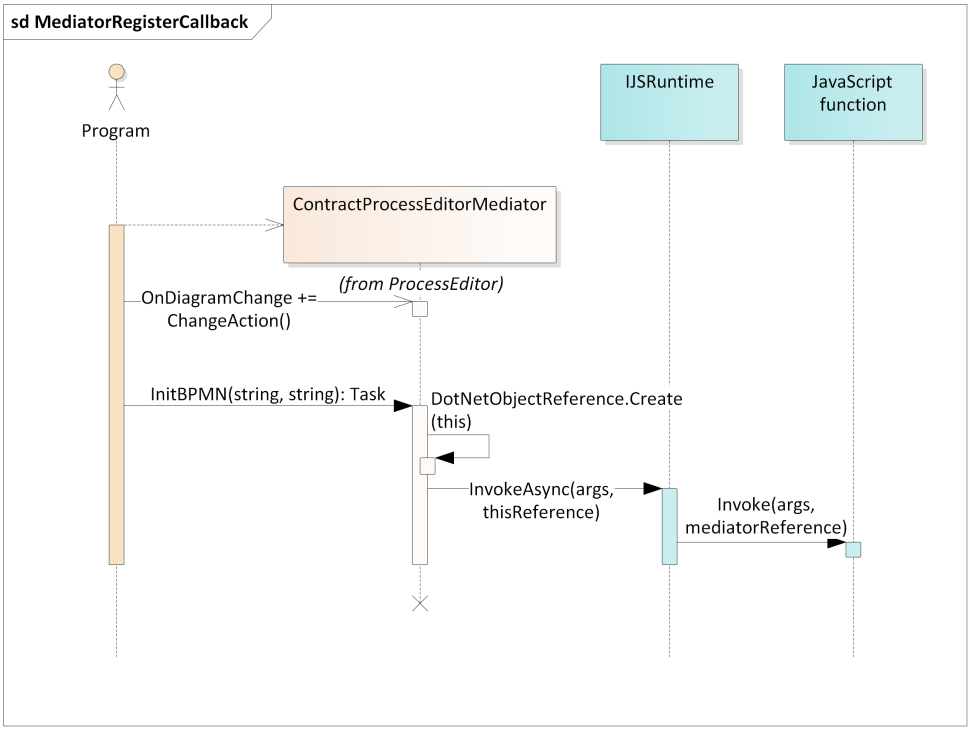

Figure 4.5: A sequence diagram that illustrates how to register a JavaScript callback using a mediator

4.3 Integration process

The integration between the back-end and the front-end of the proof of concept editor did not encounter any problems or difficulties. On the contrary, thanks to Blazors' properties, the integration was much more straightforward than if the front-end was built for example on Angular.

The communication with the back-end is achieved with a JSON REST API. This communication is abstracted using injectable singleton services. One such service is described in figure 3.10. The integration services can be injected into any Blazor component – thus any page.

Thanks to the Blazor framework, the project with entities shared between the back-end and the front-end can be referenced by both layers, making the serialization and deserialization process trivial. Additionally, no further modeling of entities had to be done, as it would be needed with frameworks such as Angular, assuming ASP.NET Core as the back-end framework.

For example, getting all sessions from the server can be done using a short code, that can be seen in listing 4.2.

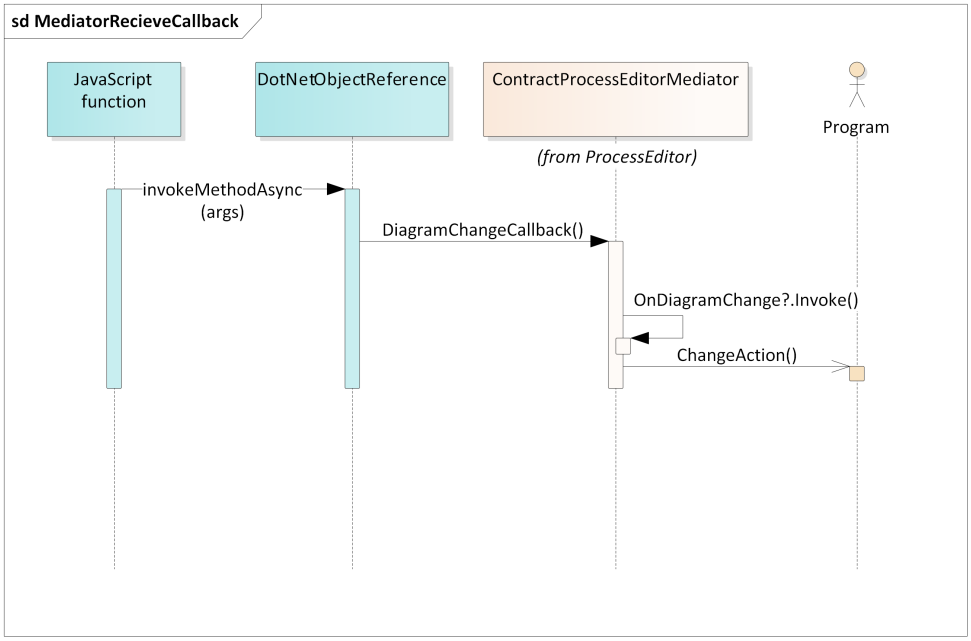

Figure 4.6: A sequence diagram that illustrates how JavaScript callbacks are handled

Listing 4.2 Request and deserialization of a class inside a service

```
public async Task<List<ContractFileSession>> GetAsync()
{
    // Reach the API and ensure success response
    var response = await http.GetAsync(Uri);
    response.EnsureSuccessStatusCode();

    // Deserialize and return
    var content = await response.Content
        .ReadAsStringAsync();
    return JsonConvert.DeserializeObject
        <List<ContractFileSession>>(content);
}
```

4.4 Testing

The proof of concept editor consists of two types of tests – integration and unit tests.

Integration tests are written in the xUnit framework and they ensure the servers' REST API correct functionality.

Unit tests are written in the NUnit framework and they ensure that individual classes and methods work correctly. The tests primarily include cases for data persistence repositories, entities serialization, entity logistics, entity integrity, and entity migrator.

The testing environment uses SQLite to mock a production database. The testing database is initialized and later disposed for each test case, making them still capable of running in parallel.

Although there are frameworks for unit testing individual Blazor components, such as bUnit, that enable automated testing, they were not used. The main reason is that Blazor components, in this case, are being refactored very often. This would result in long times spend on refactoring unit tests. Blazor components, in this proof of concept implementation, are tested in a separate project. The correct functioning is confirmed manually. The secondary reason is that the components often provide a visual value, which is hardly testable by code and the best solution is to manually check the visual correctness.

4.5 Case study

As a test that the proof of concept editor has been successfully analyzed, designed and implemented, one case study has been done. The case study is a Blockchain mortgage Smart Contract with a process taken from an open-source GitHub repository [25]. The process is displayed in figure 4.7.

The editor was able to successfully add all necessary data entities and set up user activities (and their forms). Script activities were not completed, because of the complicated nature, however, whoever can create Blockchain Smart Contract code, would be able to finish them. A cutout of the entity editor is displayed in figure 4.8 and a cutout of a user activities editor is displayed in figure 4.9.

The case study is not fully applicable in the production environment and is not completely correct. An expert in the mortgage industry and a Blockchain expert would be needed to fully design and implement a usable mortgage with the DasContract editor. However, the use case indicates, that there might a way to create secure powerful decentralized Blockchain Smart Contracts, that would improve the quality of life of many citizens. In this particular case, banks as a middleman could be eliminated and act only as a lender being managed by a mortgage contract.

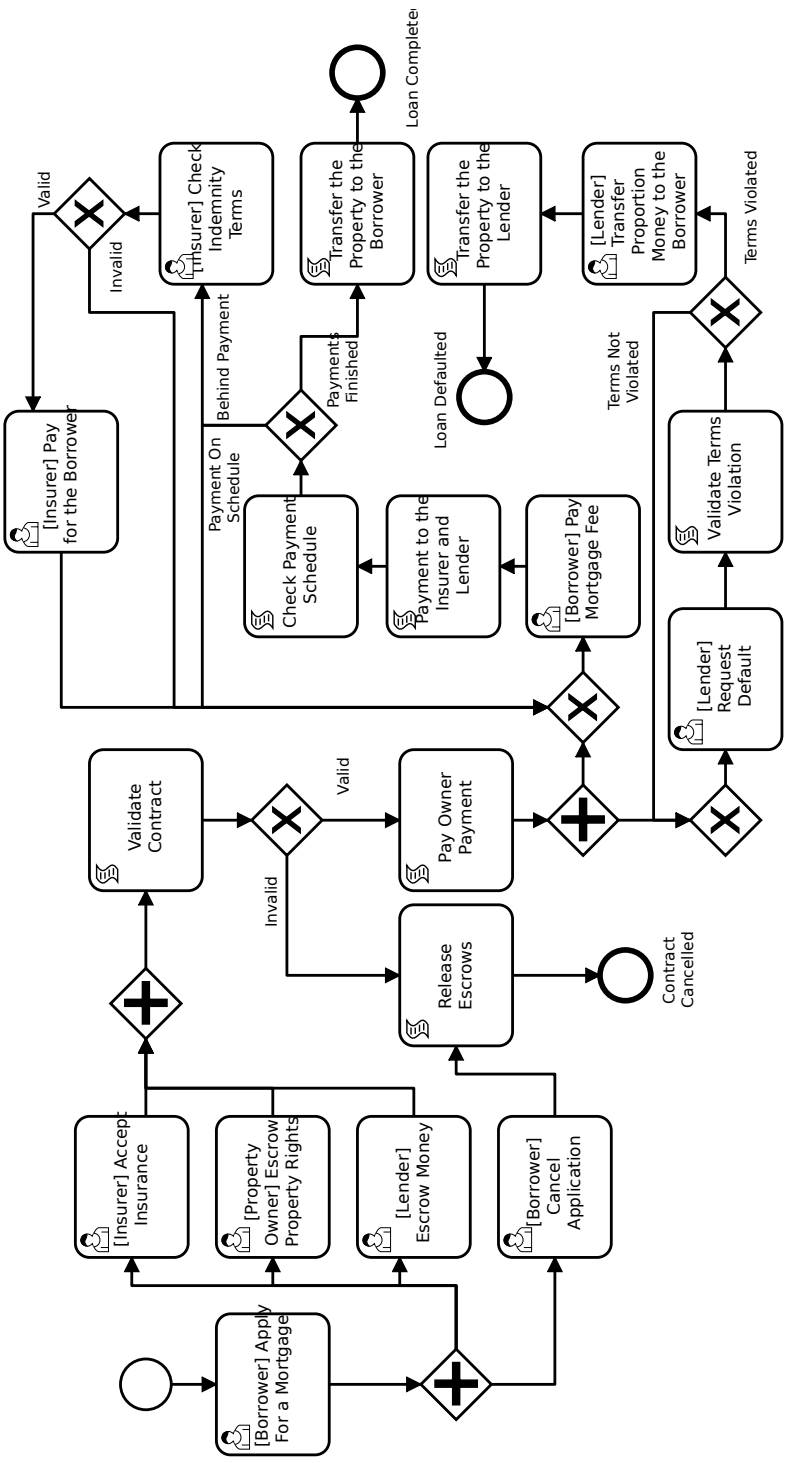

Figure 4.7: Process diagram of a mortgage contract

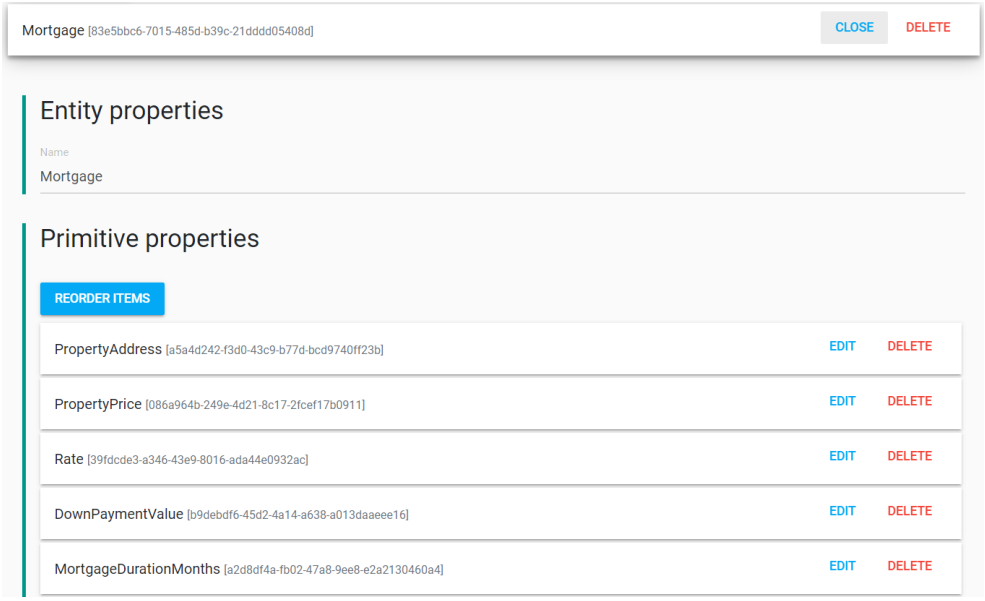

Figure 4.8: Part of a mortgage contract entities

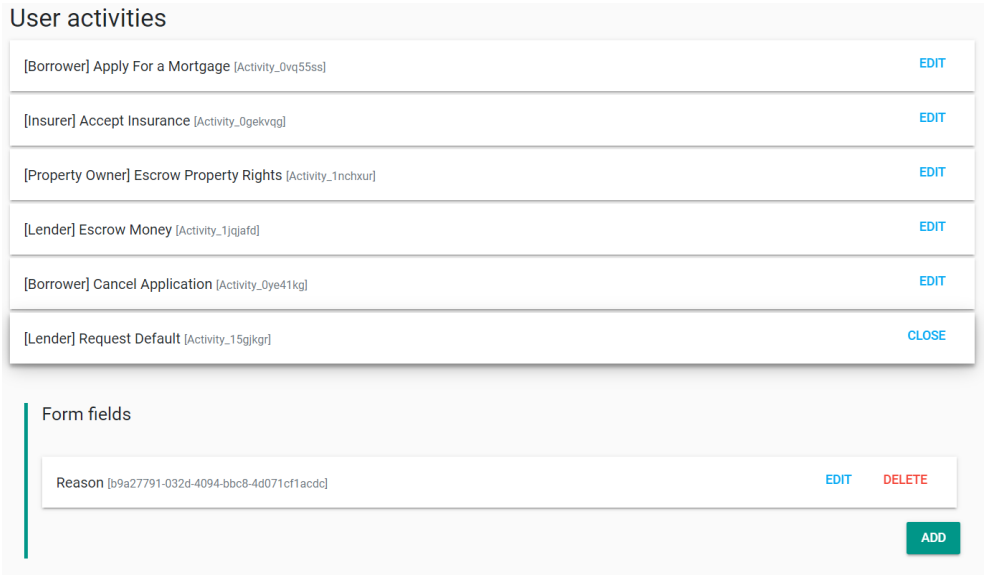

Figure 4.9: Part of a mortgage contract activities

The case study shows the idea of decentralization and displays, how the modern mortgage and other similar contracts could look like and function. This technology could bring many benefits to the lives of many.

4.6 Summary

This chapter described the used technologies, the development process, the integration process, testing details and one case study as a test, that the proof of concept editor is working correctly.

The development process went mostly smoothly because already established patterns and designs were used and the analysis and design, described in chapter 3, were followed and proved to be satisfactory. Also, JavaScript interoperability approaches were thought out and summarized. The biggest obstacle in the development process was the lack of documentation of the BPMN and DMN editor packages.

The integration process went smoothly thanks to the shared projects between the front-end and the back-end. The communication is established using a JSON REST API.

The automatic testing covers back-end REST API, data persistent repositories and entities. Blazor components are tested manually, since the test refactoring would take too much time. Blazor components also offer visual value, which is hardly testable by a code.

One case study has been done. The study focuses on creating a proof of concept mortgage contract and indicates, that the editor has been analyzed, designed and implemented successfully. In addition, the mortgage contract shows how the mortgage process could work in the future and improve the quality of life of many people, by acting as a middleman between all parties.

Chapter 5

Benefits of Blazor

The following chapter describes, how and under what circumstances can the Blazor framework benefit the development process. Furthermore, brief and high-order comparison of Blazor and other SPA frameworks and libraries has been done.

5.1 Languages

Applications based on the Blazor framework are developed using C# and Razor languages. Optionally, additional behavior can be achieved with JavaScript/TypeScript and styles can be added using CSS/SASS. The Razor language is used to mark up the dynamic HTML, binding, component usage and more.

Some front-end SPA frameworks, such as Angular [7] or Vue.js [6], presented their own language for setting up component templates. This presents a disadvantage when compared to the Razor language, which is commonly used in other areas, primarily in the ASP.NETs' views and Razor Pages. Razor components can even be reused in a multitude of ways, for example in static MVC rendering [18].

In addition to the Razor language, when developed in Visual Studio, it provides full IntelliSense support and strongly typed syntax, which is most certainly welcomed when working with 3rd party Razor Libraries or on enterprise scaled projects.

5.2 .NET platform

One of the most possibly lucrative properties of the Blazor framework is the embeddedness inside the .NET platform. More specifically, client-side Blazor currently targets .NET standard 2.1 interface (running on Mono), which includes a huge variety of System and 3rd party libraries. Server-side Blazor runs on .NET Core, which means, that it additionally can support all

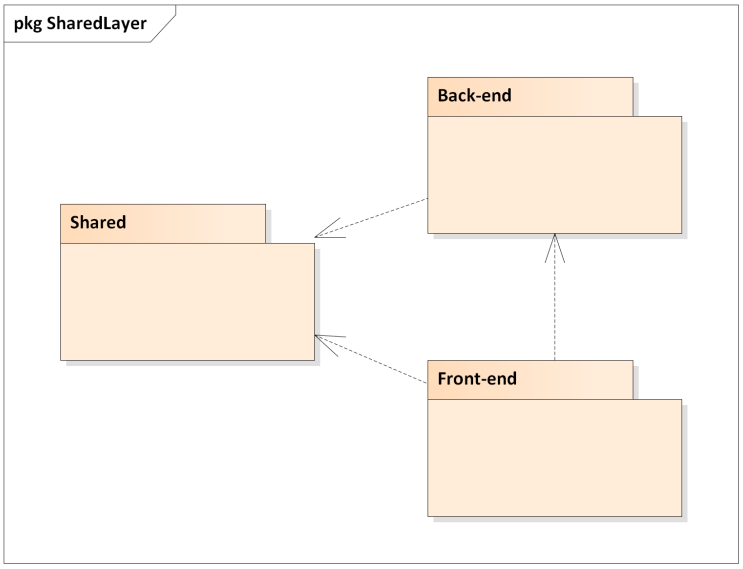

Figure 5.1: Component diagram of a web application with the same front-end and back-end language

.NET Core libraries. Furthermore, Blazor is developed using Razor and C# languages, which provide high performance and reliability [4].

The .NET platform for front-end can be lucrative not only because of the vast amount of libraries and development languages. Currently, there are no other existing front-end SPA frameworks, directly embedded inside the .NET platform that are developed using C# [3]. This means, that if the server was to be developed using a .NET framework, such as ASP.NET Core, using client-side Blazor as its front-end framework would result in uniting the back-end and the front-end under the .NET standard 2.1 interface.

By uniting the back-end framework (ASP.NET Core MVC) and the front-end framework (Blazor) under the .NET standard 2.1 interface, the possibility of shared libraries between those two layers is now opened. The most common type of a shared library would be database entity models. By having one .NET standard library with entity models, it is possible to use them both in the back-end and the front-end, which makes a whole lot of operations incredibly easy. For example, JSON/XML serialization and deserialization operations in communication can be straight-up trivial, with access to the same models.

Another good example of why it is excellent to share entity models with a front-end and a back-end is that the front-end and back-end data validations can behave consistently and be handled automatically. ASP.NET Core Entity Framework can even generate the corresponding database constrains.

More advantages of having the Blazor embedded inside the .NET framework can be expected in the future. For example, in the upcoming .NET 5,

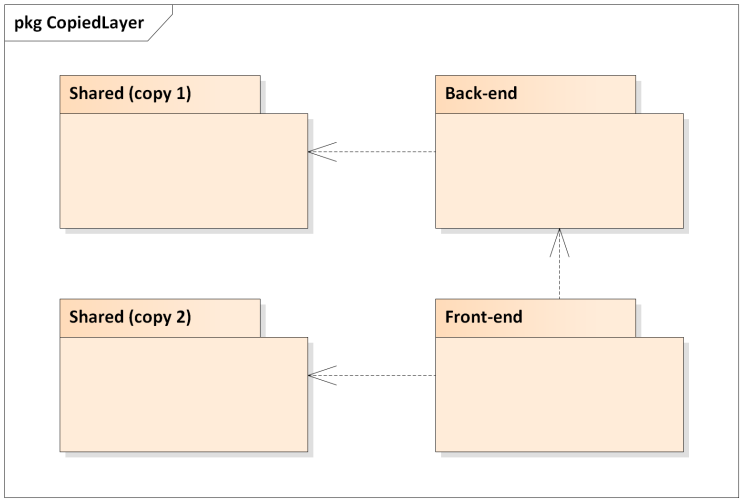

Figure 5.2: Component diagram of a web application with different front-end and back-end languages

one of the major themes is exploring Blazor usage within the WebWindow and the Electron libraries, which would make the Blazor capable of making desktop applications [26].

5.3 Comparison to other SPA frameworks

By comparison and from the programmers' point of view, Blazor does not provide any revolutionary ideas, that other frameworks, such as Angular or Vue.js, did not think of.

Blazor, just like other SPA frameworks, works with a virtual DOM to update the actual DOM as efficiently as possible, based on a data. Blazor, just as Angular [7] or Vue.js [6], provides reusable components with parameters, two-directional data binding, special markup language to easily write the components' render logic, services for routing, security, HTTP communication (and others), and more.

Blazor does offer some extra features, that frameworks such as Angular or Vue.js do not, such as templated components or cascading values, but these are not essential nor unreplicable.

From the technical point of view, Blazor is fundamentally different from Angular or Vue.js. The main reason is being a .NET framework. Blazor provides a unique implementation for all its services and its render logic.

Razor, used for components' render logic, is generated into a C# code. The generated code is enriched with `RenderTreeBuilder`, which handles all DOM related changes. Fortunately, the `RenderTreeBuilder` can be used by programmers without limitations, so any limits, that the Blazor framework

	Blazor	Angular	Vue.js	React
Primary dev. languages	C# and Razor	TypeScript	JavaScript	JSX
Components and data binding	Yes	Yes	Yes	Yes
Virtual DOM	Yes	Yes	Yes	Yes
Services	Yes	Yes	Not by default	Not by default
Full-blown framework	Yes	Yes	View layer only	UI library
Subjective complexity	Medium	Hard	Easy	Easy

Table 5.1: Comparison of various SPA frameworks and libraries

may have set, can be beaten with manually writing the rendering logic with the `RenderTreeBuilder`. Other frameworks, such as Angular, do not provide such low-level access [4].

5.4 Summary

Blazor is developed using C# and Razor languages. These languages are very performant and offer strongly typed syntax with full IntelliSense when developed in for example Visual Studio. This is an advantage when compared to for example frameworks developed with JavaScript.

Blazor is directly embedded in the .NET platform. Client-side Blazor implements .NET standard 2.1 interface running on Mono and server-side Blazor runs on .NET Core. This means an extensive variety of libraries available in the Blazor framework. Additionally, libraries under the .NET standard 2.1 interface can be shared with a back-end, making operations such as HTTP communication, serialization, and data validation easier and much more consistent.

From the programmers' point of view, by comparison, Blazor does not provide revolutionary ideas or tools, when compared to other SPA frameworks. It is just another SPA framework but in the .NET platform. The insides of the Blazor are however completely different.

Conclusion

The objective of this thesis was to investigate the possibilities of the Blazor framework, analyze, design and implement a proof of concept prototype editor capable of designing Smart Contracts using the DasContract language.

Analysis, design and a proof of concept implementation of the DasContract designer have been done and described. Additionally, one case study of a mortgage contract has been done to test the correctness of the editor. The editor succeeded in the creation of the mortgage contract and showed the possibilities and positive impact, that the editor and Blockchain Smart Contracts wield. The editors' source code is publicly available.[1]

The proof of concept implementation suggests, that the Blazor framework is suitable for production development. The .NET ecosystem does provide an enormously useful set of tools for the front-end development and other benefits, such as shared models between the front-end and the back-end save a huge amount of work.

This thesis also tested, that C# interoperability with JavaScript works and is capable of fully utilizing various JavaScript libraries, which makes the Blazor framework more lucrative, since it offers "backwards compatibility", if someone were to consider Blazor as a new framework to work with, instead of a JavaScript–based framework.

Future research

As future work, more case studies to analyze weaknesses and study Smart Contracts can be done. More case studies can cover more situations and edge cases, making future designs of the Smart Contract editor more suitable with additional specialized features and properties.

Furthermore, usability study of the editor can be done to analyze the editors UI and figure out, who should be the target audience and what should

[1] `https://github.com/drozdik-m/das-contract-editor`

change to make the editor as approachable as possible.

Another step in the DasContracts' ecosystem should be analyzing, designing and implementing a DasContract "store", which could store and offer various contract from the editor. The store could be able to generate corresponding Blockchain code and deploy contracts.

Bibliography

1. SKOTNICA, Marek; PERGL, Robert. Das Contract - A Visual Domain Specific Language for Modeling Blockchain Smart Contracts. In: AVEIRO, David; GUIZZARDI, Giancarlo; BORBINHA, José (eds.). *Advances in Enterprise Engineering XIII*. Lisbon, Portugal: Springer International Publishing, 2019, pp. 149–166. ISBN 978-3-030-37932-2. Available from DOI: 10.1007/978-3-030-37933-9_10.
2. FIRMO NETWORK. 3 Famous Smart Contract Hacks You Should Know. *Medium* [online]. 2018 [visited on 2020-04-07]. Available from: https://medium.com/firmonetwork/3-famous-smart-contract-hacks-you-should-know-dffa6b934750.
3. MICROSOFT. *Microsoft Docs: .NET Documentation* [online]. Microsoft [visited on 2020-04-07]. Available from: https://docs.microsoft.com/en-us/dotnet/.
4. ROTH, Daniel; LATHAM, Luke. *Microsoft Docs: Introduction to ASP.NET Core Blazor* [online]. Microsoft, 2020 [visited on 2020-04-07]. Available from: https://docs.microsoft.com/en-us/aspnet/core/blazor/.
5. SOURABH. ASP.NET Core updates in .NET Core 3.1. *Microsoft Devblogs* [online]. 2019 [visited on 2020-04-07]. Available from: https://devblogs.microsoft.com/aspnet/asp-net-core-updates-in-net-core-3-1/.
6. YOU, Evan. *Vue.js Documentation: Introduction* [online] [visited on 2020-04-07]. Available from: https://vuejs.org/v2/guide/.
7. GOOGLE. *Angular Documentation: Introduction to the Angular Docs* [online] [visited on 2020-04-07]. Available from: https://angular.io/docs.

8. HAAS, Andreas; ROSSBERG, Andreas; SCHUFF, Derek L.; TITZER, Ben L.; HOLMAN, Michael; GOHMAN, Dan; WAGNER, Luke; ZAKAI, Alon; BASTIEN, JF. Bringing the Web up to Speed with WebAssembly. *SIGPLAN Not.* 2017, vol. 52, no. 6, pp. 185–200. ISSN 0362-1340. Available from DOI: 10.1145/3140587.3062363.

9. W3C. *WebAssembly* [online] [visited on 2020-04-07]. Available from: https://webassembly.org/.

10. NAKAMOTO, Satoshi. Bitcoin: A Peer-to-Peer Electronic Cash System [online]. 2019 [visited on 2020-04-07]. Available from: http://www.bitcoin.org/bitcoin.pdf.

11. KASIREDDY, Preethi. What do we mean by "blockchains are trustless?" *Medium* [online]. 2019 [visited on 2020-04-07]. Available from: https://medium.com/@preethikasireddy/eli5-what-do-we-mean-by-blockchains-are-trustless-aa420635d5f6.

12. SZABO, Nick. Smart Contracts: Building Blocks for Digital Markets [online]. 1996 [visited on 2020-04-07]. Available from: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

13. BYDŽOVSKÝ, Tomáš. *A state Management in Multi-client Single Page Web Applications* [online]. Prague, 2019 [visited on 2020-04-07]. Available from: https://dspace.cvut.cz/handle/10467/83141. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology. Supervised by Ing. Marek SKOTNICA.

14. SKOTNICA, Marek. *Personal conversation: DasContract language*. Praha, 2019.

15. AXWAY SOFTWARE; BIZAGI LTD.; BRUCE SILVER ASSOCIATES; IDS SCHEER; INTERNATIONAL BUSINESS MACHINES; MEGA INTERNATIONAL; MODEL DRIVEN SOLUTIONS; OBJECT MANAGEMENT GROUP; ORACLE; SAP AG; SOFTWARE AG INC.; TIBCO; UNISYS. *Business Process Model and Notation (BPMN)* [online]. Object Management Group, 2013. Version 2.0.2 [visited on 2020-04-07]. Available from: https://www.omg.org/spec/BPMN/2.0.2/PDF.

16. 88SOLUTIONS; BOC PRODUCTS & SERVICES AG; DECISION MANAGEMENT SOLUTIONS; DEPARTMENT OF VETERANS AFFAIRS; FICO; GFSE E.V.; K.U. LEUVEN; ORACLE; PNA GROUP; RED HAT; SAPIENS DECISION NA; SIGNAVIO GMBH; THEMATIX PARTNERS LLC; TRISOTECH. *Decision Model and Notation* [online]. Object Management Group, 2019. Version 1.3 [visited on 2020-04-07]. Available from: https://www.omg.org/spec/DMN/1.3/PDF.

17. SKOTNICA, Marek; DROZDÍK, Martin; KLICPERA, Jan. *CCMiResearch/DasContract* [online]. GitHub [visited on 2020-04-07]. Available from: `https://github.com/CCMiResearch/DasContract`.

18. SMITH, Steve. *Microsoft Docs: Overview of ASP.NET Core MVC* [online]. Microsoft, 2020 [visited on 2020-04-07]. Available from: `https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-3.1`.

19. HIPP, Richard. *What Is SQLite?* [online] [visited on 2020-04-07]. Available from: `https://www.sqlite.org/index.html`.

20. GIRAUDEL, Hugo. *CSS with superpowers* [online] [visited on 2020-04-07]. Available from: `https://sass-lang.com/`.

21. MICROSOFT. *microsoft/TypeScript* [online]. GitHub [visited on 2020-04-07]. Available from: `https://docs.microsoft.com/en-us/aspnet/core/blazor/`.

22. THELARKINN et al. *Webpack Documentation: Concepts* [online] [visited on 2020-04-07]. Available from: `https://webpack.js.org/concepts/`.

23. ZIVOLO, Federico. *Bootstrap Material Design* [online] [visited on 2020-04-07]. Available from: `https://fezvrasta.github.io/bootstrap-material-design/`.

24. CAMUNDA. *Web-based tooling for BPMN, DMN and CMMN* [online]. Camunda [visited on 2020-04-07]. Available from: `https://bpmn.io/`.

25. SKOTNICA, Marek. *DEMOCaseStudies: Blockchain/Mortgage* [online]. GitHub [visited on 2020-04-14]. Available from: `https://github.com/CCMiResearch/DEMOCaseStudies/tree/master/Blockchain/Mortgage`.

26. ROTH, Daniel; AL., et. *Welcome to Blazor* [online]. Youtube, 2020 [visited on 2020-04-07]. Available from: `https://www.youtube.com/watch?v=KlngrOF6RPw`.

Appendix A

Glossary

.NET Microsoft programming developer platform.

ASP.NET Core A platform for developing web applications.

back-end A data access layer of an application.

front-end A presentation layer of an application.

interoperability Product characteristic, which implicates, that it understands and fully utilizes the interface of another product.

spaghetti code Unstructured and difficult-to-maintain source code.

Appendix B

Acronyms

API Application Programming Interface.

BPMN Business Process Model and Notation.

CSS Cascading Style Sheets.

DEMO Design and Engineering Methodology for Organizations.

DMN Decision Model and Notation.

DOM Document Object Model.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

JSON JavaScript Object Notation.

MVC Model-View-Controller.

REST Representational State Transfer.

SASS Syntactically Awesome Style Sheets.

SPA Single Page Application.

SQL Structured Query Language.

UI User Interface.

UML Unified Modeling Language.

WASM Web Assembly.

XML Extensible Markup Language.

Appendix C

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `case-study` ..... the case study source files
  - `application` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format