FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Client-Side Application Development Using Blazor Framework – a Blockchain Smart Contract Designer Case Study
Student:	Bc. Jan Klicpera
Supervisor:	Ing. Marek Skotnica
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

The main goal of this thesis is to explore the current possibilities of developing and deploying multi-platform client-side apps with the core project being developed as a client-side Blazor web application. The chosen approach will be demonstrated in a complex case study – an open-source smart contract designer. The main goal of the case study is to provide a unified visual language, DasContract, to design legal contracts between parties that can be used to generate executable blockchain smart contracts.

Steps to take:
- Review the Blazor technology and technologies allowing client-side app deployment
- Review the DasContract technology
- Design and implement a DasContract designer utilizing client-side Blazor technology
- Evaluate the benefits of the case-study approach compared to other client-side approaches

---

Electronically approved by Ing. Michal Valenta, Ph.D. on 15 November 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Client-Side Application Development Using Blazor Framework – a Blockchain Smart Contract Designer Case Study

Bc. Jan Klicpera

Department of Software Engineering
Supervisor: Ing. Marek Skotnica

May 1, 2022

Acknowledgements

My deep gratitude goes to my supervisor and mentor, Ing. Marek Skotnica, for his incredible guidance and valuable feedback. It has been a pleasure working with you, thank you. I would also like to thank my family for their unconditional love and support. Last but not least, my appreciations go to all my friends.

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 in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 1, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Klicpera, Jan. *Client-Side Application Development Using Blazor Framework – a Blockchain Smart Contract Designer Case Study.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Blockchain chytré kontrakty jsou relativně nová technologie, jenž by mohla způsobit převrat ve vytváření a vedení právních kontraktů. Příkladem jednoho z jejich benefitů je možnost eliminace potřeby ověřených autorit třetích stran. Jejich širšímu zavedení však brání komplexní, silně technický způsob kterým jsou vytvářeny. Tímto nedostatek se zabývá probíhající výzkumný projekt, DasContract, jenž má za cíl zjednodušit vytváření chytrých kontraktů poskytnutím vizuálního doménově specifického jazyka, který lze zkonvertovat do spustitelného kódu. Tato diplomová práce je součástí DasContract výzkumu a zabývá se navržením a implementováním webové aplikace v Blazor WebAssembly frameworku, která umožní uživatelům vizuálně modelovat chytré kontrakty pomocí DasContract jazyka. Práce dále prozkoumává možnosti nasazení Blazor webové aplikace jakožto samostatné multiplatformní aplikace. Vytvořená aplikace je volně dostupná (zdrojové kódy jsou open-source) a je momentálně využívána k navazujicímu výzkumu.

Klíčová slova Blazor, WebAssembly, chytrý kontrakt, blockchain, DasContract, PWA

Abstract

A Blockchain smart contract (SC) is an emerging technology that has the potential to revolutionize the practice of conducting legal contracts. The benefits of SC include, for example, the opportunity to eliminate the need for third-party authorities. However, one of the challenges associated with smart contracts, stalling their mass adoption, is the complex, highly technical method of creating them. DasContract is an ongoing research project that aims to address this challenge by defining a visual domain-specific language (DSL) that can be converted into executable smart contract code. This thesis contributes to the research project by designing and implementing a web application in the Blazor WebAssembly framework, which allows users to visually model smart contracts using the DasContract DSL. The thesis also explores the possibilities of deploying the implemented client-side Blazor web application as a standalone multi-platform application. The editor is fully open-source and is currently being utilized for conducting further SC research.

Keywords Blazor, WebAssembly, smart contract, blockchain, DasContract, PWA

Contents

List of Figures

Introduction

Recent developments in the world of web technologies have introduced new, exciting approaches to developing and deploying web applications. One of these exciting technologies is WebAssembly, an instruction format that allows to execute code directly in a web browser.

Microsoft has recently released client-side Blazor, which compiles C# code into the WebAssembly instruction format. This gives developers the opportunity to create client-side web applications with the help of libraries from the vast .NET ecosystem. Running the web application directly in the browser, on the one hand, increases the hardware demands but, on the other hand, makes the application easier and cheaper to scale, as opposed to the server-side approach.

The client-based approach also provides another opportunity. One of the properties of web applications that made them so widely popular is their accessibility. If a user wants to visit a web application, they only need a web browser and the address of the application. There is no need for explicit installation; everything is conveniently downloaded on the fly and run in the browser. However, there is a tradeoff in terms of the application's capabilities, as it is launched in an isolated sandbox with limited permissions. The performance of a classical web application is also limited by the connection speed. Several case studies have found that providing an option to install web apps as a standalone application increases user engagement [1].

One way to provide the users with the option to choose their preferred type of app would be to develop a separate standalone application in a platform-specific framework. This would, however, significantly increase the complexity of the project. An alternative approach is to utilize a technology that allows to deploy the web application as standalone whilst preserving the web-oriented codebase.

One of the focuses of this thesis is to explore and demonstrate the possibilities of providing both a web and a fully standalone version of an application without the need for separate codebases.

The DasContract Project

Blockchain smart contracts have proven to be a technology that could help to fully digitalize conducting of legal contracts whilst removing the need for trusted intermediaries to mediate the terms of the contract. One of the current caveats of smart contracts is that they are defined using specialized programming languages, making them difficult to create and convey to other people.

DasContract is an ongoing research project focusing on making smart contracts more comprehensible and accessible to a broader audience. It aims to achieve this by providing a custom visual language that is technologically independent and easy to understand even by people with no technical background.

The practical part of the thesis is a contribution to the research project, as it consists of designing and implementing a visual smart contract editor. The purpose of the editor is to provide a user-friendly application for modelling smart contracts using the DasContract language and converting them into a specific smart contract platform code.

Structure of the Thesis

Chapter 1 covers the current state of the DasContract project and related technologies, like blockchain and smart contracts. Chapter 2 introduces the Blazor framework and compares the technologies that can be used to deploy web apps as standalone applications – Electron, Progressive web applications and Blazor Hybrid.

The third chapter contains the analysis, design and implementation of the DasContract Editor case study. Finally, chapter 4 describes which standalone app deployment technology has been chosen and the steps needed to convert the implemented web application.

Chapter 1

Review of DasContract and Related Technologies

This chapter mainly focuses on introducing the reader the the DasContract research project. In section 1 and 2, the technologies that the DasContract projects builds upon, blockchain and smart contracts are briefly presented. Section 3 provides a summary of the goals of the DasContract project and the current state of the research.

1.1 Blockchain

From a technological viewpoint, blockchain is simply a set of data blocks that are publicly distributed over a peer-to-peer network [2]. That means blockchain can be viewed as a decentralized database – it provides a way to store and retrieve data without the need for a central authority. When a new data block is added to the blockchain, it must contain a cryptographical hash of the block that was previously last. The blocks then form a chain, all containing a unique hash of their "parent" block – hence the term blockchain. The internal structure of a data block can be seen in Figure 1.1.

1.1.1 State Synchronization and Immutability

The decentralization introduces a challenge in terms of all participants having to agree on the contents of the blockchain, with minimalized risks of malicious participants manipulating the data contents in their favour. To prevent misuse, most blockchain implementations employ the following set of rules to confirm the validity of a newly added block [3]:

1. It must contain the hash of the previous block.

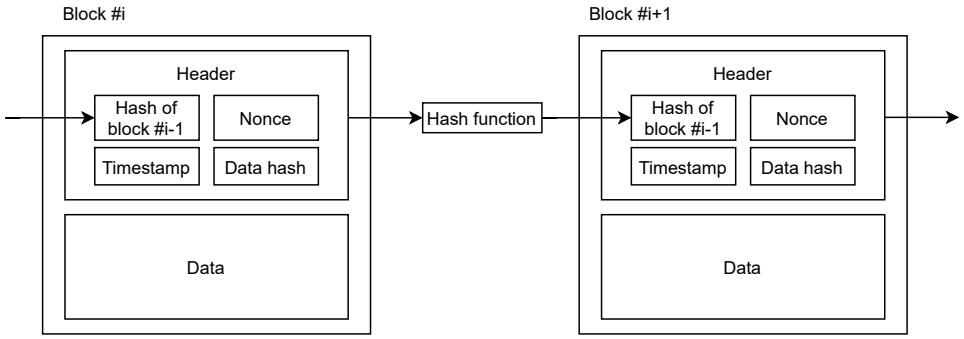

Figure 1.1: Structure of the data blocks in a blockchain.

2. A subset of the block's cryptographic hash must be some determined constant (for example, the hash must start with 20 zeros). This is achieved by finding a special number (called nonce) that produces the hash with the desired subset when appended to the data block. Because the hash function is cryptographic and thus not reversible, the only way to find this number is to manually iterate through the possible combinations, making it a computationally demanding task.

3. Based on the nature of the stored data, there might be additional checks on the data content itself. For example, if the blockchain stores financial transactions, then each transaction must be digitally signed by the involved parties and must be valid in regards to their transactional history (cannot spend more than they own).

The consequence of the first rule is that altering an existing block in the chain is computationally not feasible if enough blocks have already been chained behind it. Changing the contents of the block would not only alter the hash of the block itself, but it would also alter the hash of the next block and all the other blocks down the line, as the hash of a block is partly computed from the hash of its parent block. That means that all of the nonces would have to be recomputed.

The second rule means that adding a new block into the blockchain is not an instant process, as finding the nonce requires a significant amount of computational power. When someone wants to add data to the blockchain, they broadcast this request to all network nodes that wish to participate in computing the nonce. These nodes are called miners. Once the miner has gathered enough valid requests, it compiles them into a block and tries to compute the nonce. When one lucky miner manages to guess the correct nonce, it broadcasts the new block to the rest of the network, requesting them to update their local blockchain.

Since the rules do not define the order in which the incoming data should be compiled into blocks, diverging forks may occur in the chain (different nodes are broadcasting different chains). If that happens, then the longest one is chosen as the "correct" chain. Together with the rules for block validation, this forms a system of voting for the "correct" version of the chain based on computing power. A malicious attacker might be able to produce a different version of the chain, but they would have to keep computing new blocks to maintain the longest chain's position. As long as they do not own a majority of the computing power in the network, this chain will be eventually outperformed by the honest nodes.

1.1.2 Cryptocurrency

To this day, the most well-known usage of blockchain technology is cryptocurrency. Cryptocurrency blockchain implementations store digitally signed transactions between participants. New cryptocurrency is minted with each new block as a reward to the miner that found the correct nonce and thus created the block. Some cryptocurrencies, such as Bitcoin, have implemented a hard cap on the number of coins in circulation; other cryptos, such as Ethereum, limit the amount of cryptocurrency introduced per year.

Bitcoin was notably the first implementation of the blockchain technology, as the author (or authors, the real identity of the author is unknown) proposed a concrete solution to the distributed ledger problem [4]. Since then, the number of cryptocurrencies has skyrocketed, with a plethora of different blockchain implementations. As of writing this thesis, a popular tool for monitoring cryptocurrencies, CoinMarketCap, provides an overview of over 9000 cryptocurrencies.

1.2 Smart Contracts

The term "smart contracts" was first coined by Nick Szabo in 1994 [5], long before the blockchain technology was conceptualized. Szabo proposed that cryptography and digital protocols could be used to automate the interaction between parties, based on a predefined agreement, in a legally binding way.

The concept of smart contracts is not tied to any particular technology. Still, blockchain has proven to be an excellent platform for implementing smart contracts, as it allows smart contracts to utilize the properties of blockchain mentioned in the previous section – blockchain smart contracts are transparent, decentralized and very resistant to tampering. This means that blockchain smart contracts can safely and automatically facilitate defined agreements between multiple parties without the need for a trusted intermediary [6].

Currently, one of the most popular platforms for deploying smart contracts is the Ethereum blockchain project [7]. Ethereum has been defined as a distributed state machine, which stores its machine code and state in

the blockchain. The machine code is in specialized bytecode that is executed inside of the Ethereum virtual machine (EVM) [8].

The Ethereum smart contracts can be written using specialized programming languages that compile into the EVM bytecode [9]. One of the most commonly used languages for Ethereum development is Solidity, an object-oriented language mostly inspired by C++ syntax. Another popular language is Vyper, which describes itself as a contract-oriented pythonic language. It deliberately provides fewer features than Solidity to make contracts easier to audit and more secure [10].

1.3 DasContract

Blockchain smart contracts have turned out to be a promising technology that could cause a major breakthrough in conducting legal contracts. Most importantly, utilizing blockchain smart contracts would not require trust in individuals, organizations or the government, as smart contracts are autonomous and immutable. This would eliminate the need for a trusted middleman to mediate the terms of the contract, which would lower the cost and complexity of the process and would also prevent potential abuse [11].

Until smart contracts can be adopted on a mass scale, several caveats will need to be addressed. The DasContract project addresses one of the caveats, the difficulty of creating smart contracts, and proposes a solution. The current smart contract platforms, such as Ethereum [7], only provide specialized programming languages for creating smart contracts. This makes smart contracts difficult to create and, more importantly, challenging to read and understand [11].

DasContract is an ongoing research project with the primary objective of providing a high-level platform for defining, deploying and managing blockchain smart contracts [12, 13]. The proposed approach of the DasContract project to deliver such a platform can be seen in Figure 1.2. It consists of three parts [13]:

Human Understanding This is where the contract is formally specified in a way that can is legally binding and understandable by all parties without the requirement of extensive technical or legal knowledge. The abstract DasContract language that allows defining contracts in such a way is further described in subsection 1.3.1. An editor application, mentioned in subsection 1.3.4, is also needed to model contracts using the DasContract language.

Technical Implementation Once the contract is conceptually defined, it can be automatically converted into executable smart contract code and deployed onto the blockchain. The converters are mentioned in subsection 1.3.2.

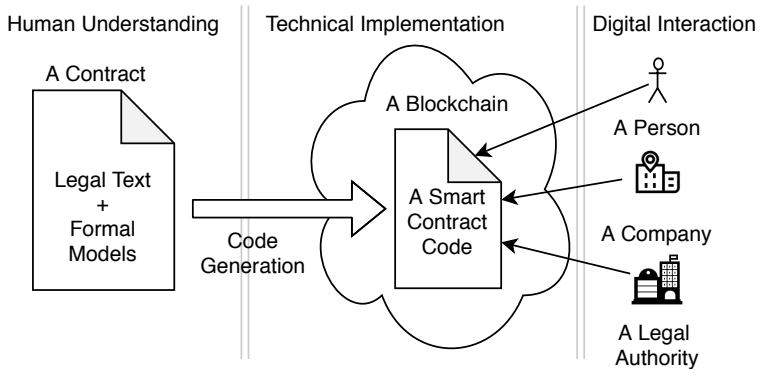

Figure 1.2: Concept architecture of DasContract. [13]

Digital interaction Once the contract is deployed, the affected parties are provided with an interface to interact with the contract. The approach to providing such an interface is described in subsection 1.3.3.

1.3.1 DasContract DSL

The building stone of the project is the DasContract visual domain-specific language (DSL). It allows defining smart contracts in a way that is easier to comprehend even by non-technical users, as opposed to defining smart contracts using code. It also aims to separate the contract model from a specific SC platform. An overview of the metamodel, represented as a UML class diagram, can be seen in Figure 1.3. The metamodel can be split into three parts, which are further described in the following subsections.

Process model

The process model visually describes the flow of rules and user activities in the contract. It uses an extended subset of the BPMN [14] 2.0 level 3 notation [13].

A process model consists of elements that are connected using directional sequence flows. When a process element is successfully completed or evaluated, the execution continues in the direction of the outgoing sequence flows. All process elements and sequence flows must be placed into a process. Multiple processes may be defined in a single contract; interaction between them can be done using call activities.

The process elements can be classified into three categories: events, gateways and tasks. Gateways control the process flow by evaluating a condition or by starting/synchronizing parallel flows. Start events are used to initialize new flows, whilst end events are used to terminate them. A timer boundary event can be attached to a task to redirect the flow if the task is not completed in a defined time frame.

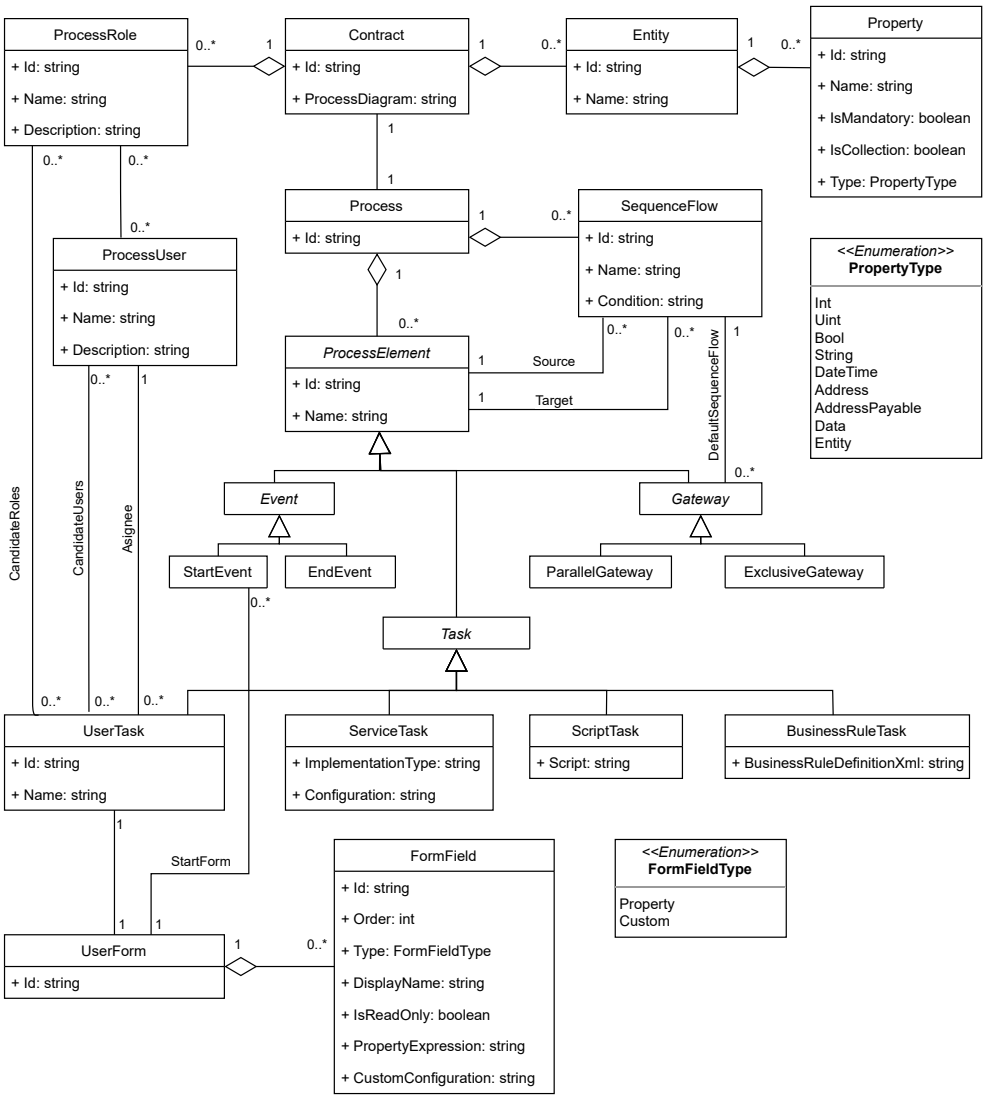

Figure 1.3: The DasContract DSL metamodel. [12]

Script tasks are used to automatically execute logic when reached by the flow. Business rule tasks allow defining complex business logic using the Decision Model and Notation (DMN). User tasks allow to define the point of interaction with the contract by the user. The interaction is described in more detail in the users and forms model subsection.

Data model

The data model defines data structures that can be referenced in the process and forms models. The data model can contain three kinds of data types [12]:

Entity Serves the purpose of a regular data structure that can contain any number of properties. The supported property types are: int, unsigned int, boolean, string, date time, address, reference. The reference property can contain a reference to any other entity, enum, or token. All properties can also be defined as an array, or dictionary.

Enum Is used to define an enumeration of string values.

Token Is used to represent blockchain tokens. Similarly to the entity data type, the token data type can also contain properties. Special token attributes, the symbol and fungibility, can also be specified.

Forms model

User tasks represent a point where an interaction by a contract party is required. The party might need to confirm an action, but they might also need to provide additional information. The forms model allows formalising the form presented to the party when interaction with a given user task is necessary.

The model serves two purposes:

- It is used to generate an off-chain graphical user interface (GUI) for easy user interaction with the deployed contract.
- It is also used to define the internal parameters of the on-chain procedures and bind the input parameters to the data model, making the input data persistable.

1.3.2 Smart Contract Converters

A practical area of the project is the implementation of algorithms for converting the DasContract DSL into code deployable on a concrete smart contract platform. Two converters are currently in development for Solidity[15] and Plutus[16] blockchain platforms.

1.3.3 Forms Wallet and Editor

An approach to designing the user forms model has been demonstrated in [17]. The work has proposed a text-based approach to creating the forms model, powered by a custom domain-specific language (DSL). Automatic conversion of the model into an off-chain graphical user interface has also been implemented.

Lastly, a proof-of-concept DasContract wallet for Ethereum, which connects the generated user form to the deployed SC, has been introduced.

1.3.4 DasContract Editor

An integral part of the DasContract project is an application for creating and editing DasContract models in a user-friendly manner. It should also provide a built-in interface to allow conversion of the model into a target blockchain smart contract code.

An editor for an older version of the DasContract language was created in [18]. It is built using the server-side Blazor framework with a backend server to handle active connections and store user sessions. Since the publication of the older version of the editor, the Blazor framework has officially released a new hosting model, Blazor WebAssembly, which allows the web applications to entirely run inside of the browser (Blazor hosting models are described in greater detail in subsection 2.1.2).

The DasContract project aims to shift the focus from dependence on a backend server to running the application fully on the client-side. This would not only improve the scalability of the application, but it would also provide an opportunity to make the web app deployable as a standalone offline-capable application.

One of the goals of this thesis is to provide an editor for a newer version of the DasContract language that introduces support for multiple processes, user and forms model, additional process elements and more. The new editor is planned to be implemented using Blazor WebAssembly, eliminating the need for a backend server and allowing the app to function in offline mode. The new editor should also be available both as a web and standalone application. Furthermore, the new editor should provide a simpler, more streamlined user interface when modelling for increased effectiveness and user experience.

Chapter 2

Blazor Framework and Standalone App Development

The first section of this chapter introduces the Blazor framework, which will be used to implement the web version of the DasContract Editor.

As mentioned in the introduction, web applications can be easily accessed using only a web browser. However, this ease of access could also carry risks for the user – untrusted websites might contain malicious code. For this reason, browsers employ a technique known as sandboxing. In order to avoid security risks, untrusted code is run in an environment with restricted access to the operating system [19]. For example, the Chromium browser project runs all website code in a sandboxed process, which only allows free access to CPU cycles and memory. As illustrated in Figure 2.1, access to other processes or resources, such as the filesystem, is not permitted inside of the sandboxed process [20].

The downside of this security feature is the limited capabilities of web apps compared to native applications. Another drawback of web applications is their strong dependence on internet connection. User experience can be negatively impacted if the connection is slow or unreliable. In order to enjoy the "best of both worlds", several technologies have emerged in the past years that provide the means to deploy web apps as standalone applications whilst preserving the web codebase.

One of these technologies is directly tied to the Blazor framework, so it is mentioned in subsection 2.1.2. Another two technologies are explored in sections 2.2 and 2.3. Finally, the technologies are compared in section 2.4.

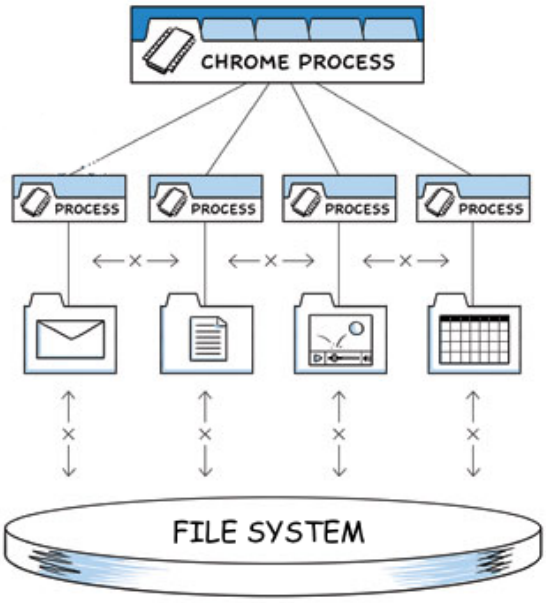

Figure 2.1: The restriction model in chromium-based browsers. The renderer processes are not allowed to directly communicate between each other, or with the file system [21].

2.1 Blazor

Blazor [22] is an open-source framework that allows to develop web applications using the C# programming language. It is a part of the powerful .NET ecosystem, which is a developer platform consisting of tools and libraries for building cross-platform applications [23].

To generate the markup the browser uses to render the page, Blazor utilizes a special syntax, Razor [24], for embedding .NET code into webpages. The syntax is a combination of Razor markup, C# code and HTML. During runtime, the Razor and C# expressions are evaluated and converted into plain HTML, which is then rendered by the browser.

The primary building blocks of Blazor apps are components. Typically written using the Razor markup, they allow to define flexible UI rendering logic and handle user events. They are fully modular, which means that they can be nested, reused in the project, or even across different projects. They can also be shared and distributed as NuGet packages, allowing other developers to reuse them [22].

2.1.1 Dynamically Interacting With the Browser Model

During most interactions with a web application, user actions usually only result in minor changes to the underlying web HTML document. Therefore, it would be rather wasteful to reload the entire HTML document every time a change is made. For this reason, modern web browsers build a representation of the document in memory, called the Document Object Model (DOM). DOM then exposes a programming interface, allowing external scripts to dynamically modify the document [25].

To communicate the changes that need to be done to the DOM, Blazor utilizes an abstraction layer called the render tree. It is a lightweight representation of the browser's DOM, allowing changes to be made to the document more efficiently and flexibly [22, 26]. Several changes may be done in the render tree during a single update cycle. At the end of each update cycle, the smallest set of DOM edits necessary to reflect the changes is calculated [27]. Blazor components are converted into the render tree automatically, but the tree can also be modified programmatically if necessary.

2.1.2 Hosting Models

Blazor provides three different hosting models that determine how the content is built and delivered to the target platform and how the user events are handled. The Razor syntax is identical across the hosting models, which means that components can be easily shared and reused among the different hosting models.

Blazor Server

In the Blazor server hosting solution, also known as server-side Blazor, the app logic is run on a server in an ASP.NET core app. The server builds a render tree for each connection and periodically synchronizes it with the DOM in the browser, as shown in Figure 2.2.

The communication between the server and browser is handled over a connection in the SignalR standard, which is established during the initial request to the website. All UI updates, events and javascript calls in the browser are then handled over the established connection. The connections are resilient to temporary network interruptions, as the server stores disconnected connections for a configurable interval[28].

Since most of the computations are done on the server, technical requirements on the end user's device are very low. On the other hand, scaling the app is more costly and complex, as the server must actively handle each open connection.

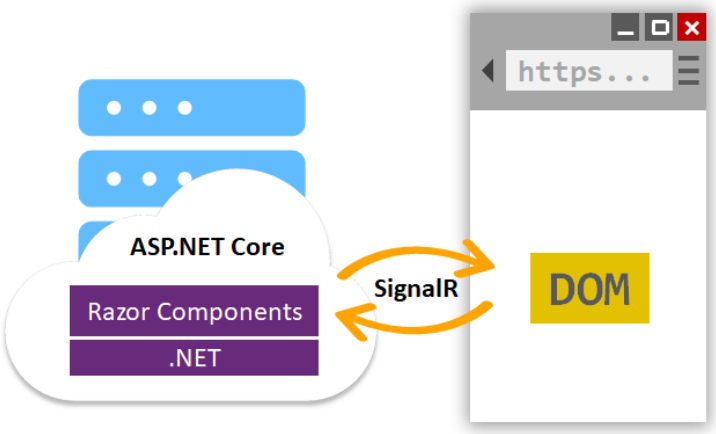

Figure 2.2: Blazor server hosting model schema [22].

Blazor WebAssembly

Blazor WebAssembly, illustrated in Figure 2.3, runs its code and interacts with the DOM directly in the browser. It does so by compiling the source code into WebAssembly (WASM), which is a low-level bytecode format capable of running in modern browsers at near-native speed [29, 30]. It is also worth noting that C# is not the only language that can be compiled into WASM; other languages such as C++, TypeScript, or Rust are also fully supported, with many still being in active development [31].

When the browser produces the initial request, it gets sent the WASM-based .NET runtime along with the app's assemblies and dependencies. No further communication with the server is required beyond that point, which means that the app remains functional even if the internet connection is lost (unless an external data source needs to be accessed, of course). If needed, the app can interact with endpoints over the network, using a variety of protocols, such as web API or SignalR [28].

The client-side Blazor app can be categorized as a static web application – the web content files are delivered directly to the browser without any custom server-side alterations [32]. The app can thus be deployed in a serverless environment, as a dedicated ASP.NET Core web server is not strictly required.

Since the computational workload is shifted to the end user's device, the requirements for capable hardware and software are higher than for Blazor Server apps. The initial download size is also much higher, as the .NET runtime and assemblies of the app must first be loaded [27]. On the other hand, the performance no longer suffers from network latency once the app fully loads.

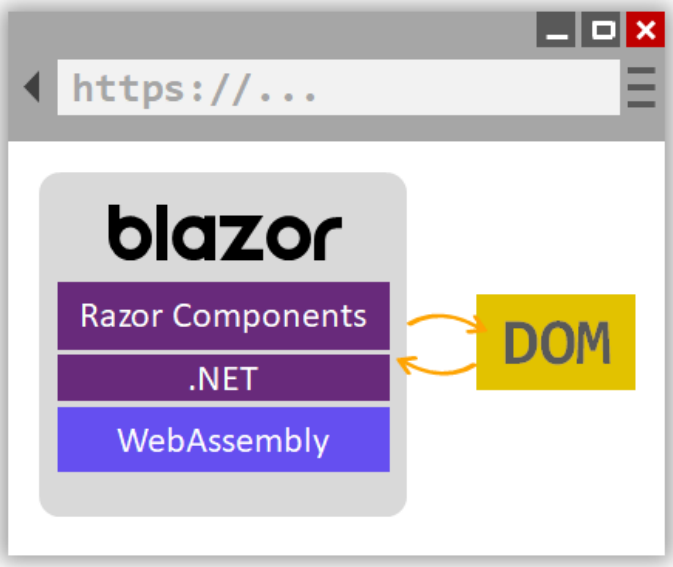

Figure 2.3: Blazor webassembly hosting model schema [22].

Blazor Hybrid

Blazor provides a hosting model, Blazor Hybrid, that allows direct deployment as a native client-side application. As shown in Figure 2.4, by using an embedded web view, Blazor Hybrid makes it possible to render Razor components directly in a native app on both mobile and desktop platforms. This means that components that work with server-side and client-side Blazor also work with this hosting model. Since these apps are not run in the browser sandbox, they can leverage the full capabilities of the native platform, such as access to the file system. It also means that WebAssembly is not needed, as the Razor components can be run directly in the native app. [28, 33]

The disadvantage, when compared to the other two hosting models, is that separate apps must be built, deployed and managed for each platform. It is also worth noting that as of writing this thesis, Blazor Hybrid is in preview and is not recommended for use in production environments. [28]

2.1.3 JavaScript Interoperability

While most common use cases can be handled entirely in the Blazor's .NET ecosystem without needing to write a single line of JavaScript, specific ways to interact with the browser are still only possible using JavaScript, such as accessing the browser's local storage. Another reason for needing to be able to execute JavaScript code is the vast number of js libraries that can be leveraged to speed up development.

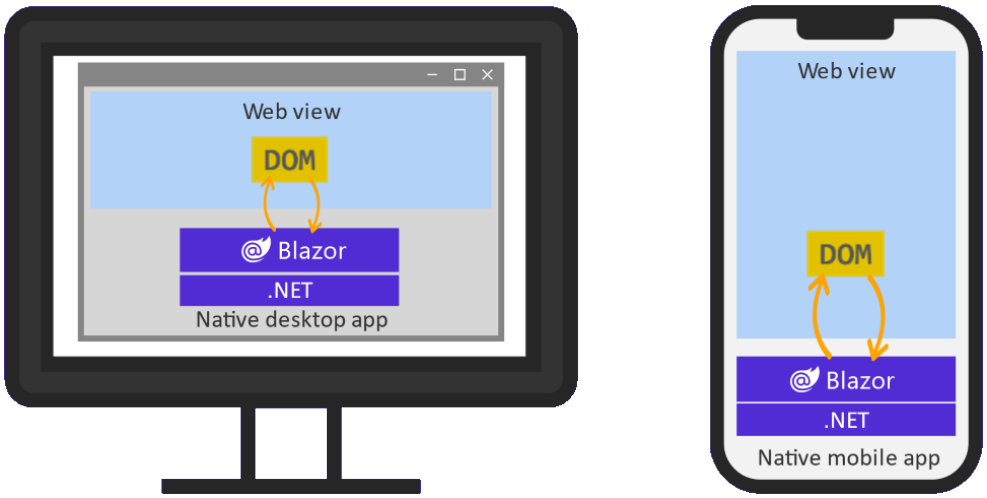

Figure 2.4: Blazor hybrid hosting model schema [28].

Luckily, Blazor supports JavaScript interoperability – JavaScript can be called from .NET and vice versa. It is also supported for both client-side and server-side Blazor. This allows the developer to manipulate the DOM directly and to integrate JavaScript libraries into their codebase.

2.2 Electron

Electron is an open-source framework that makes it possible to deploy web-based applications as standalone desktop applications. It achieves this by embedding Chromium and Node.js into its binary format [34].

Chromium is an ongoing open-source browser project. It serves as a codebase for many popular modern browsers, such as Google Chrome, Opera, or Microsoft Edge. The goal of the project is to provide a "*safer, faster and more stable way for all Internet users to experience the web* " [35].

Node.js is an open-source JavaScript runtime environment, primarily designed to build scalable asynchronous network applications [36]. This allows the developers to build both the frontend and backend of an application using JavaScript. As opposed to running JavaScript in the browser sandbox, Node.js provides extended functionality, such as access to the filesystem.

2.2.1 Process Model

Electron apps run in a multi-process model, which is inspired by the architecture of Chromium. As shown in Figure 2.5, the architecture consists of two types of processes [37]:

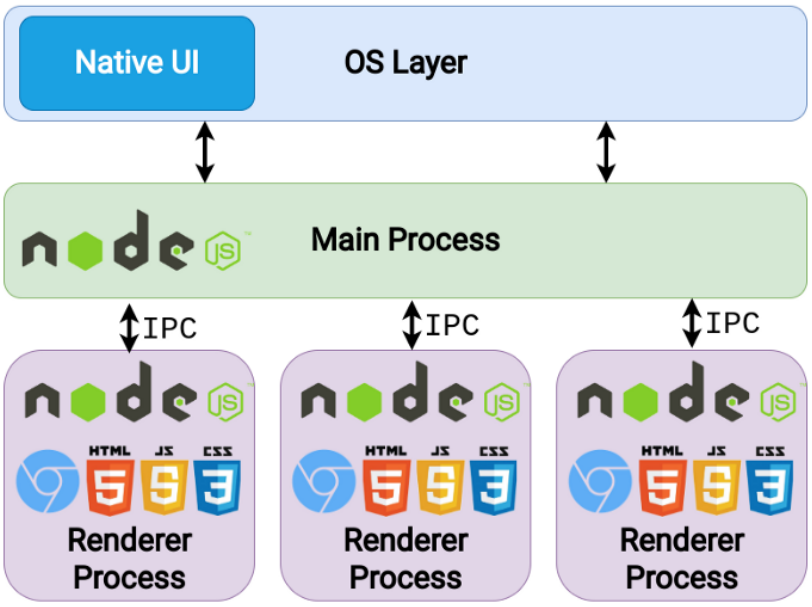

Figure 2.5: A top-level view of the process hiearchy in Electron [38].

Main process The primary purpose of this process is to create and manage application windows. It is launched in the Node.js environment, which means it has access to the extended system APIs. It can directly interact with the render processes and provide data to them.

Render process Each open app window gets assigned a render process, which is responsible for rendering the content. Unlike the main process, render processes do not have direct access to the Node.js APIs.

2.2.2 Context Isolation and Sandboxing

Each render process can attach a preload script, which is executed before the page is rendered. The preload scripts are granted access to the Node.js APIs. For security reasons, the scripts with elevated rights run in a separate context than the websites loaded inside the render processes. This prevents the potentially unsecured website code from having direct access to the powerful Node.js APIs. A context bridge can be used to selectively expose parts of the API between contexts [39].

Electron also uses inter-process communication (IPC) to allow messaging between the main and renderer processes. The communication is done over bidirectional messaging channels, with the the access to the messaging API being allowed only to the preload scripts inside of the renderer process. The

following communication patterns can be used to exchange messages between the processes [40]:

Renderer to main (one-way) The renderer sends a message to the main process without expecting a response.

Renderer to main (two-way) The renderer sends a message to the main process and awaits a response from the main process.

Main to renderer The main process sends a message to a specified renderer.

Renderer to renderer Direct communication between renderers is not possible by default. In order to achieve this, the main process could be set up to act as a message broker, forwarding messages between the renderers.

2.2.3 Distribution and Updating

In order to distribute Electron apps, they must first be packaged for the target platform. This can be done manually with prebuilt binaries or using official automatic build tools. Electron bundles both the Chromium and Node.js libraries into the published application. This, on the one hand, means that the end-users do not need to install additional software to run Electron apps, but it also, on the other hand, negatively impacts the size and performance of the application [41].

In terms of providing updated app versions to the users, Electron provides several methods. The Squirrel framework, in combination with the auto-updater module, is the officially recommended way to distribute these updates.

2.3 Progressive Web Apps

Unlike Electron apps, Progressive Web Apps (PWAs) are not built using a standardized framework. PWAs are instead regular web applications that adhere to a set of techniques and recommendations in order to utilize available web technologies, making web apps more capable and reliable [42].

Web apps are easily discoverable by users and can be quickly shared among users. Native apps, on the contrary, are richer in terms of features that can be provided to the user. They are also able to function regardless of the internet connection. The main goal of PWAs, as illustrated in Figure 2.6, is to combine the benefits of each app type.

There is no official list of properties that a web app must satisfy in order to be deemed as a PWA. Most commonly, these requirements include [42, 44, 45]:

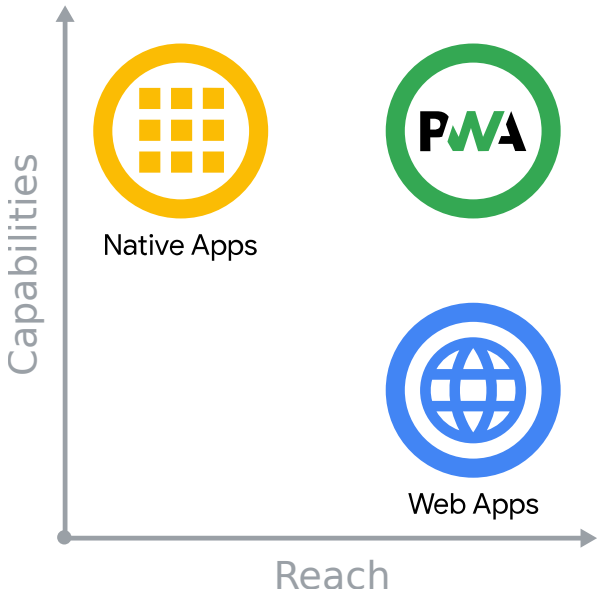

Figure 2.6: Capabilities vs. reach of native apps, web apps and PWAs [43].

Network Independence Once the app has fully loaded, it should remain functional even if the network becomes unreliable. The user should also be able to access the content they have previously visited, even if they are offline. When the user tries to access a page that has not yet been loaded, the app should provide a custom offline page, keeping the user immersed in the PWA experience.

Installability The user should be able to add a shortcut for the app directly to their device. Futhermore, the app should be installable directly from the browser. This shortcut should also open the app in a container that is native to the underlying platform.

Discoverability The app should use the current web standards to describe its content to the search engines, making them easier to discover by users and more likely to be exposed by the search engines.

Responsive design The layout of the UI components should adapt to the user's viewport so that they can use the app comfortably.

Performance The app should load fast and stay fast when browsing through the content.

Platform Independance The app should be able to function regardless of the platform (Windows, Android, ...) and the user should be able to use it comfortably regardless of the input type (mouse and keyboard, touch screen, ...).

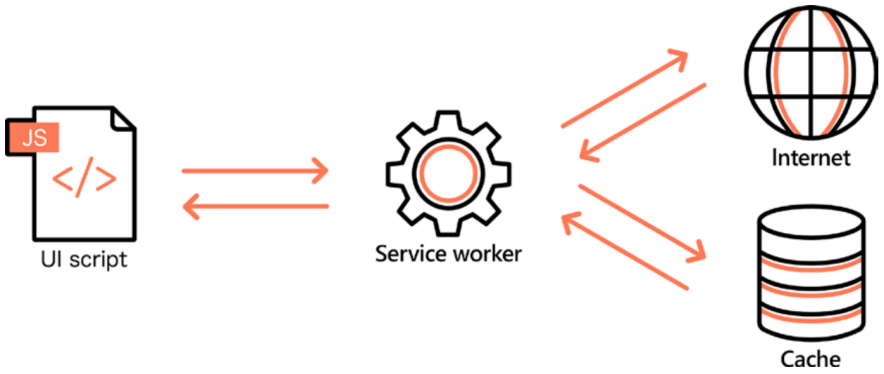

Figure 2.7: Dataflow of requests when using a service worker [42].

2.3.1 Service Workers

Service workers are one of the most significant technologies employed by PWAs, making network-independent web apps a possibility. Service workers are scripts that act as a proxy between the web application and the network, intercepting requests made from the browser [46] (see Figure 2.7). They allow the developer to control how the app should behave in specific network situations. For example, if the network becomes unreachable, instead of displaying an error, the service can attempt to access a snapshot of the resources from the cache and serve it instead.

Scope

All service workers have a defined scope that dictates which requests they will be able to intercept. By default, service workers have control over the scope their script is located in – for example, if a worker has a script at the URL `www.example.com/content/service-worker.js`, it can intercept any request that begins with `www.example.com/content`. It is also important to note that only one service worker can be active per scope [47, 48].

Lifecycle

Service workers run on a different thread than the render process, making them able to run in the background without blocking the UI layer. They also have a lifecycle independent of the page's lifecycle. When the user visits the page for the first time, the service worker will be registered and installed. If no errors occur in the script during installation, the worker goes through the activation phase. These phases emit an event inside the service worker script that allows actions to be performed (caching resources, clearing old cache). The service worker stays activated even if the user fully closes and reopens the

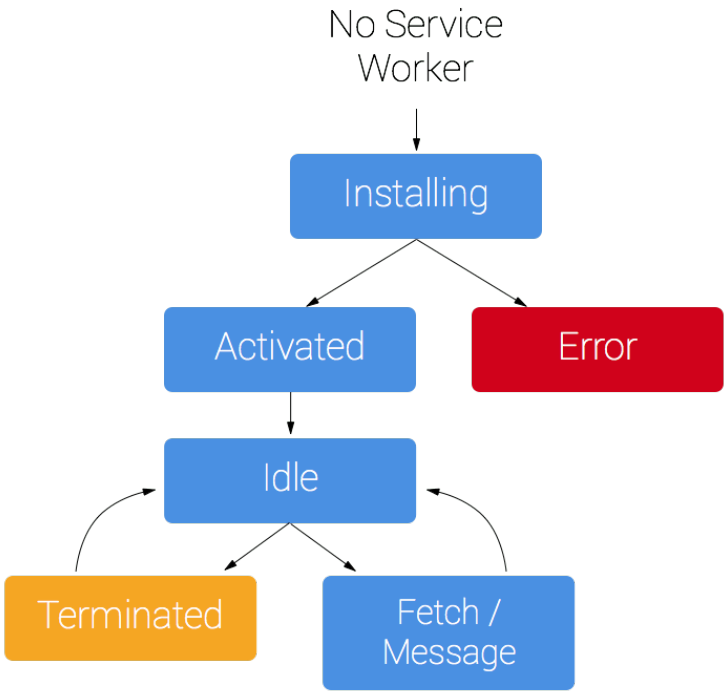

Figure 2.8: Lifecycle of a service worker [47].

webpage, which means that these two phases occur only once in the lifetime of the service worker [47, 46].

When activated, the service is in an idle state by default, waiting for requests to be intercepted. If it stays idle for too long, the browser can temporarily terminate it to save resources. It can be then brought back directly into the idle state (it does not go through the installing and activation phases) [47]. A diagram of the described lifecycle can be seen in Figure 2.8.

Updating a service worker

When a user navigates to a site with an active service worker, the browser checks whether the running service worker script matches the script it downloads from the site. If they do not match, it will install the new worker. The new worker does not get activated right away; it instead goes into a waiting phase while the old service worker remains in control. After the page is closed, the old worker is terminated, and the new worker gets activated [47].

2.3.2 Web App Manifest

Another crucial part of all PWAs is an app manifest, a text file in the JSON format. It describes how the app should appear and behave when installed

onto the device. The following entries are mandatory [49, 50]:

`name` Name of the installed application.

`icons` Icons of different sizes that will be used in the native environment (home screen, app launcher).

`start_url` Defines where the app should start when it is launched.

`display` Allows to customize how the app is displayed in the native environment. The options are `fullscreen`, `standalone` (app opens in a standalone window that contains some control UI elements) and minimal-ui (similar to standalone, but hides non-crucial UI elements).

The manifest may also contain several other optional entries. Most notable ones include [50]:

`short_name` If both name and short name are defined, then the short name is used on the user's home screen, launcher, or other places where space may be limited.

`theme_color` Sets the color of the tool bar on the target platform.

2.3.3 Supported Browsers and Platforms

Unfortunately, full support of all PWA features is inconsistent across the most widely used browsers. Unlike Electron, however, PWA apps are installable even on the mobile platform. Table 2.1 contains an overview of supported features per browser.

It is worth noting that service workers are supported by all mentioned browsers, which means that offline mode is implicitly supported. Installation is supported on all major desktop platforms (Windows, Linux, Mac) and mobile platforms (Android and iOS). Not all browsers on these platforms support installability – Mozilla has abandoned all development of the feature on their desktop Firefox browser [51].

2.4 Comparison of the Technologies for Standalone Deployment

The approaches to deploying the client-side Blazor web app as a standalone application, presented in the previous sections, are compared. Three approaches to client-side app deployment were mentioned: Blazor Hybrid, Electron and PWAs. The properties of each technology are discussed and compared in the following subsections.

Browser	Platform	Supported features
Chrome	Desktop	Service Workers, Installable, File System Access
Firefox	Desktop	Service Workers
Opera	Desktop	Service Workers, File System Access
Edge	Desktop	Service Workers, Installable, File System Access
Safari	Mac	Service Workers, File System Access
Chrome for Android	Android	Service Workers, Installable
Firefox for Android	Android	Service Workers, Installable
Safari on iOS	iOS	Service Workers, Installable, File System Access

Table 2.1: An overview of supported PWA features on the most popular browsers [52].

2.4.1 Supported Platforms

This subsection compares the device platforms that each technology can target.

Electron apps are supported only on the desktop platforms (Windows, Linux and Mac). [34]

Blazor Hybrid is planned to be supported on all the major desktop and mobile platforms; however, Linux is not yet officially supported. [53]

PWAs are available on all platforms that support PWA-capable browsers (all the major desktop and mobile platforms). A more detailed list of the PWA-capable browsers can be seen in subsection 2.3.3. [52, 43]

2.4.2 Maintainability

An important measure is how much additional maintenance the client-side solution would bring if it were deployed alongside the pure web version of the application. Since all of the solutions use a web view or a web browser directly, most of the web application code can be reused for the client-side application.

Electron apps use the Node.Js framework, so regardless of the web technology being wrapped, additional javascript code needs to be written and maintained to control the lifecycle of the application and the access to the native platform features. [34]

Blazor Hybrid apps, similarly to the Electron apps, require an additional codebase to be maintained to make full use of the capabilities of the target platform. [33]

PWAs run directly in a browser instance; no platform-specific code is required. The only files that need to be maintained are the app manifest and service worker code. [54]

2.4.3 Capabilities

This subsection compares the ability of each technology to access the native features of its target platforms.

Electron apps can make full use of the native capabilities of the platform, such as access to the file system or the ability to read sensor data on mobile devices. [34]

Blazor Hybrid apps can also make full use of the native capabilities of the platform. [28]

PWAs are constrained by very similar limitations as web applications when it comes to their access to the native capabilities of the platform. Certain features, such as push notifications, are available. It is also worth noting that new features, such as filesystem access, are being gradually introduced. [43, 54]

2.4.4 Maturity of the Technology

This subsection summarizes whether the technologies are ready for production workloads and whether they are still actively supported.

Electron has first been officially released in 2016 and is still in active development. It is fully open-source, so anyone can contribute by opening an issue or submitting a pull request in the GitHub repository. [55]

Blazor Hybrid has not yet been officially released and is not recommended for production workloads. [33]

PWAs have first been adapted by Google in 2016 as a new development standard. Other companies like Microsoft, Mozilla and Apple have also provided support for PWAs in the following years [54]. Unfortunately, Mozilla has withdrawn its support for PWAs for their popular desktop browser, Firefox, in 2021 [51].

2.4.5 Distribution

This subsection describes whether separate installation packages for each platform must be provided and how the installation packages are delivered to the user.

Electron apps must be bundled into an installable app for each target platform. The installer must then be distributed to the users, for example, by using GitHub Releases feature [56]. Electron apps may also be published to the Windows and Mac app stores. [34]

Blazor Hybrid apps must also be bundled for each target platform and distributed, similarly to the electron apps. [33]

PWAs have the benefit that they can be directly installed as a native app by visiting the web version of the application. This means that the distribution of PWAs is not platform-dependent, as they do not need to be specially bundled and distributed. [54]

2.4.6 Size and Performance

The overhead of each technology in terms of storage requirements is mentioned in this subsection. Overall performance expectations are also briefly mentioned.

Electron apps bundle Chromium and Node.js libraries with each distribution, which means that even simple applications require around 100 MB of disk space. Since the apps run in a browser instance, their performance is also limited as opposed to fully native apps. [34, 57]

Blazor Hybrid apps are still in prerelease, so no conclusive observations can yet be made about the size and performance of the final product. Since they run natively on the device, it is very likely that they will achieve better performance than their browser-based counterparts. [33]

PWAs run in the browser, but unlike Electron, use the existing browser installation instead of bundling it with each app. This means that PWAs can be very lightweight in terms of storage space. Similarly to electron apps, their performance is limited by running in the browser. [54]

Chapter 3

DasContract Editor Case Study

This chapter describes the development process of the DasContract Editor web application. It does not cover the integration of a standalone app deployment approach, as that is independent of the web app implementation and will instead be covered in the next chapter. The chapter starts with an analysis of the case study. Requirements are formalized in section 1, the top-level business process is shown in section 2, and the use cases are described in section 3.

The second part of the chapter, oriented towards the design of the application, begins with section 4. It describes the navigational structure of the application and proposes the design of each page. Section 5 reveals an architectural overview of the application. The following four sections provide a detailed description of the structural design of essential parts of the application, accompanied by class diagrams. The services used to manage the contract data model and lifecycle are introduced in section 6. The design of the process model page is described in section 7. Section 8 mainly covers how the undo/redo functionality was achieved in the user model page. Lastly, section 9 proposes a design for an easy-to-extend integration of the external DasContract converters.

The final part of the chapter begins with section 10, which summarizes the implementation process and mentions notable tools that were used and libraries that were integrated. Last but not least, section 11 describes the methods used for testing the application.

3.1 Functional and Non-Functional Requirements

This section contains the Functional (F) and Non-Functional (NF) requirements and also specifies features that are out of scope (OS) of this thesis project. The requirements are categorized into five sections. The first section defines the general requirements for how the app should behave and how the opening of contracts should be handled. The second, third and fourth sec-

tions define the features needed to create models in the DasContract DSL. The last section contains requirements for converting the DasContract DSL into a smart contract on the target platform.

3.1.1 General Requirements

F1 – Create a new contract
The app allows the user to create a new contract, initialising a new contract context and opening the modeller page.

F2 – Upload and open a contract
The app allows the user to upload an existing contract in the DasContract file format. The modeller page is then opened in a context defined by the file.

F3 – Reopen a recently edited contract
The app allows the user to select a recently edited contract, which opens the modeller page in a context defined by the serialized recent contract.

F4 – Browse and open example contracts
The app displays a library of example contracts to the user. The user can select an example contract, which opens the modeller page in a context defined by the example contract.

F5 – Save and download an open contract
Inside the modeller with an open contract context, the user can download the contract as a file, allowing them to persist the serialized context on their filesystem.

F6 – Rename a contract
The app allows the user to change the name of an open contract.

NF1 – Supported on modern desktop browsers
The app is fully functional on the latest stable versions of Edge, Firefox and Chrome on Windows and Linux desktop platforms.

NF2 – Functional in offline mode
After the initial download of the application, no further internet connection is needed for the app to stay functional.

NF3 – Installable
The app can be installed and launched in a window incorporated into the native environment of the user's platform.

3.1.2 Process Section Requirements

F1 – Add, edit or remove a process
The app allows the user to add, edit or remove processes.

F2 – Add, edit or remove a process element
The app allows the user to add, edit or remove process elements using an interactive visual interface that directly reflects the changes made by the user. The added element must be a part of a process. The properties of process elements are described in subsection 1.3.1.

F3 – Undo/redo an action
The app allows the user to undo or redo actions done in the process editor (removing a process element, for example). The app does not have to persist the history of changes between sessions.

F4 – Download an image representation of the process diagram
The app allows the user to download a diagram representing the process in a bitmap and vector file format.

NF1 – Extensible interface to edit properties of elements
The user interface for editing properties of process elements is designed to be highly extensible. It allows to add new fields and integrate complex editors that might require a significant amount of screen space.

OS1 – Implementation of complex element editors
The developed app is supposed to provide support for integrating complex element editors (for example, the user forms editor); however, implementing all of them is beyond the scope of this thesis project.

3.1.3 Data Section Requirements

F1 – Add, edit or remove a data model element
The user can define the data model by adding, editing or removing a data model element. The properties of data model elements are described in subsection 1.3.1.

F2 – Display a visual representation of the data model
If the data model defined by the user is valid, then the app displays a visual representation of the data model.

F3 – Download a visual representation of the data model
If the data model defined by the user is valid, then the app allows the user to download the visual representation of the data model in a bitmap and vector file format.

F4 – Undo/redo an action
The app allows the user to undo/redo actions in the data model editor.

3.1.4 User Section Requirements

F1 – Add, edit or remove users and roles
The user can add, edit and delete process users/roles using a graphical interface.

F2 – Filter existing users and roles
The app allows the user to filter existing user/role records based on their properties.

F3 – Undo/redo an action
The app allows the user to undo/redo actions done in the user section editor.

3.1.5 Converter Section Requirements

F1 – Choose a target smart contract platform
The app allows the user to choose a target smart contract platform.

F2 – Convert into a smart contract
The app converts and displays the smart contract code based on the chosen target platform.

F3 – Download the converted smart contract
The app allows the user to download the converted smart contract code.

OS1 – Converter logic
The logic required to convert the DasContract model into a smart contract is an external dependency. The app only provides the DasContract model to the converter and displays the result back to the user.

OS2 – Deployment of the contract
The app does not provide an interface to deploy the converted smart contract. The user must download the converted smart contract code and deploy it using the existing tools for the target platform.

3.2 Business Process

The general business process is visualized in an activity diagram in Figure 3.1. Firstly, the user must choose a way to open the contract modeller. Up to four different options are available to the user:

Open a new contract Initializes a new contract.

Open an existing contract Allows the user to upload a DasContract file from their device. The contract is then opened if parsed successfully.

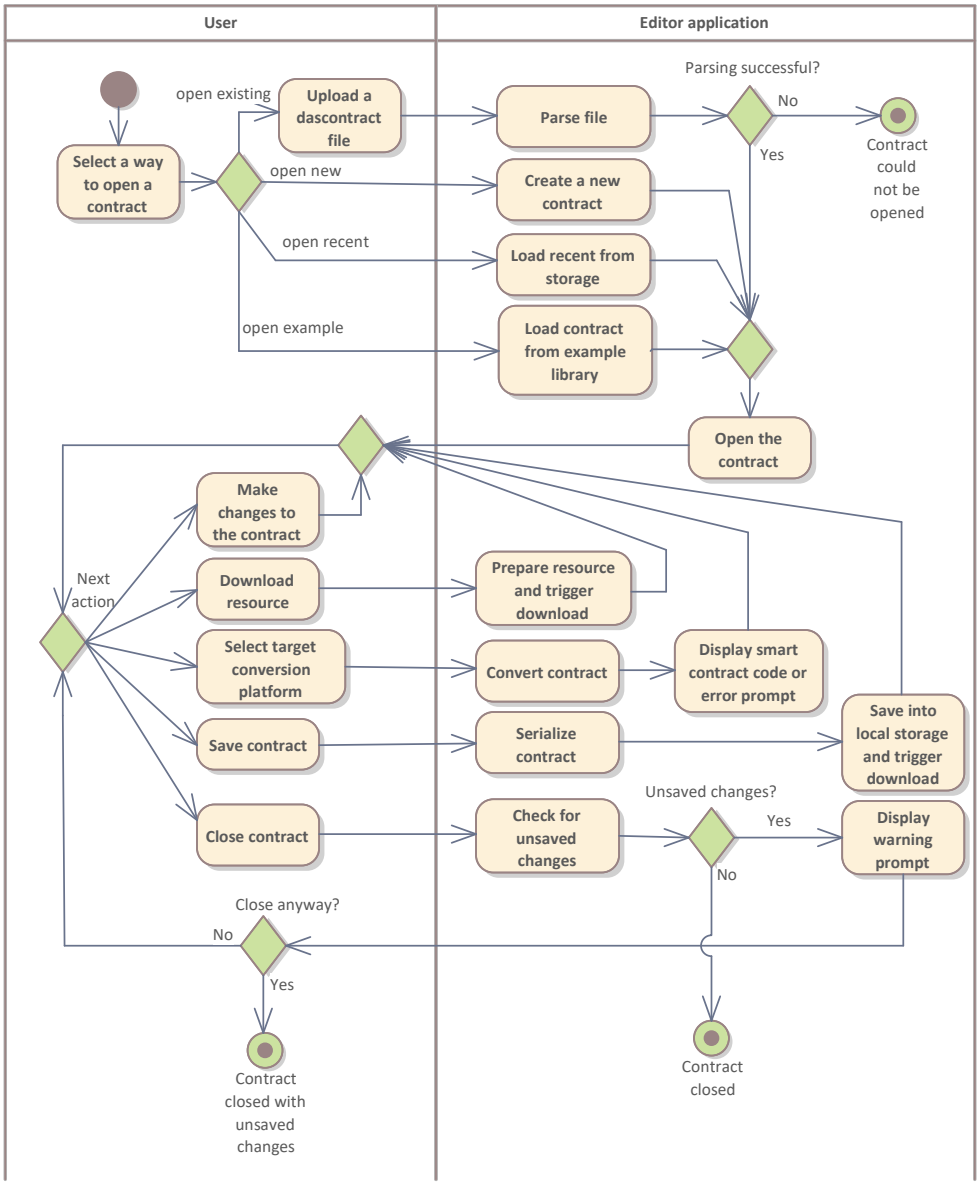

Figure 3.1: Activity diagram describing the business process of the dascontract modeler.

Open a recently edited contract The application shows a list of contracts that have been recently edited by the user and allows to reopen them.

Open an example contract The application offers a library of example contracts that can be opened and freely edited by the user.

Once the contract modeller is open, the user's primary action is making changes to the contract model via the graphical interface. The user can save the contract directly in the application or download it as a DasContract file. The user can also download any model diagrams in both bitmap and vector formats.

If satisfied with the changes, the user can choose from a list of available smart contract platforms to convert their contract. If the conversion is successful, the code is displayed with the option to copy or download it as a file. If the contract cannot be converted, an error prompt describes the issue.

The user can exit the application at any time, but they are alerted if they try to exit with unsaved changes.

3.3 Use Cases

A use case diagram, constructed based on the functional requirements specified in section 3.1, can be seen in Figure 3.2. It contains a single actor – the end-user of the DasContract modeller.

Most of the use cases are self-explanatory, as the topic has already been covered in the previous sections. Use cases 2 to 5 all include use case 1 (the user must always have a contract open). The diagram does not name the concrete steps the user can perform in terms of UC2. These steps include:

- Adding, editing, or removing a process.
- Adding, editing, or removing a process element.
- Adding, editing, or removing a process user or role.
- Adding, editing, or removing a data model element.
- Undoing or redoing an action.

3.4 User Interface Design

After analyzing the previous version of the editor, it has been decided to create the design of the UI from scratch. The application is split into five main pages that the user can navigate to and interact with. The entry point of the application, called the landing page, contains the options to open a contract.

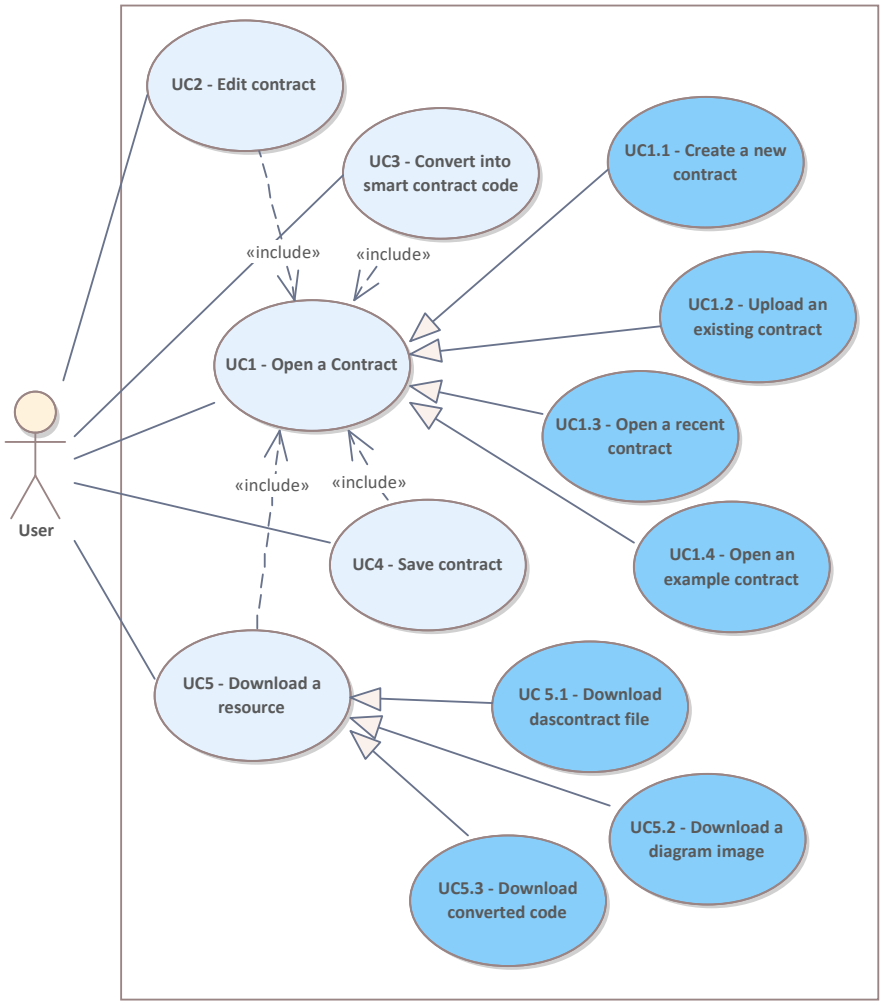

Figure 3.2: Use case diagram for the dascontract modeler.

The goal of the new design is to make the interface as simple as possible, eliminating any unnecessary elements. When editing a contract in the previous version of the editor, three different navigation bars were present simultaneously – a top header for top-level navigation, a breadcrumb bar and a bar for switching between model views. This took up valuable horizontal space and reduced clarity.

Instead, the new design proposes to use a single header to contain all navigations and tool items, such as buttons for saving and downloading. Namely, the header should contain the following elements:

- Logo of DasContract. Clicking on it brings the user back to the landing page.

- Name of the open contract. Clicking on it allows the user to edit it.
- Buttons to navigate between the process, data and user model pages.
- Button to convert a contract. It provides the user with the option to convert the contract into any of the supported blockchain platforms.
- Button to save the open contract. This makes the current state of the contract available and reopenable from the landing page.
- Button to download resources. The options are based on the page the user is on (process diagram on the process page, for example). An option to download the DasContract file is present on all pages.

Another design goal is to eliminate whole-page vertical scrolling (scrolling where the header becomes hidden). Wherever possible, the components should resize automatically to satisfy this criteria.

3.4.1 Landing Page

As specified in the functional requirements, the is presented with four options to open a contract. The landing page, shown in Figure 3.3, contains buttons to create a new contract, open an existing contract and browse example contracts. The button to browse example contracts displays a list of example contracts that the user can open. Lastly, the page contains a list of recently edited contracts, sorted by the last updated date.

3.4.2 Process Model Page

The previous version of the editor required the user to use separate screens for creating process elements and editing their properties. Whenever a new process element was created, the user would have to navigate to a different page and manually look for it in a list of elements to edit its custom properties. This reduced effectiveness and user comfort. To address this, the new editor aims to allow editing of properties without having to switch windows by displaying a resizable sidebar.

The process model page allows the user to define processes and process elements using an interactive visual modeller. As shown in Figure 3.4, the page is split into two primary parts. The left part contains the process model editor, in which the user can add and organize the elements. The visual process model editor will not be further specified, as it will be integrated in the form of an external library.

Clicking any element in the process model editor displays the element's properties in the detail pane on the right side of the page. It contains the custom DasContract model properties that the user can edit. The properties are organized into tabs that can be navigated using buttons on top of the

Figure 3.3: Wireframe of the landing page of the modeler application.

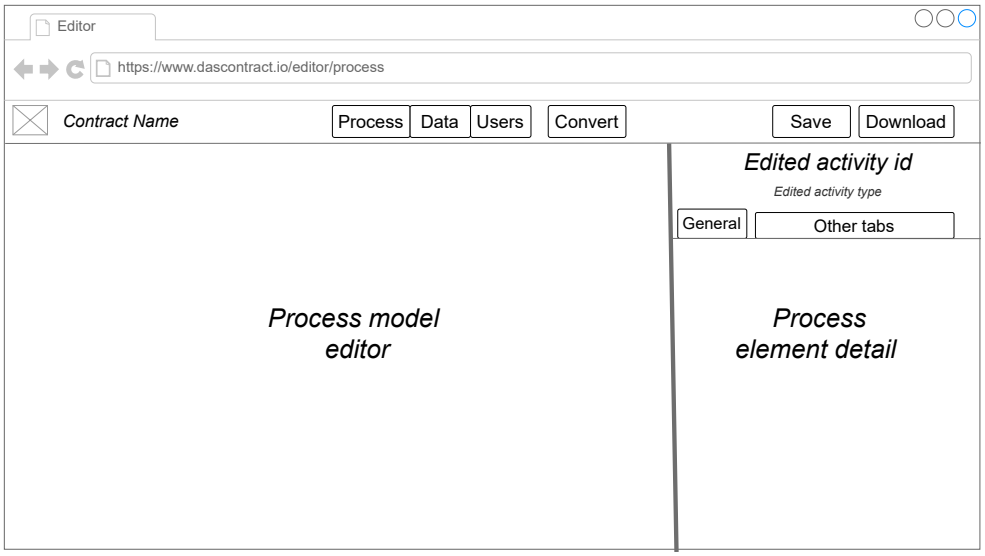

Figure 3.4: Wireframe of the process section of the modeler application.

detail pane. The tabs and contents of the tabs are customized based on the type of the clicked element.

Some tabs might contain complex property editors that require a sizeable amount of screen space, so the detail pane can be horizontally resized by the user. The user can choose the preferred size ratio by dragging the divider between the process model and process element detail. The rest of the section describes the different types of detail tabs.

General Detail Tab

The general tab contains properties of primitive types (boolean, string, single-choice or multi-choice enumeration, ...). The fields are further categorized into sections, which are determined based on the type of the element:

All elements include text fields for editing their id and name.

Processes include a checkbox for specifying whether they are executable.

Multi-instance tasks include a single-choice select box for defining the loop collection. They also include a text field for defining the loop cardinality.

Call activities include a single-choice select box for defining the called process.

User tasks include fields for defining the assignee of the task and the candidate users and roles of the task. The assignee field is a single-choice select box; candidate users and roles are multi-choice select boxes. The section also contains a text field for defining the task's due date.

Timer boundary events include a text box for specifying the time expression for triggering the event.

Script Detail Tab

Some process elements allow attaching custom script logic. The script tab contains a text editor that supports basic features such as key shortcuts, undo/redo, search and replace and syntax highlighting. The elements that include a script tab are:

- Script task
- User task
- Process
- Sequence flow

Business Rules Detail Tab

The business rule task includes this tab to allow the user to define complex business rules using the Decision Model and Notation (DMN). The tab contains an interactive visual modeller similar to the process modeller. The modeller will be integrated using an external library.

User Form Detail Tab

User tasks represent points of interaction by the participants of the contract. The user forms tab allows to define the types of input that the participant (user) must provide to carry out the task. The forms model is defined using a domain-specific language developed by Petr Ančinec [17]. The language serves two purposes – it defines what types of input parameters the underlying blockchain process will have, but it can also be used to automatically generate a user interface to interact with the deployed contract.

The tab itself contains a text editor, in which the user form DSL can be written. It also contains an option to render a preview of the user form interface. When the preview is shown, it replaces the process model editor on the left portion of the screen. That way, the user can make changes to the user model definition whilst having a live preview of how the changes affect the generated user form interface.

3.4.3 Data Model Page

The data model page is used to define the data structure of the contract. The previous version of the editor used a GUI-based approach, in which the user could define the entire data model using visual elements like buttons and text boxes. While this approach is easy for the user to understand initially, the model quickly becomes challenging to navigate as more data elements are added. There was also no way to visually display the data model, making it difficult to analyze and convey to other people.

Defining the Data Model

The new editor utilizes a different approach – instead of defining the data model using GUI elements, it could be defined using a declarative domain-specific language (DSL) in a text editor. While it might initially take the user some time to familiarize themselves with the declarative language, it allows to create models much more effectively. The text editor makes it possible to easily search for keywords, undo/redo actions, and copy/paste elements.

The carrier syntax for the DSL is the Extensible Markup Language (XML). The syntax of XML is human-readable and easy to understand. Thanks to various available tools and libraries, parsing and validating XML is also very straightforward.

An example of a data model definition is shown in listing 1. The root of the XML document is the `DataTypes` element. It can contain any number of the following child elements:

Entity An attribute `IsRootEntity` can be used to specify the root entity flag (if the attribute is not defined, then it is false by default). It can also contain any number of `Property` child elements.

Token Fungibility flag, issued flag and symbol can be specified using the `IsFungible`, `IsIssued` and `Symbol` attributes, respectively. The mint and transfer scripts can be specified using child elements `MintScript` and `TransferScript`. Similarly to the entity element, it can also contain any number of `Property` child elements.

Enum Can contain any number of `Value` elements. These elements are of string type and contain the individual values of the enumeration.

The `Token`, `Entity`, `Enum` and `Property` elements also contain mandatory attributes `Id` and `Name`. The values of the attributes must be unique.

The `Property` element mentioned above defines all of its data properties as attributes. The mandatority flag can be specified as `IsMandatory`. Type of the property can be specified as `PropertyType` using values *single*, *collection*, or *dictionary*. Data type of the property can be specified as `DataType`. If the value of `DataType` is *reference*, then a `ReferencedDataType` attribute must also be specified with the id of the referenced data type.

Visually Representing the Data Model

In order to further improve the clarity of the defined data model, the designer also automatically converts the DSL into a visual representation and displays it alongside the data model definition. The layout of the page can be seen in Figure 3.5, with the text editor being placed on the left and the visual representation being displayed on the right. Similarly to the process section, the vertical divider can also be dragged to adjust the ratio.

The structure of the visual diagram is based on a UML class diagram. It displays each defined data type in a separate box. Each box contains the type of the data type and its name; enums contain a list of their options; Entities and tokens contain a list of their properties, displaying their type and name; tokens also display the values of their symbol, fungibility and issuability attributes.

Furthermore, if any entity or token contains a reference property, a relationship between the entity and referenced data type is displayed. An example diagram based on the data model definition in listing 1 can be seen in Figure 3.6.

The generated diagram can also be downloaded in bitmap and vector formats using the download button in the header.

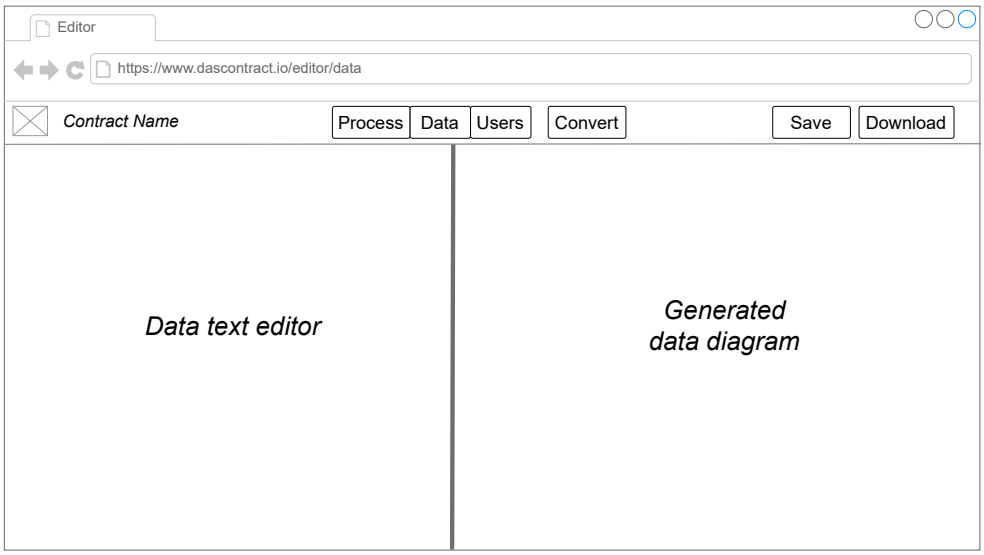

Figure 3.5: Wireframe of the data section of the modeler application.

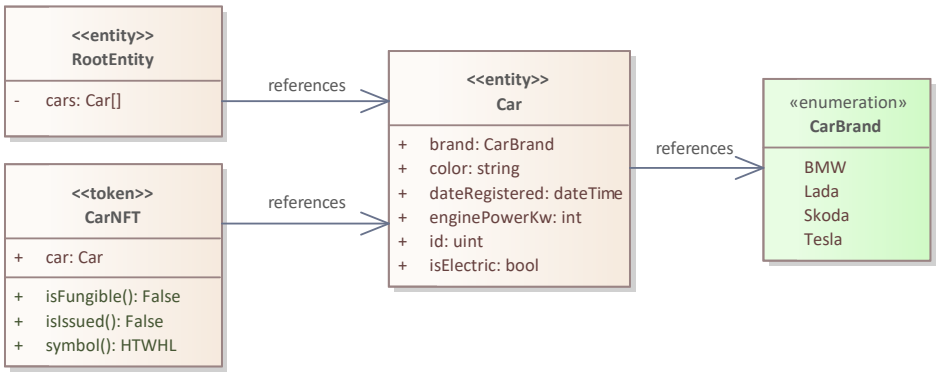

Figure 3.6: Converted visual diagram based on the example data model definition in listing 1.

```
<DataTypes>
  <Token Id="Token_1" Name="CarNFT" IsFungible="false" IsIssued="false" Symbol="HTWHL">
    <MintScript>Insert mint script here</MintScript>
    <TransferScript>Insert transfer script here</TransferScript>
    <Property Id="Property_3" Name="car" IsMandatory="true" PropertyType="Single"
      DataType="Reference" ReferencedDataType="Entity_2" />
  </Token>

  <Enum Id="Enum_1" Name="CarBrand">
    <Value>BMW</Value>
    <Value>Lada</Value>
    <Value>Skoda</Value>
    <Value>Tesla</Value>
  </Enum>

  <Entity Id="Entity_1" Name="RootEntity" IsRootEntity="true">
    <Property Id="Property_1" Name="Cars" IsMandatory="true" PropertyType="Collection"
      DataType="Reference" ReferencedDataType="Entity_2" />
  </Entity>

  <Entity Id="Entity_2" Name="Car">
    <Property Id="Property_2" Name="id" IsMandatory="true"
      PropertyType="Single" DataType="Uint" />
    <Property Id="Property_3" Name="brand" IsMandatory="true"
      PropertyType="Single" DataType="Reference" ReferencedDataType="Enum_1" />
    <Property Id="Property_4" Name="color" IsMandatory="true"
      PropertyType="Single" DataType="String" />
    <Property Id="Property_5" Name="enginePowerKw" IsMandatory="true"
      PropertyType="Single" DataType="Int" />
    <Property Id="Property_6" Name="isElectric" IsMandatory="false"
      PropertyType="Single" DataType="Bool" />
    <Property Id="Property_7" Name="dateRegistered" IsMandatory="true"
      PropertyType="Single" DataType="DateTime" />
  </Entity>
</DataTypes>
```

Listing 1: An example of a valid data model definition.

3.4.4 Users and Roles Model Page

This page is used to specify participants of the contract in the form of contract users and roles. Roles have only two attributes – name and description. Contract users contain a name, description, a blockchain address of the user, and a list of roles assigned to the user.

While it would be possible to apply the same DSL approach as in the data model section for defining the user model, it has been decided against it. Unlike the data model, the user model is simple, with the only relationships between elements being the role assignments. For that reason, a more traditional GUI-based approach has been chosen.

As shown in Figure 3.7, the page is vertically split into two equal panes, the left one containing the users and the right one containing the roles. New roles and users can be added using a button, which creates a new form box containing the definable properties.

Both the panes also contain a search bar, which allows to filter the records

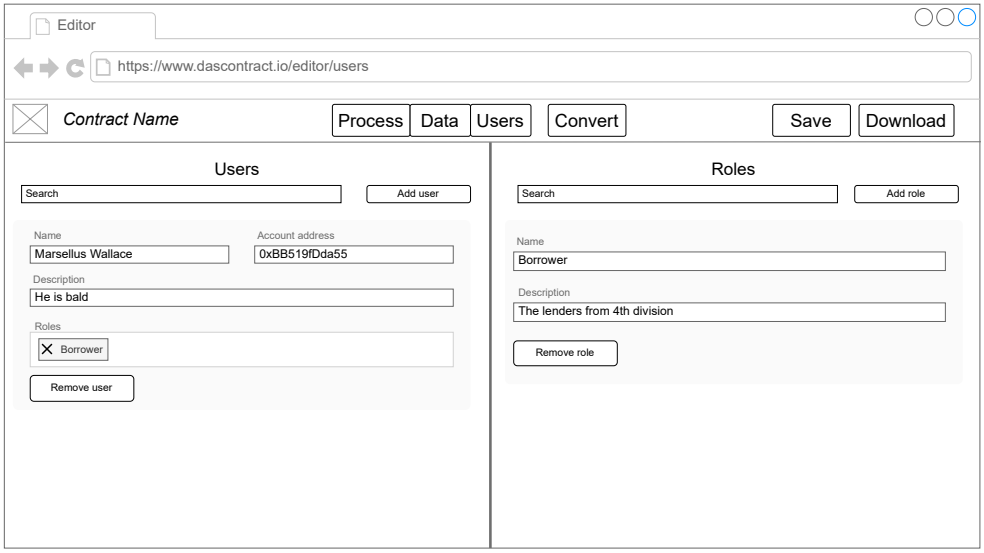

Figure 3.7: Wireframe of the user and roles section of the modeler application.

based on a provided keyword. The search matches a record if any of its attributes contain the keyword.

3.4.5 Converted Contract Page

When the user chooses to convert the contract into a blockchain smart contract, the DasContract model is delegated into a converter for the target platform. An error prompt is displayed on the screen if the conversion fails, describing why the conversion could not be done. If the conversion is successful, the converted smart contract code is displayed to the user in a text editor. The user can then download the converted code using the download button in the header.

3.5 Project Structure Overview

The project is developed using the Blazor WebAssembly framework. The web application is designed to run fully on the client's device, which means that no backend server is required, as the application can be entirely served as static files.

The structure of the solution is shown in Figure 3.8. The main project `DasContract.Editor.Web` contains four key namespaces. The `Pages` namespace contains the razor views responsible for rendering the pages the user can navigate to. It utilizes the `Shared` package, which contains razor views that are shared across pages, such as Layouts and the navigation menu. The

`Components` namespace contains all razor components that can be reused across the application.

The business logic is contained inside of the `Services` namespace. It provides interfaces for working with the contract data models, downloading resources, interacting with javascript libraries, converting the contract and so on. The services can be accessed from the razor views using dependency injection. All of the services have a scoped lifetime, meaning that a single instance of a service object is shared across all its clients.

All namespaces import an external project `DasContract.Abstraction`, which contains the DasContract data model definitions and logic for serialization and deserialization. The `Services` namespace also imports the converters responsible for converting a DasContract model into blockchain smart contract code for a specific platform.

The project also includes several javascript files, which primarily contain methods for interacting with the javascript libraries used in this project. The used libraries are mentioned in further sections.

3.6 Contract Management

The entire DasContract data model is stored in a class called `Contract`, which is defined in the `DasContract.Abstraction` external namespace. The instance of this class should not be directly exposed, as incorrect manipulation of the contract might lead to inconsistencies and code duplications. Instead, the contract can be indirectly accessed and updated using management interfaces defined in the `Services.ContractManagement` namespace. The interfaces can be seen in Figure 3.9.

The `ContractManager` class is responsible for managing the lifecycle of the currently open contract. It implements methods to restore a contract from its serialized form or to initialize a new contract. It also provides the contract instance to the other management classes.

The `IProcessModelManager` provides an interface for manipulating the contract process model. It is primarily used by the `BpmnSynchronizer` described in the next section.

The `IDataModelManager` provides an interface for setting the contract data model and retrieving information about it. Unlike other manager classes, it does not provide methods to add individual data types, as the data model is defined and parsed using the custom domain-specific language.

The `IUserModelManager` provides an interface for modifying and reading the contract user model. It also exposes events that notify about users and roles being added and removed.

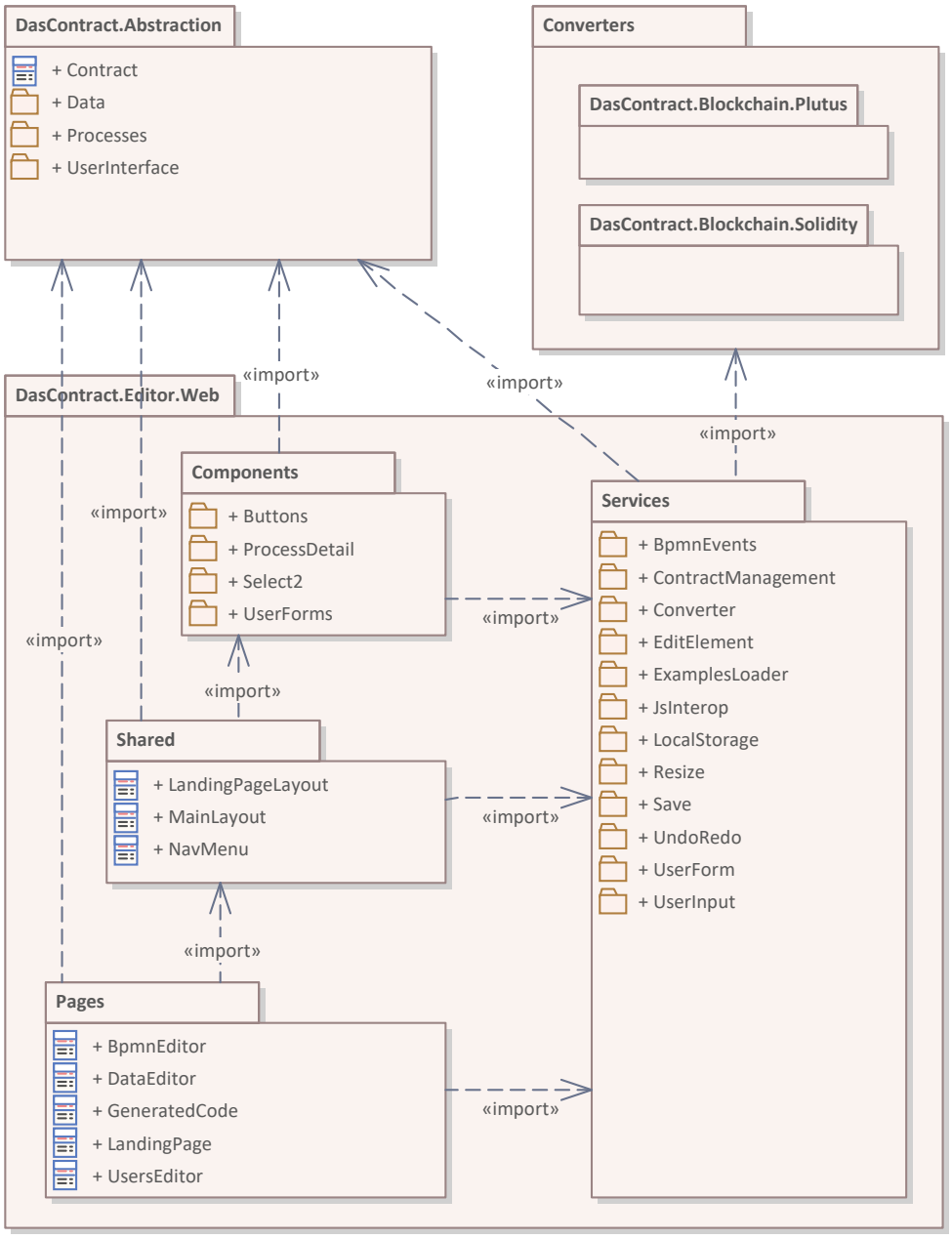

Figure 3.8: Package diagram view of the DasContract Editor.

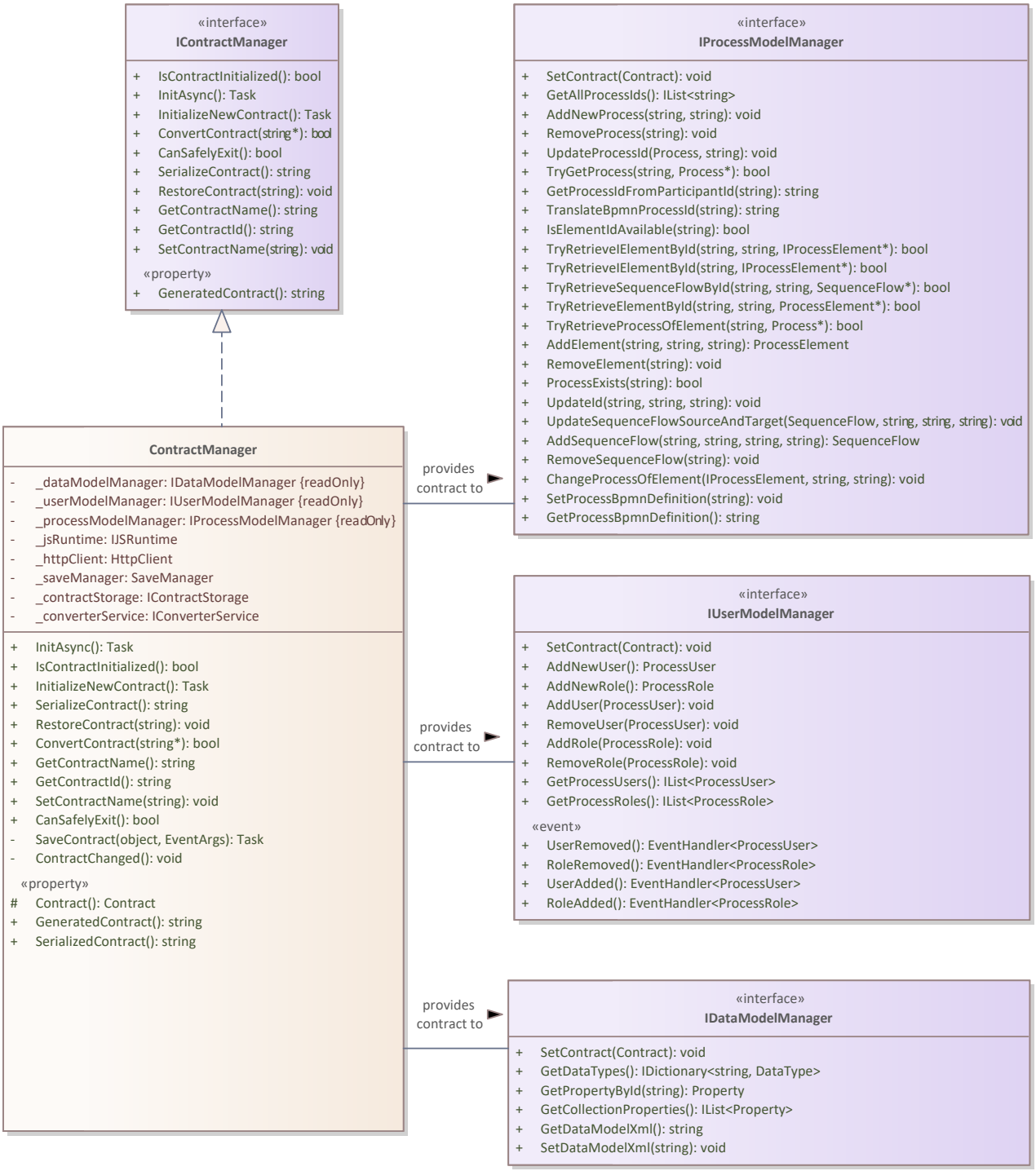

Figure 3.9: The interfaces of classes responsible for managing the lifecycle of the dascontract datamodel.

3.7 Process Modeller

As mentioned in subsection 3.4.2, the process model will utilize an external library to allow visual BPMN process modelling. This section describes the approach to synchronizing process models and handling other incoming events from the external library. It also covers the functionality of the process element sidebar. Lastly, a solution to supporting the undo/redo functionality is presented.

3.7.1 Synchronizing the Process Data Models

Since the DasContract language is an extension of BPMN, the data models of the process elements contain additional properties that are not present in the original bpmn-js data model. Three possible approaches to tackle this problem have been identified:

1. **Extending the javascript process element objects**. Since javascript objects can be dynamically extended with additional properties, extending the process data model directly in javascript would be possible. This would, however, make it difficult to work with the process data model in the .NET parts of the application, as it would require js interop calls for each interaction. It would also shift the codebase away from the Blazor framework into javascript, which is not desired.

2. **Parsing the entire bpmn-js process model definition into DasContract**. A simple solution would be to parse the entire javascript process data model into the .NET process data model. This approach is unfortunately not very practical since the data models need to be synchronized after every change made by the user. This would mean that after every change in the bp mn-js model, the entire data model would need to be parsed.

3. **Listening to bpmn-js events and synchronizing the data model on a per-event basis**. The bpmn-js library provides an event bus, which can be used to notify the .NET application about the process model modifications. This way, the process data model can be selectively synchronized without needing to parse the entire process data model. The downside of this approach is the need for additional synchronization logic. The synchronization logic also needs to be well tested, as a bug could result in the two models becoming desynchronized.

The third option has been chosen as the most suitable for the current scenario, as it preserves the .NET codebase and does not require the entire model to be reparsed after every change.

The classes responsible for receiving and handling events can be seen in Figure 3.10. The `BpmnEventHandler` class acts as the entry-point for the

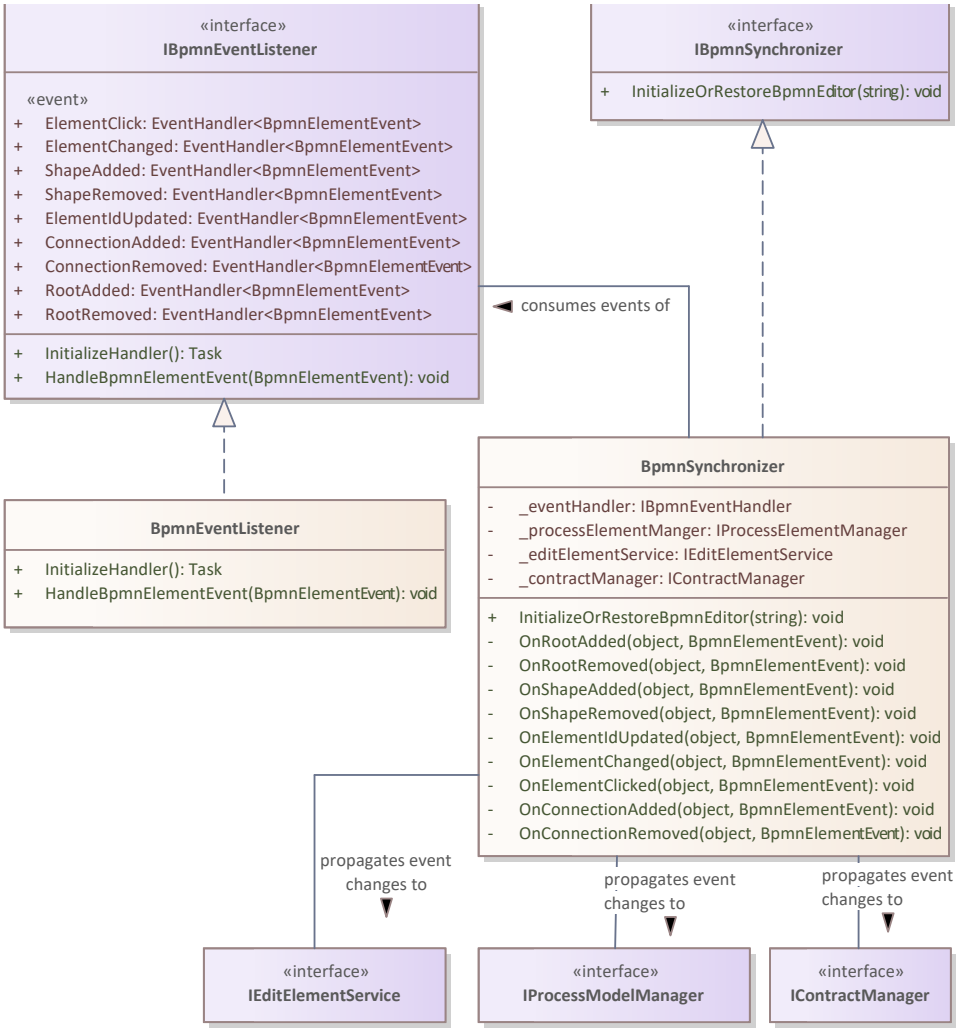

Figure 3.10: Classes responsible for handling incoming bpmn modeler events.

BPMN events, as it defines a method marked with the `JsInvokable`, allowing javascript code to invoke the method. The method handles the javascript event and, based on the type of the received event, executes one of the .NET events it exposes.

The `BpmnSynchronizer` class listens to events invoked by the `BpmnEvent Handler`, each being processed by a private method. These private methods contain business logic for extracting the information about the changed elements and calling the appropriate methods in the `ProcessModelManager`, synchronizing the .net process data model with its javascript counterpart.

An example of an event being handled can be seen in Figure 3.11. The

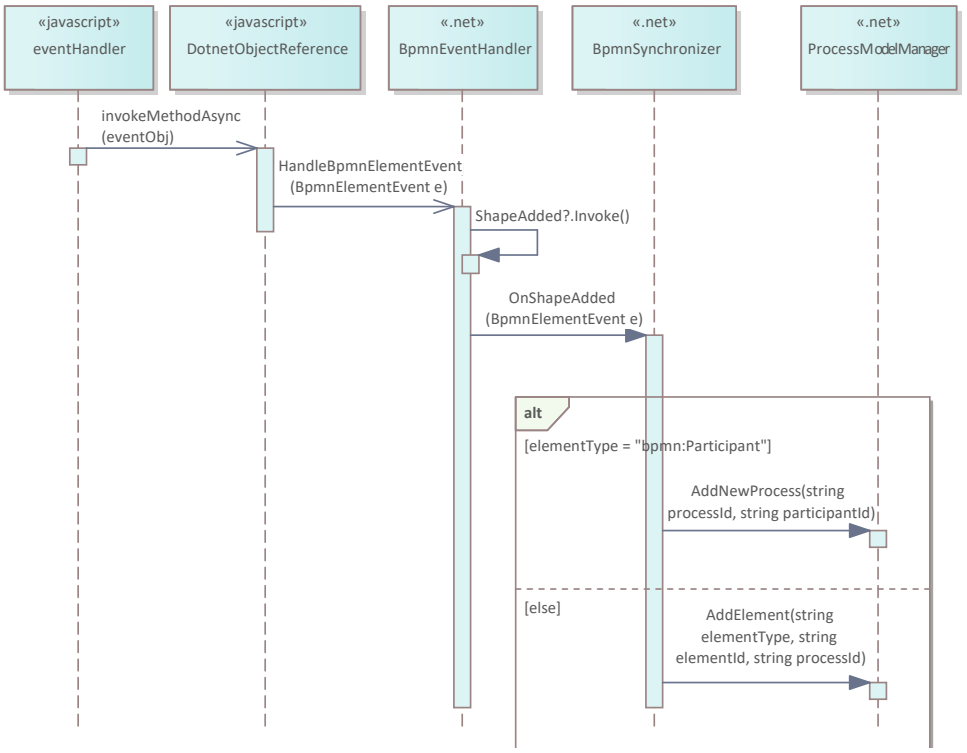

Figure 3.11: Sequence diagram showcasing how a bpmn event is handled when a new shape is added in the visual modeler.

event is first caught in a javascript `eventHandler`, which extracts the relevant information into a javascript object. The object is passed onto the `DotnetObjectReference` object, which in simplified terms acts as a bridge between javascript and .net. The `HandleBpmnElementEvent` method is called, which looks at the type of the event and invokes the appropriate .net event. `BpmnSynchronizer` consumes this event and, based on the type of the element, calls the `ProcessModelManager` to either add a new process or a new process element.

3.7.2 Element Detail Sidebar

As mentioned in subsection 3.4.2, clicking on an element in the process model view opens a sidebar that allows the user to edit the properties of the selected element.

To keep track of the currently selected element, `EditElementService` is used. It contains a public field that allows to get and set the currently edited element and also provides two events that notify about a new element being

assigned and about the element being modified. The razor views consume these events, as the razor component needs to be rerendered whenever such change happens.

To determine which element is currently selected, the javascript event handler listens to an `elementClicked` event. This event is then delegated to .NET and processed using the same structure described in the previous subsection. The `BpmnSynchronizer` tries to find the element object based on the id provided in the event arguments and sets it as the edited element in the `EditElementService`.

General tab

By default, every element detail contains at least one tab – the general tab. In order to avoid code duplication, the general tab is rendered using nested razor components, which are visualized in Figure 3.12. The hierarchy of the razor views mimics the hierarchy of the contract element classes, allowing property inputs such as `ID` and `Name` to be reused. If a new property or element were to be added in the future, the razor component structure could also be easily extended to accommodate this change.

Scripts tabs

Process elements such as script tasks, user tasks, and sequence flows require a script tab for specifying additional logic. The tab will make use of an external text editor library, which is further described in subsection 3.10.1.

Forms tab

The user task also contains a tab for defining user forms. Similarly to the data model, the user forms can be defined using a custom XML-based DML. The actual implementation of the forms tab content has been done in [17]; this work only integrates it into the editor.

Business rules tab

Business rule tasks allow defining custom business rules. As described in subsection 1.3.1, this is achieved by using the Decision Model and Notation (DMN) standard. An external library that allows visually modelling in the notation will be integrated.

3.7.3 Supporting Undo/Redo Operations

The bpmn-js modeller natively supports the undo/redo functionality for all actions performed by the user. When undo/redo request is triggered, the

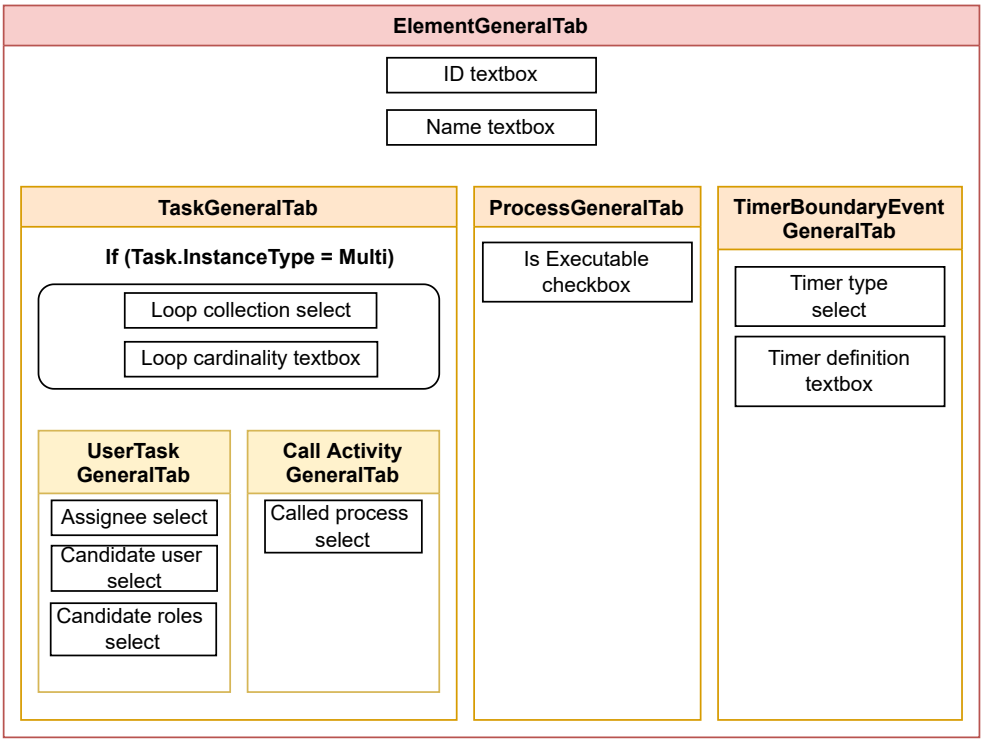

Figure 3.12: A visualization of how the razor component views are nested based on the type of the edited element.

modeller also invokes the events that correspond to the action done during the undo/redo, keeping the javascript and .net process data models in sync.

The only obstacle was that when a delete action was undone, the readded element lost all of its DasContract properties, as those exist only in the .NET process data model. This was solved by storing deleted process elements in memory. When a `element-created` event for a process element is handled, it first checks whether the element's id is not contained inside the list of deleted objects. If the element is found, then it is restored, including all of the DasContract properties.

3.8 User Model

In order to satisfy one of the functional requirements, all of the actions in the user model page must be undoable. One of the most common ways of implementing undo/redo functionality in object-oriented programming is to make use of the command design pattern [58]. Instead of performing actions directly (such as adding a new role), the actions are represented as instances

of classes called commands. A separate class is created for each type of action, extending an abstract command class that defines a method to execute the action. The command class can also define an undo method, which returns the receiver of the action to the state prior to execution.

The benefit of using the class instances to represent actions is that executed actions can be stored in a stack, allowing the commands to be retrieved and undone when requested. To also support redo functionality, the undone commands are simply moved from the undo stack into a redo stack.

The implementation of the command pattern can be seen in Figure 3.13. An abstract command class `UserModelCommand` defines the `Execute` and `Undo` methods, which are implemented by the concrete command classes. It also defines a public property `UserModelManager`, which acts as the receiver of the actions. The commands are created, stored and executed in `UsersRoleFacade`.

3.9 Smart Contract Conversion

New conversion target platforms are expected to be added in the future, so it is vital to minimalize the impact on existing code when adding a new target platform. There is no standard interface that all converter packages must adhere to, which means custom code needs to be written for each converter to handle it. To encapsulate this custom code and to allow converters to be interchangeable in the rest of the application, the strategy design pattern [59, 58] can be employed.

The strategy design pattern says that different behaviours, which need to be assigned dynamically, should adhere to a common interface. Each behaviour should then implement a concrete class containing the logic. By defining a common abstract class, called `ConversionStrategy`, which acts as a base class for the concrete conversion methods, the logic for handling the conversion is encapsulated.

The strategies can be then used interchangeably and dynamically assigned. When a conversion platform needs to be supported, a new concrete strategy class is simply added. The only changes that need to be made to the existing code are in the `NavMenu` class.

3.10 Implementation and Used Technologies

The implementation of the application went as expected without any major issues. As previously mentioned, the app was implemented in the Blazor WebAssembly framework.

To style the generic components of the application, Bootstrap [60], a popular open-source CSS framework, was used. Custom style sheets were written

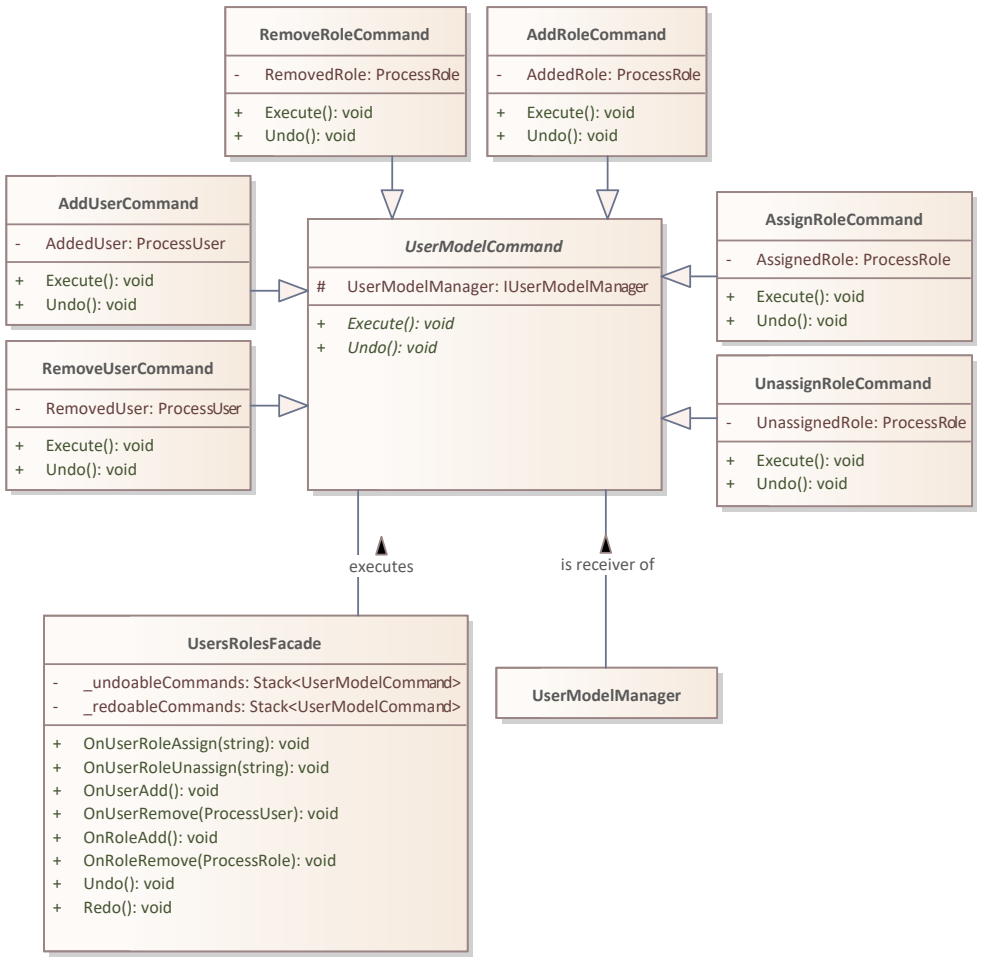

Figure 3.13: Class diagram of the undo/redo functionality for the users and roles editor page.

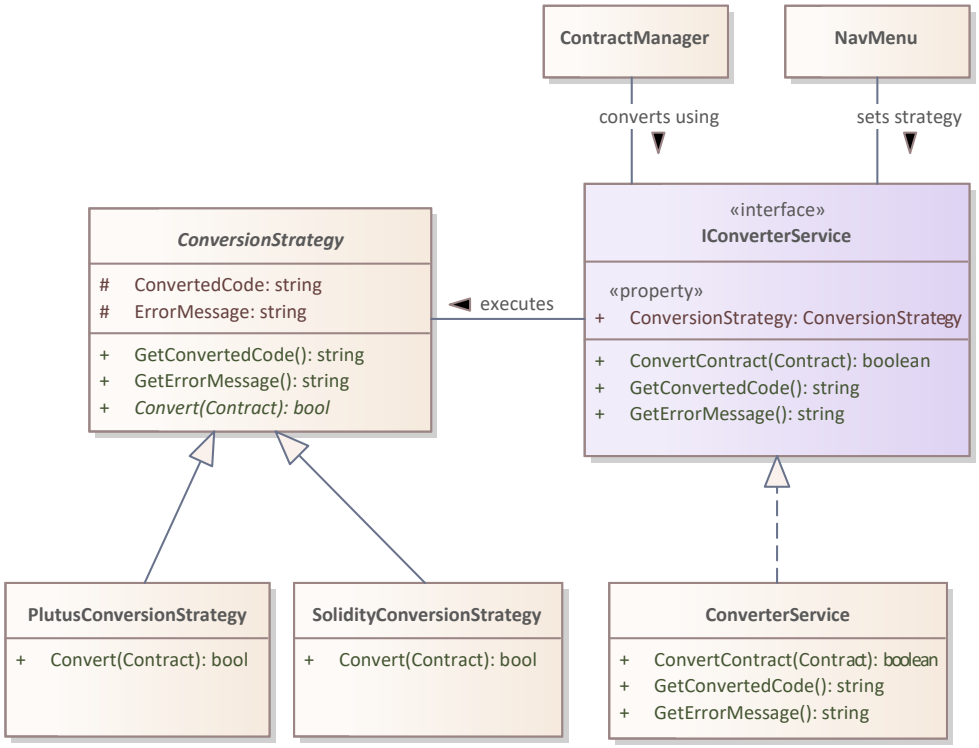

Figure 3.14: Classes related to contract conversion

using Sass [61], which is an extension language for CSS that supports advanced features such as variables, mixins and functions.

Third-party javascript libraries were managed using the npm package manager. The libraries were integrated using bridge javascript files that are called from the .NET codebase. The javascript files were bundled using webpack, which compiled the files including their module dependencies into minified static javascript files.

The rest of the section mentions the third-party libraries that have been used and describes the process of integrating them.

3.10.1 Text Editor Integration

As mentioned in the analysis chapter, a text editor is required in several areas of the application. The Monaco editor [62], which powers the popular VS Code text editor, has been chosen as a suitable library. The library is written in javascript, so javascript interoperability must be used to integrate the library. A project called BlazorMonaco [63] makes the Monaco editor available as a Blazor component, making integration into Blazor applications easier.

3.10.2 Data Model Diagram Generation

One of the requirements for the data model definition section was to render a visual representation of the data model. Mermaid [64], an open-source javascript library, provides the functionality to generate UML diagrams and flowcharts from text definition.

One of the supported diagram types is the UML class diagram. Since Mermaid uses a custom language for defining the diagram structure, the data model definition needs to be converted into this language. The conversion is handled by the `DataModelConverter` service. It accepts a dictionary of the defined datatypes and returns a string that can be used to generate the diagram.

3.10.3 BPMN Modeller

An open-source javascript library, bpmn-js[65], has been chosen to visually define the DasContract process model, which is heavily based on BPMN. The process of synchronization has already been described in subsection 3.7.1.

The modeller is highly configurable, allowing custom model elements and rules to be added. The modeller was customized by removing elements that are not present in the DasContract DSL. Custom modelling rules were also applied. For example, the DasContract DSL currently only permits intermediary events to be directly attached to tasks.

3.10.4 DMN Modeller

A DMN open-source javascript modeller, dmn-js [66], has been integrated into the business rule tab of business rule tasks. DasContract uses the DMN without any modifications, so the synchronization of the models is effortless, as the library allows downloading and importing of models in a serialized XML format. The business rule task class contains a string field for persisting the serialized format.

3.10.5 Advanced Select Component

Some input fields permit multiple entries to be selected (when choosing the assignees of a user task, for example). Since the user can also define the entries, there is no upper limit on the number of options. It is thus important to use a UI component that can accommodate this functionality. The default select component supports multi-choice; however, it does not provide a good way of filtering the choices.

A library that provides an advanced select component, select2 [67], has been used. It supports filtering, intuitive multi-choice selections, dynamic item creation and customizable styling. In order for it to be easily usable in

the Blazor project, a `Select2` razor component has been created, which wraps all of the communication with the javascript library.

3.10.6 Split

The process and data model pages contain a dynamic vertical split, allowing the user to alter the ratio between the left and right modeller panes by dragging the split line. A javascript library, split-js [68], has been used to provide this functionality. Since the panes might contain complex components, which require manual refreshing, the .NET codebase subscribes to an event that fires whenever the user drags the split.

3.11 Testing

Two approaches are most commonly used when testing Blazor apps [69]:

Unit testing is a preferred option if the tested .NET classes and Razor components do not depend on JS interop logic that manipulates the browser DOM. Unit tests are fast to execute, allow access to the Razor components and are generally more reliable.

E2E testing is used whenever a complex interaction with javascript codebase needs to be tested. The test cases are run in a browser instance against the built application, so E2E tests are much slower to execute and more difficult to maintain.

3.11.1 Unit Testing

The business logic of the application is contained within .NET classes called services. Any dependencies between services are defined using an interface, which means that the dependencies can be mocked, making it possible to test each service independently using unit tests. The Unit tests are contained within the `Web.Tests.Unit` project and cover the majority of the implemented services.

3.11.2 End-to-End Testing

An important functionality of the DasContract editor, which is susceptible to potential issues, is the synchronization of the DasContract and bpmn-js process models. The app relies on receiving and correctly processing synchronization events in order to keep the two models consistent, as inconsistency between the two models might lead to unexpected bugs and problems.

The full integration process cannot be tested using only unit tests, as the bpmn-js modeller is a javascript library and cannot be run independently in the context of the .NET tests [69]. An end-to-end testing approach was chosen

instead, where the entire application is launched in a browser window and automatically controlled using predefined commands. The bpmn-js modeller can be controlled programmatically using javascript commands, making it possible to simulate user actions without relying on the application's visual layout. This makes the test more robust, as changes to the layout will not affect the test cases.

Testing Framework

The Playwright for .NET [70] testing framework has been chosen for implementing the synchronization tests, as it provides a simple testing interface with all of the needed features. It allows to access and filter the DOM, making it possible to trigger events on browser elements. It also provides an interface to evaluate javascript code, which can be used to control the instance of the bpmn-js modeller.

Structure of the Test Cases

The test cases are run in accordance with the following steps:

1. A fresh browser context is created. The test case also creates a separate DasContract model, which is edited alongside the executed commands and serves as the expected state of the application. An option to create a new contract is then selected on the landing page of the editor.

2. The bpmn-js commands are executed to make changes to the process model. When executing the command, the test fixture also directly makes appropriate changes to the model that acts as the expected state.

3. When all commands have been executed, a download of the DasContract file is requested in the browser. The downloaded *actual* model is then compared to the *expected* model that has been directly modified alongside the executed commands.

3.11.3 User Feedback

Stable versions were periodically deployed during the development of the editor, making them publicly available to other researchers and developers of the DasContract project. Feedback was collected in the form of consultations and also in the form of GitHub issue pages, where the users could report any bugs and suggestions.

Some of the proposals that have been addressed include:

- A developer of a DasContract converter proposed that the users should have the option to change the IDs of process elements since the IDs were automatically generated and immutable. A change has been made

to make the IDs editable, according to certain rules (must be unique, no special characters)

- When saving a contract to a device, the default filename in the download prompt would be "contract.dascontract". A user proposed that the default filename should be filled in dynamically based on the actual name of the contract.
- When a new contract was created, the data model page did not contain an example. It has been proposed that it would be more user-friendly if it contained an example data model by default, which users can use as a template for their work.

Chapter 4

Standalone App Deployment

A client-side Blazor web application was presented in the previous chapter. This chapter describes the deployment of the implemented application as a progressive web application.

The first section summarizes the benefits and drawbacks of the standalone deployment technologies in the context of the implemented web app case study and explains why the PWA approach was chosen. The second section describes the process of making the existing web app available as a PWA. The third section describes how the user can install and update the app. The chosen approach is evaluated in the last section, and its benefits are summarized.

4.1 Choosing the Technology

Three approaches to converting a web app into a standalone application have been described in chapter 2 – Blazor Hybrid, Electron and PWAs. This section goes over the properties of standalone app frameworks in the context of the DasContract Editor and evaluates the most suitable framework.

Supported Platforms The DasContract editor is targeted to desktop platforms only. All technologies support all desktop platforms, except for Blazor Hybrid, which is not supported on Linux.

Maintainability Since the case study is a part of an ongoing open-source research, it is more than expected for the project to evolve and include new requirements. It is therefore imperative for the chosen technology to be as easy to maintain as possible. Based on the research, PWAs have the lowest maintenance overhead out of the mentioned technologies.

Capabilities The case study does not require any special platform-specific capabilities, such as access to the file system or system settings.

Maturity of the technology Both the Electron and PWA technologies have been in production use for several years and can be deemed as stable. Blazor Hybrid is still in preview and is not recommended for production environments.

Distribution The ease of distribution is also worth mentioning, as it concerns both the developers and users. The easiest technology to both distribute and consume is, without doubt, the PWA. The standalone PWA can be installed directly from the web version of the application.

Size and performance Since the app is not computationally demanding, top-level performance is not required. In terms of the size of the application, the PWA is the most lightweight out of the three.

Based on these observations, the PWA approach seems to be the most suitable technology in the context of the current case study – it is easy to maintain and distribute, is lightweight, stable and provides all of the necessary capabilities.

4.2 Integrating the PWA Approach

In order to make the web app installable as a standalone application using the PWA technology, two criteria need to be met: the web app manifest must be defined, and a service worker script must be present.

4.2.1 Web App Manifest

The manifest is a text file in JSON format containing information about how the app should be displayed on the target platform. The fields have been described in subsection 2.3.2. The contents of the web app manifest file can be seen in listing 2.

4.2.2 Service Worker

As described in subsection 2.3.1, service workers make it possible to cache static web resources, making them available offline. Blazor provides a template setup for making a Blazor app available as a PWA, including a basic service worker code that caches the static files [71]. The template service worker code has been slightly modified to fit the purposes of the case study app. A snippet of the most important parts of the service worker can be seen in listing 3.

The `onInstall` function is called when a new service is successfully installed. Blazor provides an assets manifest file, which is generated during compilation. This file lists all static resources served by the app, such as .NET assemblies, javascript and css files, etc [71]. The manifest is filtered using regular expressions, specifying which file extensions are required to be

```
{
  "name": "DasContract Editor",
  "short_name": "DasContract",
  "start_url": "./",
  "display": "standalone",
  "theme_color": "#212529",
  "icons": [
    {
      "src": "dist/logo/icon-512x512.png",
      "type": "image/png",
      "sizes": "512x512"
    },
    {
      "src": "dist/logo/icon-256x256.png",
      "type": "image/png",
      "sizes": "256x256"
    },
    {
      "src": "dist/logo/icon-192x192.png",
      "type": "image/png",
      "sizes": "192x192"
    }
  ]
}
```

Listing 2: Web app manifest defined for the standalone application.

cached by the service worker. The files that have been filtered are then saved into cache using an identifier that is unique to the version of the service worker.

The `onActivate` function is called when a new worker is activated. Since each service worker installation has its own cache, when a service worker gets deactivated, its cache gets left behind. This function clears the cache left behind by any deactivated service workers.

The `onFetch` function is called whenever a request to the server is being made. The function acts as a proxy, allowing to return cached resources instead of making requests to the server. The function first checks the method of the request, as only `GET` requests can be cached. Since Blazor dynamically inserts its pages into the index.html file, all navigate requests should return the index.html file, regardless of the address (the internals of Blazor handle the routing).

4.3 Installing and Updating

The app can be “installed” directly in the browser when visiting the web application (the install prompt in a google chrome browser can be seen in

```
const cacheNamePrefix = 'offline-cache-';
const cacheName = `${cacheNamePrefix}${self.assetsManifest.version}`;
const offlineAssetsIncl = [/\.dll$/, /\.pdb$/, /\.wasm/, /\.html/, /\.js$/,
    /\.json$/, /\.css$/, /\.woff$/, /\.ttf$/, /\.eot$/, /\.woff2$/, /\.svg$/,
    /\.png$/, /\.jpe?g$/, /\.gif$/, /\.ico$/, /\.blat$/, /\.dat$/, /\.dascontract$/,
    /\.xml$/];
const offlineAssetsExcl = [ /^service-worker\.js$/, /^routes\.json$/];

async function onInstall(event) {
    // Cache items from the assets manifest based on regex rules
    const assetsRequests = self.assetsManifest.assets
        .filter(asset => offlineAssetsIncl.some(pattern => pattern.test(asset.url)))
        .filter(asset => !offlineAssetsExcl.some(pattern => pattern.test(asset.url)))
        .map(asset => new Request(asset.url, { integrity: asset.hash }));
    await caches.open(cacheName).then(cache => cache.addAll(assetsRequests));
}

async function onActivate(event) {
    // Delete unused caches left behind by inactive service workers
    const cacheKeys = await caches.keys();
    await Promise.all(cacheKeys
        .filter(key => key.startsWith(cacheNamePrefix) && key !== cacheName)
        .map(key => caches.delete(key)));
}

async function onFetch(event) {
    let cachedResponse = null;
    if (event.request.method === 'GET') {
        // For all navigation requests, try to serve index.html from cache
        const shouldServeIndexHtml = event.request.mode === 'navigate';
        let request = shouldServeIndexHtml ? 'index.html' : event.request;
        const cache = await caches.open(cacheName);
        // Try to retrieve the request from cache, ignoring any query params
        cachedResponse = await cache.match(request, {ignoreSearch: true});
    }

    return cachedResponse || fetch(event.request);
}
```

Listing 3: A snippet of the service worker script.

Figure 4.1). The process of installation was put in quotes because the app is already cached on the device by the time the install prompt becomes available.

A new service worker is automatically installed whenever a new version of the web app is deployed. It does not interrupt the active service worker, so while the new files are being downloaded, the user can still use the old version of the app.

4.4 Evaluation

A cache-first PWA approach has been successfully implemented. Since the web app is entirely served as static pages and does not require additional communication with a backend server, it is fully functional in offline mode. The standalone app window can be seen in Figure 4.2.

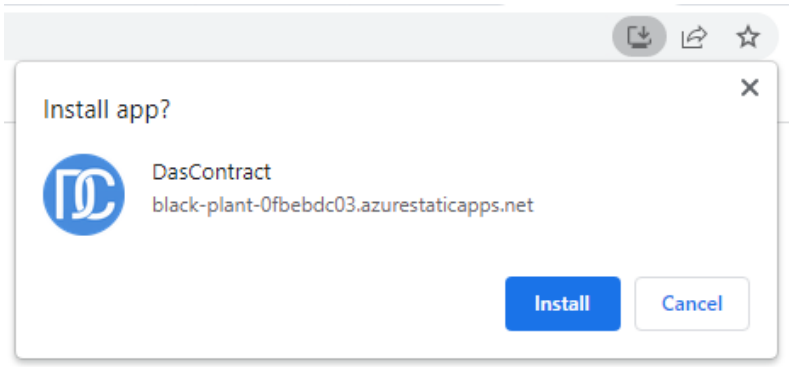

Figure 4.1: The PWA install prompt in a google chrome browser.

Compared to the other approaches, a major advantage of the PWA approach is that new releases of the web app version do not require separate deployment and testing of the standalone app. Instead, the version of the PWA is directly tied to the version of the web app, so no new deployment cycle has to be established and managed.

Another benefit is the lower storage requirements of the PWA. Unlike Electron apps, which have a storage overhead due to bundling chromium and node.js, PWAs make use of the installed browser to run in. The deployed application downloads and caches around 16MB of data.

Users also gain benefits even if they do not install the app to their device – the service worker caches the static files even when the app is not installed, which means that load times are reduced, and the web app is accessible in offline mode.

The drawback of the PWA approach is that not all native device features are available (such as direct access to the file system). This limitation is not a problem for the case study implementation, as the app does not require any special capabilities.

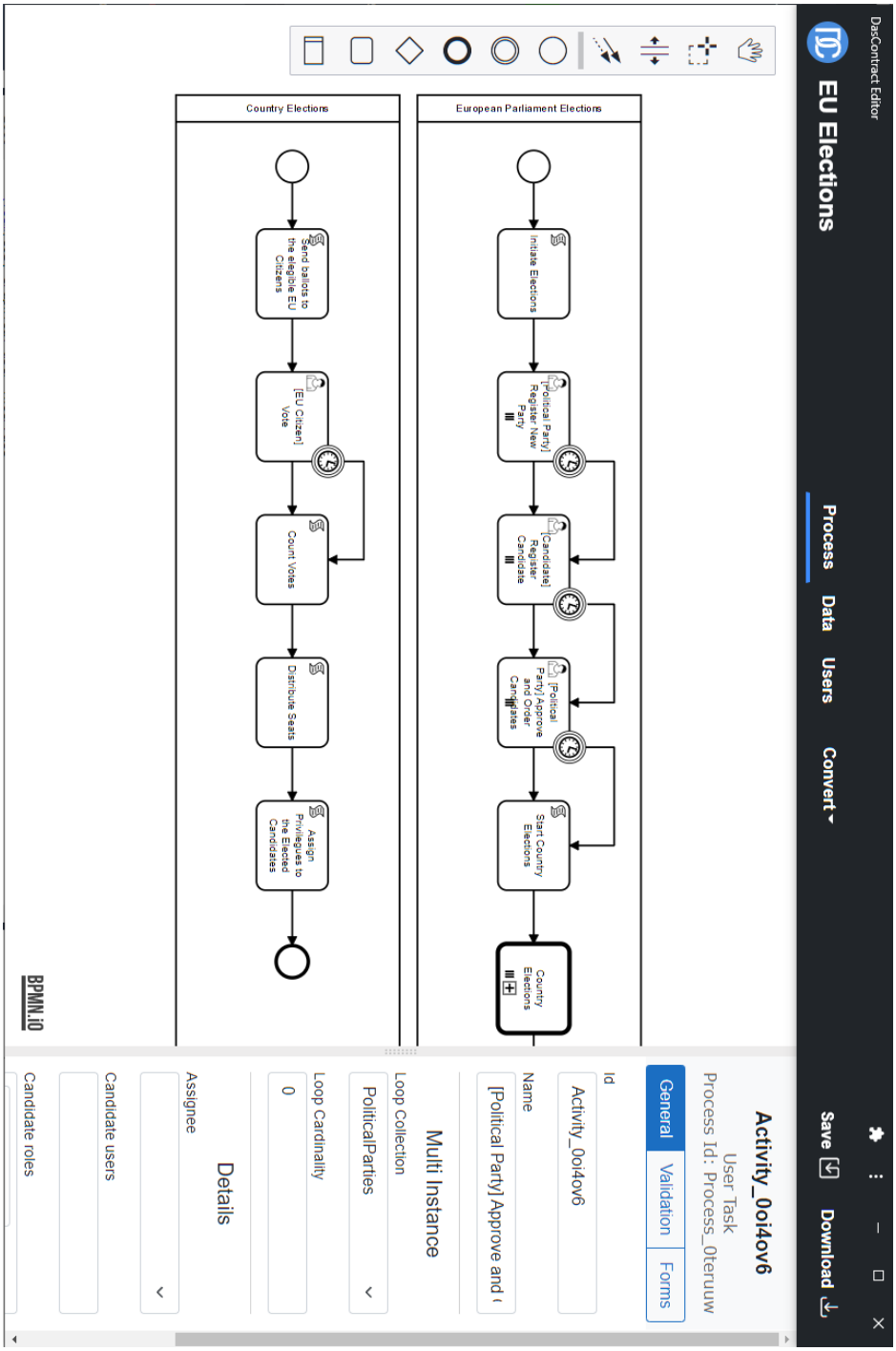

Figure 4.2: A screenshot of the PWA installed on Windows 10 using Google Chrome browser.

Conclusion

One of the aims of this thesis was to explore the possibilities of combining web app development in the Blazor WebAssembly framework with standalone app deployment whilst preserving the same codebase. Three different approaches were explored and compared. One of these approaches was then used in a complex case study, demonstrating that a client-side Blazor web application can be deployed as a Progressive Web Application, making it fully offline-capable with very little maintenance overhead.

In the practical part of the thesis, a visual blockchain smart contract editor was designed, implemented and successfully deployed as a standalone application. The implementation is fully open-source and is available in a public GitHub repository [15].

The editor is currently being used by other DasContract researchers to create smart contract case studies. The deployed application, including a video tutorial, is also freely accessible, allowing anyone interested in the DasContract project to design and generate smart contracts.

Future Work

DasContract is an ongoing research project, so it is expected that the domain-specific language will be extended in the future, requiring changes to be made to the editor. This was reflected in the application's design, making the parts of the applications most prone to changes easily extensible. An example is the detail bar of the process elements, which allows new properties and tabs to be easily added.

There are also several areas in which the DasContract Editor could be improved. The Monaco script editor could be configured to provide advanced features such as auto code completion, hints, automatic formatting and suggestions. Advanced process model validations could also be implemented, directly warning the users about errors in their process model before converting.

Bibliography

1. FOURAULT, Sébastien. *How Progressive Web Apps can drive business success* [online] [visited on 2022-04-24]. Available from: `https://web.dev/drive-business-success/`.
2. NOFER, Michael; GOMBER, Peter; HINZ, Oliver; SCHIERECK, Dirk. Blockchain. *Business & Information Systems Engineering*. 2017, vol. 59, no. 3, pp. 183–187. ISSN 1867-0202. Available from DOI: `10.1007/s12599-017-0467-3`.
3. GAYVORONSKAYA, Tatiana; MEINEL, Christoph. *Blockchain*. Springer International Publishing, 2021. ISBN 978-3-030-61559-8. Available from DOI: `10.1007/978-3-030-61559-8`.
4. NAKAMOTO, Satoshi. Bitcoin: A Peer-to-Peer Electronic Cash System [online]. [N.d.] [visited on 2022-01-26]. Available from: `https://bitcoin.org/bitcoin.pdf`.
5. SZABO, Nick. *Smart Contracts* [online] [visited on 2022-01-29]. Available from: `https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html`.
6. SWAN, Melanie. *Blockchain*. Sebastopol, CA: O'Reilly Media, 2015.
7. *Ethereum: EthereumProject* [online] [visited on 2022-04-17]. Available from: `https://ethereum.org/`.
8. *ETHEREUM VIRTUAL MACHINE (EVM)* [online] [visited on 2022-04-17]. Available from: `https://ethereum.org/en/developers/docs/evm/`.
9. *INTRODUCTION TO THE ETHEREUM STACK* [online] [visited on 2022-04-17]. Available from: `https://ethereum.org/en/developers/docs/ethereum-stack/`.

10. *SMART CONTRACT LANGUAGES* [online] [visited on 2022-04-17]. Available from: https://ethereum.org/en/developers/docs/smart-contracts/languages/.

11. KLICPERA, Jan. *A novel way of conducting legal contracts* [online] [visited on 2022-04-24]. Available from: https://janklicpera.medium.com/a-novel-way-of-conducting-legal-contracts-be54ceda39ad.

12. SKOTNICA, Marek; KLICPERA, Jan; PERGL, Robert. Towards Model-Driven Smart Contract Systems – Code Generation and Improving Expressivity of Smart Contract Modeling. In: *CIAO! Doctoral Consortium, EEWC Forum 2020*. CEUR, 2020. Available also from: http://ceur-ws.org/Vol-2825/paper1.pdf.

13. 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*. Cham: Springer International Publishing, 2020, pp. 149–166. ISBN 978-3-030-37933-9. Available also from: https://link.springer.com/chapter/10.1007/978-3-030-37933-9_10.

14. *BPMN Specification – Business Process Model and Notation* [online] [visited on 2022-04-24]. Available from: https://www.bpmn.org/.

15. SKOTNICA, Marek; KLICPERA, Jan; DROZDÍK, Martin; ŠELDER, Ondřej. *DasContract GitHub repository* [online] [visited on 2022-04-24]. Available from: https://github.com/CCMiResearch/DasContract.

16. DROZDÍK, Martin. *DasContract Plutus generator* [online] [visited on 2022-04-24]. Available from: https://github.com/drozdik-m/das-contract-plutus-generator.

17. ANČINEC, Petr. *Domain-Specific Languages for Off-chain UI in Decentralized Applications*. Praha, 2021. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, Department of Software Engineering. Supervisor: Marek Skotnica.

18. DROZDÍK, Martin. *Open-Source Legal Process Designer in .NET Blazor*. Praha, 2020. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, Department of Software Engineering. Supervisor: Marek Skotnica.

19. GOLDBERG, Ian; WAGNER, David; THOMAS, Randi; BREWER, Eric. A Secure Environment for Untrusted Helper Applications. 1996. Available also from: https://www.usenix.org/legacy/publications/library/proceedings/sec96/full_papers/goldberg/goldberg.pdf.

20. *Chromium sandbox* [online] [visited on 2022-02-26]. Available from: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/design/sandbox.md.

21. MCCLOUD, Scott; GOOGLE CHROME TEAM. *Chrome Google book* [online] [visited on 2022-02-26]. Available from: `https://www.google.com/googlebooks/chrome/big_26.html`.

22. *Introduction to ASP.NET Core Blazor* [online] [visited on 2022-01-29]. Available from: `https://docs.microsoft.com/en-gb/aspnet/core/blazor`.

23. *.NET | A developer platform for building all your apps.* [Online] [visited on 2022-04-20]. Available from: `https://dotnet.microsoft.com/`.

24. *Razor syntax reference for ASP.NET Core* [online] [visited on 2022-01-29]. Available from: `https://docs.microsoft.com/en-gb/aspnet/core/mvc/views/razor`.

25. *Document Object Model (DOM)* [online] [visited on 2022-01-30]. Available from: `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model`.

26. CHARBENEAU, Ed. *Blazor RenderTree Explained* [online]. 2020-06 [visited on 2022-01-30]. Available from: `https://www.infoq.com/articles/blazor-rendertree-explained`.

27. HIMSCHOOT, Peter. *Microsoft Blazor building web applications in .NET*. United States: Apress, 2020. ISBN 9781484259283.

28. *ASP.NET Core Blazor hosting models* [online] [visited on 2022-01-30]. Available from: `https://docs.microsoft.com/en-gb/aspnet/core/blazor/hosting-models`.

29. *WebAssembly* [online] [visited on 2022-02-27]. Available from: `https://webassembly.org/`.

30. *WebAssembly MDN Web Docs* [online] [visited on 2022-02-27]. Available from: `https://developer.mozilla.org/en-US/docs/WebAssembly`.

31. *Awesome WebAssembly Languages – A curated list of languages that compile directly to WASM* [online] [visited on 2022-02-27]. Available from: `https://github.com/appcypher/awesome-wasm-langs`.

32. *Defining Static Web Apps* [online] [visited on 2022-02-27]. Available from: `https://www.staticapps.org/articles/defining-static-web-apps/`.

33. *ASP.NET Core Blazor Hybrid* [online] [visited on 2022-04-16]. Available from: `https://docs.microsoft.com/en-us/aspnet/core/blazor/hybrid`.

34. *Electron Documentation* [online] [visited on 2022-02-26]. Available from: `https://www.electronjs.org/docs`.

35. *Chromium Browser Project* [online] [visited on 2022-02-26]. Available from: `https://www.chromium.org/Home/`.

36. *About Node.js* [online] [visited on 2022-02-26]. Available from: https://nodejs.org/en/about/.

37. *Electron Process Model* [online] [visited on 2022-02-26]. Available from: https://www.electronjs.org/docs/latest/tutorial/process-model.

38. MARTINEZ, Juan Cruz. How to Build Desktop Applications the Right Way Using Electron [online]. 2020 [visited on 2022-02-27]. Available from: https://betterprogramming.pub/how-to-build-desktop-applications-the-right-way-using-electron-aee5deedeb8c.

39. *Electron Context Isolation* [online] [visited on 2022-02-26]. Available from: https://www.electronjs.org/docs/latest/tutorial/context-isolation.

40. *Electron Inter-Process Communication* [online] [visited on 2022-02-26]. Available from: https://www.electronjs.org/docs/latest/tutorial/ipc.

41. *Electron Distribution* [online] [visited on 2022-02-26]. Available from: https://www.electronjs.org/docs/latest/tutorial/application-distribution.

42. SHEPPARD, Dennis. *Beginning Progressive Web App Development: Creating a Native App Experience on the Web*. Apress, 2017. ISBN 9781484230893. Available from DOI: 10.1007/978-1-4842-3090-9.

43. RICHARD, Sam; LEPAGE, Pete. *What are Progressive Web Apps?* [Online] [visited on 2022-03-05]. Available from: https://web.dev/what-are-pwas/.

44. *Introduction to progressive web apps* [online] [visited on 2022-03-05]. Available from: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction.

45. *What makes a good Progressive Web App?* [Online] [visited on 2022-03-05]. Available from: https://web.dev/pwa-checklist.

46. *Service Worker API* [online] [visited on 2022-03-05]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.

47. GAUNT, Matt. *Service Workers: an Introduction* [online] [visited on 2022-03-05]. Available from: https://developers.google.com/web/fundamentals/primers/service-workers.

48. *Service workers* [online] [visited on 2022-03-05]. Available from: https://web.dev/learn/pwa/service-workers.

49. *Web app manifests* [online] [visited on 2022-03-06]. Available from: https://developer.mozilla.org/en-US/docs/Web/Manifest.

50. LEPAGE, Pete; BEAUFORT, François; STEINER, Thomas. *Add a web app manifest* [online] [visited on 2022-03-06]. Available from: `https://web.dev/add-manifest`.

51. NEWMAN, Jared. *Firefox just walked away from a key piece of the open web* [online] [visited on 2022-03-08]. Available from: `https://www.fastcompany.com/90597411/mozilla-firefox-no-ssb-pwa-support`.

52. DEVERIA, Alexis; SCHOORS, Lennart. *Can I Use?* [Online] [visited on 2022-03-08]. Available from: `https://caniuse.com/`.

53. *Supported platforms for .NET MAUI apps* [online] [visited on 2022-04-16]. Available from: `https://docs.microsoft.com/en-us/dotnet/maui/supported-platforms`.

54. RAKOWSI, Filip; GRZYBOWSKA, Kaja; KWIECIEŃ, Aleksandra; KARWATKA, Piotr. *The PWA Book – Open guide to progressive web apps* [online]. Divante eCommerce Software House, 2019 [visited on 2022-04-17]. Available from: `https://www.divante.com/pwabook`.

55. *Electron* [online]. GitHub, [n.d.] [visited on 2022-04-16]. Available from: `https://github.com/electron/electron`.

56. *Releasing projects on GitHub* [online] [visited on 2022-04-16]. Available from: `https://docs.github.com/en/repositories/releasing-projects-on-github`.

57. H., Gordon. *Performance Comparison: Flutter Desktop vs. Electron* [online]. 2021 [visited on 2022-04-17]. Available from: `https://getstream.io/blog/flutter-desktop-vs-electron/`.

58. SHVETS, Alexander. *Dive Into Design Patterns*. 2019.

59. DOOLEY, John. *Software development, design and coding : with patterns, debugging, unit testing, and refactoring*. Berkeley, Californial: Apress, 2017. ISBN 9781484231524.

60. *Bootstrap* [online] [visited on 2022-04-23]. Available from: `https://getbootstrap.com/`.

61. *Syntatically Awesome StyleSheets (SASS)* [online] [visited on 2022-04-23]. Available from: `https://sass-lang.com/`.

62. *Monaco Editor* [online]. GitHub [visited on 2022-04-03]. Available from: `https://github.com/Microsoft/monaco-editor`.

63. *BlazorMonaco* [online]. GitHub [visited on 2022-04-03]. Available from: `https://github.com/serdarciplak/BlazorMonaco`.

64. *Mermaid JS* [online]. GitHub, [n.d.] [visited on 2022-04-03]. Available from: `https://github.com/mermaid-js/mermaid`.

65. *bpmn-js – BPMN 2.0 viewer and editor.* [Online] [visited on 2022-04-16]. Available from: `https://bpmn.io/toolkit/bpmn-js/`.

66. *dmn-js – DMN viewer and editor.* [Online] [visited on 2022-04-16]. Available from: `https://bpmn.io/toolkit/dmn-js/`.

67. *Select2 - the jQuery replacement for select boxes* [online] [visited on 2022-04-16]. Available from: `https://select2.org/`.

68. *Split JS* [online] [visited on 2022-04-16]. Available from: `https://split.js.org/`.

69. *Test Razor components in ASP.NET Core Blazor* [online] [visited on 2022-04-19]. Available from: `https://docs.microsoft.com/en-us/aspnet/core/blazor/test`.

70. *Playwright for .NET* [online] [visited on 2022-04-19]. Available from: `https://playwright.dev/dotnet/`.

71. *ASP.NET Core Blazor Progressive Web Application (PWA)* [online] [visited on 2022-04-18]. Available from: `https://docs.microsoft.com/en-us/aspnet/core/blazor/progressive-web-app`.

Appendix A

Acronyms

BPMN Business Process Model and Notation

CSS Cascading Style Sheet

DMN Decision Model and Notation

DOM Document Object Model

DSL Domain-specific Language

EVM Ethereum Virtual Machine

GUI Graphical User Interface

HTML HyperText Markup Language

JS JavaScript

JSON JavaScript Object Notation

PWA Progressive Web Application

SASS Syntatically Awesome Style Sheets

SC Smart Contract

UML Unified Modeling Language

WASM WebAssembly

XML Extensible markup language

Appendix B

Contents of Enclosed SD Card

- `readme.txt` ..... the file with SD card contents description
- `tutorial.mp4` ..... user tutorial video
- `src` ..... the directory of source codes
  - `DasContract.Abstraction` ..... DasContract Abstraction project dependency
  - `DasContract.Blockchain.Solidity` . DasContract Solidity converter project dependency
  - `DasContract.Editor.Web` .... source code of the DasContract Editor
  - `DasContract.Editor.Web.Tests.E2E` ..... end-to-end tests
  - `DasContract.Editor.Web.Tests.Unit` ..... unit tests
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format