FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání bakalářské práce

Název:	Přepracování a rozšíření webového vývojového prostředí pro jazyk Karel
Student:	Jan Jörka
Vedoucí:	Ing. Jan Blizničenko
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Cílem práce je kompletní přepracování webového vývojového prostředí pro programovací jazyk Karel karlbot.cz z důvodu snazší dlouhodobé udržitelnosti, testovatelnosti a modernizace. Dále také doplnění některých nových funkčností:
- Možnost ukládat programy v Karlovi na serveru a sdílet je s ostatními uživateli.
- Ladění programů včetně krokování, breakpointů a zobrazení zásobníku volání.
- Výzvy ke splnění v jazyku Karel s automatickým hodnocením.

Stávající vývojové prostředí umožňuje na jednom místě vytvářet a spouštět programy v programovacím jazyce Karel a editovat města, ve kterých se robot Karel pohybuje. Součástí je také interpret jazyka samotného. Aplikace je dostupná přímo v prohlížeči jako single page aplikace.

- Proveďte analýzu stávajícího řešení karlbot.cz a také dalších podobných nástrojů.
- Proveďte rešerši vhodných technologií.
- Vytvořte návrh aplikace.
- Implementujte aplikaci.
- Otestujte a zdokumentujte vytvořené řešení.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 3. ledna 2023 v Praze.

Bakalářská práce

PŘEPRACOVÁNÍ A ROZŠÍŘENÍ WEBOVÉHO VÝVOJOVÉHO PROSTŘEDÍ PRO JAZYK KAREL

Jan Jörka

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jan Blizničenko
11. května 2023

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2023 Jan Jörka. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.

Odkaz na tuto práci: Jörka Jan. *Přepracování a rozšíření webového vývojového prostředí pro jazyk Karel.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Obsah

Poděkování viii

Prohlášení ix

Abstrakt x

Seznam zkratek xi

Úvod 1

1 Programovací jazyk Karel 3
- 1.1 Původní verze . . . 3
- 1.2 Pozdější varianty . . . 4

2 Analýza existujících řešení 5
- 2.1 Robot Karel . . . 5
- 2.2 Stanford Karel . . . 6
- 2.3 Karel 1981 . . . 7
- 2.4 Karel 3D . . . 8
- 2.5 karlbot.cz . . . 9
- 2.6 Shrnutí . . . 10
  - 2.6.1 Výsledek . . . 10

3 Analýza řešení karlbot.cz 11
- 3.1 Varianta jazyka Karel . . . 11
  - 3.1.1 Programy pro interakci s městem . . . 12
  - 3.1.2 Programy pro testování okolí . . . 12
- 3.2 Neopravené chyby . . . 12
- 3.3 Použité technologie a struktura projektu . . . 12
- 3.4 Implementace jazyka Karel . . . 13
  - 3.4.1 Fáze kompilace . . . 13
  - 3.4.2 Immutabilní syntaktický strom . . . 14
  - 3.4.3 Interpret . . . 15
  - 3.4.4 Ukázka použití . . . 16
- 3.5 Implementace aplikace . . . 16
  - 3.5.1 Editor . . . 17
- 3.6 Shrnutí nalezených nedostatků . . . 17

4 Analýza požadavků 19
- 4.1 Funkční požadavky . . . 19
- 4.2 Nefunkční požadavky . . . 20

5 Návrh **21**
5.1 Architektura aplikace . . . 21
5.2 Výběr technologie pro klienta . . . 22
5.2.1 React . . . 22
5.2.2 Angular . . . 22
5.2.3 Vue . . . 23
5.2.4 Rozhodnutí . . . 23
5.3 Výběr technologie pro server . . . 23
5.3.1 ASP.NET Core . . . 23
5.3.2 Spring . . . 24
5.3.3 Express . . . 24
5.3.4 Rozhodnutí . . . 24
5.4 Výběr rozhraní mezi klientem a serverem . . . 24
5.4.1 REST API . . . 24
5.4.2 GraphQL . . . 25
5.4.3 gRPC . . . 25
5.4.4 Rozhodnutí . . . 25
5.5 Výběr databáze . . . 25
5.5.1 SQL . . . 26
5.5.2 NoSQL . . . 26
5.5.3 Rozhodnutí . . . 26
5.6 Přihlašování uživatelů . . . 27
5.7 Automatické hodnocení výzev . . . 28
5.7.1 Pomocí skriptu . . . 28
5.7.2 Pomocí testovacích případů . . . 28
5.8 Databázové schéma . . . 29
5.9 REST API . . . 30
5.10 Shrnutí architektury . . . 30
5.11 Obrazovky . . . 31
5.11.1 Editor projektu . . . 31
5.11.2 Seznam projektů . . . 32
5.11.3 Seznam výzev . . . 33
5.11.4 Výzva . . . 33
5.11.5 Editor výzvy . . . 34
5.11.6 Přihlášení . . . 35

6 Implementace **37**
6.1 Implementace serverové části . . . 37
6.1.1 Architektura . . . 37
6.1.2 Celkový pohled . . . 39
6.1.3 Entity . . . 39
6.1.4 Repozitáře . . . 39
6.1.5 Služby . . . 40
6.1.6 Kontrolery . . . 40
6.1.7 Uživatelé a jejich autentizace . . . 41
6.1.8 Hodnocení výzev . . . 41
6.1.9 Dependency injection . . . 42
6.1.10 Konfigurace aplikace . . . 43
6.2 Implementace klientské části . . . 43
6.2.1 Architektura . . . 43
6.2.2 Celkový pohled . . . 43
6.2.3 Správa stavu . . . 44

6.2.4 Routování stránek 45
6.2.5 Editor 46
6.2.6 Editor kódu 47
6.2.7 Editor města 50
6.2.8 Markdown 52
6.2.9 Přihlašování uživatelů 52
6.2.10 Komunikace se serverem 53
6.2.11 Vzhled 54
6.2.12 Dependency injection 54
6.3 Implementace jazyka Karel 55
6.3.1 Kompilátor 56
6.3.2 Interpret 56
6.3.3 Město 57
6.3.4 Ostatní změny 58
6.3.5 Ukázka použití 58

7 Testování 59
7.1 Typy testů 59
7.2 Testování serverové části 59
7.3 Testování klientské části 60
7.4 Testování knihovny jazyka Karel 61
7.5 Continuous integration 61
7.6 Uživatelské testování 62

8 Dokumentace 63
8.1 Třídy, metody a vlastnosti 63
8.2 REST API 64

9 Nasazení 65
9.1 Server 65
9.2 Doména 65
9.3 Certifikát 65

Závěr 67

A Snímky obrazovek 69

Obsah přiloženého archivu 77

Seznam obrázků

1.1 Ilustrace Karlova světa v Pattisově knize [2, s. 3] 4

2.1 Aplikace Robot Karel s programem pro vytvoření rámečku kolem města [6] . . . 5
2.2 Aplikace Stanford Karel s řešením finální lekce – nalezení středu města[7] 6
2.3 Aplikace Karel 1981 s programem pro přenesení značky [3] 7
2.4 Aplikace Karel 3D s programem pro nalezení cesty z bludiště [8] 8
2.5 Aplikace karlbot.cz s programem pro nalezení cesty z bludiště [10] 9

3.1 Adresářová struktura řešení karlbot.cz 13
3.2 Fáze kompilace 13
3.3 Zelený a č strom – myšlenka z kompilátoru Roslyn [22] 15

5.1 Realizace tříúrovňové architektury v této aplikaci 21
5.2 Navržený proces autentizace 27
5.3 Konceptuální model databáze 29
5.4 Navržená architektura aplikace 30
5.5 Wireframe obrazovky editoru projektu 31
5.6 Wireframe obrazovky editoru projektu ve verzi pro úzké rozlíšení 32
5.7 Wireframe obrazovky seznamu projektů 32
5.8 Wireframe obrazovky seznamu výzev 33
5.9 Wireframe obrazovky výzvy 33
5.10 Wireframe obrazovky editoru výzvy 34
5.11 Wireframe obrazovky přihlášení 35

6.1 Přehled složek v kořeni repozitáře 37
6.2 Porovnání třívrstvé a zvolené architektury 38
6.3 .NET projekty serverové části aplikace 38
6.4 Middleware pipeline zpracovávající HTTP požadavky 39
6.5 Složky s moduly nového řeření 43
6.6 One-way data flow 44
6.7 Služby používané editorem 46
6.8 Výsledný editor kódu 49
6.9 Diagram tříd uživatelského rozhraní editoru města 50
6.10 Diagram tříd pro vykreslování města 51
6.11 Výsledný editor města 52
6.12 Stránka editoru projektu výsledné aplikace ve světlém režimu 54
6.13 Nová adresářová struktura knihovny jazyka Karel 55

7.1 Ukázka z průběhu end-to-end testu v uživatelském rozhraní frameworku Cypress 60

8.1 Ukázka části dokumentace REST API v nástroji Swagger UI 64

A.1 Obrazovka seznamu projektů 69
A.2 Obrazovka výzvy 70

A.3 Obrazovka editoru výzvy (její část) 70

Seznam tabulek

2.1 Porovnání funkcionalit existujících řešení 10

5.1 Příklad REST API 25
5.2 Návrh endpointů REST API 30

6.1 Implementovaná rozšíření CodeMirror editoru 47

Seznam výpisů kódu

3.1 Implementace DFS algoritmu ve variantě Karla karlbot.cz [10] 11
3.2 Instrukce bajtkódu do kterých se přeloží cyklus s předem daným počtem opakování 15
3.3 Ukázka použití knihovny jazyka Karel ve starém řešení 16
6.1 Společné generické rozhraní repozitáře ze kterého dědí všechny ostatní 39
6.2 Ukázka použití knihovny Entity Framework Core v repozitáři projektů 40
6.3 Ukázka metody kontroleru pro získání aktuálního uživatele 40
6.4 Ukázka obsahu JWT tokenu vydaného serverem 41
6.5 Ukázka komunikace s TypeScript stranou pomocí knihovny ClearScript 42
6.6 Ukázka části konfigurace DI kontejneru 43
6.7 Ukázka routovacích pravidel ze souboru `app.routes.ts` 45
6.8 Ukázka tokenů Lezer gramatiky pro jazyk Karel 47
6.9 Ukázka neterminálních symbolů Lezer gramatiky pro jazyk Karel 48
6.10 Výsledné veřejné rozhraní komponenty editoru 49
6.11 Vykreslení obdélníku pro zvýraznění vybraných dlaždic města 51
6.12 Ukázka implementace dotykového gesta pro přiblížení 51
6.13 Ukázka použití vytvořené Markdown komponenty 52
6.14 Veřejné rozhraní služby pro přhlašování 53
6.15 HTTP interceptor přidávající do požadavků autorizační token 53
6.16 Ukázka konfigurace kořenového injectoru 55
6.17 Veřejné rozhraní třídy `SourceMap` 56
6.18 Příklad funkce předávající kontrolu zpět prohlížeči 57
6.19 Ukázka použití knihovny jazyka Karel v novém řešení 58
7.1 Ukázka konfigurace SQL Server kontejneru v nastavení CI/CD pipeline 61

Chtěl bych poděkovat vedoucímu mé práce Ing. Janu Blizničenkovi za jeho cenné rady a připomínky. Dále bych chtěl poděkovat rodině a přátelům za podporu během celého studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 11. května 2023 . . . .

Abstrakt

Tato bakalářská práce se zabývá tvorbou webového vývojového prostředí pro výukový programovací jazyk Karel. Vývojové prostředí umožňuje pokročilou editaci zdrojového kódu a města, ve kterém se robot Karel pohybuje, spouštění vytvořených programů, jejich ladění, ukládání na server, sdílení a tvorbu automaticky hodnocených programovacích úkolů (výzev). Navazuje na aplikaci karlbot.cz, která je však jen velmi jednoduchá, nedokončená a ve špatném technickém stavu. V rámci práce je celá od začátku přepracována.

V práci je provedena její analýza a analýza dalších aplikací stejného zaměření. Na jejich základě jsou sestaveny požadavky na novou aplikaci a navržena její implementace. Serverová část je implementována ve frameworku ASP.NET Core a klientská ve frameworku Angular. Je mimo jiné popsán také proces integrace editoru zdrojového kódu nebo automatického hodnocení výzev. Dále je aplikace otestována, zdokumentována a nasazena. Výsledkem je plně funkční webová aplikace dostupná komukoliv na doméně karlbot.dev.

Klíčová slova programovací jazyk Karel, vývojové prostředí, webová aplikace, Angular, ASP.NET Core, TypeScript, C#

Abstract

This bachelor thesis deals with the creation of a web development environment for the Karel educational programming language. The development environment allows advanced editing of the source code and the town in which the Karel robot moves, running the created programs, debugging them, saving them to the server, sharing them and creating automatically evaluated programming tasks (challenges). It builds on the karlbot.cz application, which is, however, very simple, unfinished and in poor technical condition. As part of the thesis, it is reworked from scratch.

The thesis includes an analysis of the application and other applications with the same focus. Based on this analysis, the requirements for the new application are specified and its implementation is designed. The server part is implemented in ASP.NET Core framework and the client part in Angular framework. Among other things, the process of integrating a source code editor or automatic challenge evaluation is also described. Furthermore, the application is tested, documented and deployed. The result is a fully functional web application available to anyone on the karlbot.dev domain.

Keywords Karel programming language, development environment, web application, Angular, ASP.NET Core, TypeScript, C#

Seznam zkratek

ACID	Atomicity Consistency Isolation Durability
API	Application Programming Interface
ASP.NET	Active Server Pages .NET
BaaS	Backend as a Service
CDK	Component Dev Kit
CI/CD	Continuous Integration / Continuous Delivery
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
DFS	Depth First Search
DI	Dependency Injection
DNS	Domain Name System
ER	Entity Relationship
FTP	File Transfer Protocol
FTPS	FTP Secure
gRPC	gRPC Remote Procedure Call
GUID	Globally Unique Identifier
HSTS	HTTP Strict Transport Security
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identifikátor
IIS	Internet Information Services
IP	Internet Protocol
JSON	JavaScript Object Notation
JSX	JavaScript XML
JWT	JSON Web Token
LINQ	Language Integrated Query
MVC	Model View Controller
NoSQL	Not only SQL
ORM	Object Relational Mapping
R.U.R.	Rossumovi univerzální roboti
REST	Representational State Transfer
RPC	Remote Procedure Call
RxJS	Reactive Extensions for JavaScript
SDK	Software Development Kit
SPA	Single Page Application
SQL	Structured Query Language
TLS	Transport Layer Security
UI	User Interface
URL	Uniform Resource Locator
XML	Extensible Markup Language

Úvod

Výukový programovací jazyk Karel je na světě již velmi dlouho. Ovládání virtuálního robota ve 2D mřížce pomocí jednoduchého programovacího jazyka se ukázalo být chytlavým konceptem, a za několik desítek let vzniklo nepřeberné množství jeho variant a implementací. Přesto pro něj na dnešní poměry neexistuje moderní vývojové prostředí, které by v něm umožňovalo vytvářet programy s úrovní podpory editoru, jaká je dnes běžná ve vývojových prostředích klasických programovacích jazyků. Nejlépe ještě přímo ve webovém prohlížeči.

Dřívějším pokusem o takové prostředí od autora této práce byla webová aplikace karlbot.cz, která ale stále má k němu velmi daleko. Nabízí pokročilejší možnosti editace kódu s městem a umožňuje vytvořený program spustit. Nicméně tam její možnosti končí a stále chybí další nástroje pro ladění programů, jejich jednoduché online ukládání a sdílení mezi ostatní uživatele. Dále by se z povahy jazyka nabízelo třeba také zadávání a automatické hodnocení programovacích úkolů.

Kromě toho, že je toto prostředí nedokončené, obsahuje také mnoho chyb a kvalita jeho technického návrhu a implementace je pro další rozvoj nevyhovující. Potřebovalo by celé od základu vytvořit znovu. V aktuální fázi navíc řešení neobsahuje ani serverovou část, která je nutná pro implementaci některých funkcionalit.

Cíle práce

Hlavním cílem práce je navázat na toto řešení, napravit jeho technický stav a přidáním výše zmíněných funkcionalit vytvořit moderní a pokročilé webové vývojové prostředí pro jazyk Karel, které by konkurovalo těm stávajícím, a i je ve většině ohledů předčilo. S pomocí metod softwarového inženýrství tak bude třeba ho od začátku znovu analyzovat, navrhnout a přepracovat. To tak ve své podstatě znamená, na jeho základě vytvořit řešení nové.

V zadání práce pak byly specifikovány požadované rozšiřující funkcionality. Nejprve půjde o vytvoření serverové části pro ukládání a sdílení vytvořených projektů. Poté přidání podpory pro ladění programů, což mimo jiné zahrnuje krokování a nastavování breakpointů na jednotlivé řádky kódu. Poslední novou funkcionalitou bude vytváření a automatické hodnocení výzev v jazyku Karel – administrátor aplikace vytvoří zadání výzvy s definicí pravidel pro její splnění a uživatel pak může odevzdat její řešení. To mu bude následně automaticky ohodnoceno.

V teoretické části nejdříve budou pro vhled do současného stavu problematiky vývojových prostředí jazyka Karel porovnána dostupná řešení a provedena podrobnější analýza aktuálního nevyhovujícího řešení karlbot.cz. To bude analyzováno včetně jeho implementace a jejích nedostatků. Následovat bude analýza požadavků na nové řešení. V závěru teoretické části poté rešerše a výběr technologií, vhodných pro jeho implementaci. Tou se bude zabývat praktická část, která bude dále zahrnovat i proces testování a dokumentace. Na závěr pak bude řešení nasazeno.

Kapitola 1
Programovací jazyk Karel

Myšlenka na vznik jazyka se zrodila v hlavě profesora Richarda E. Pattise, v té době studujícího na univerzitě Stanford. Ten zastával názor, že by se studenti měli začít učit programování v prostředí, které je jednoduché a nezatěžuje je nepotřebnými technickými detaily. Navrhl tak jednoduchý programovací jazyk, pomocí jehož příkazů je možné ovládat virtuálního robota Karla a plnit s ním různé úkoly. [1] Jazyk stejně jako robota pojmenoval Karel a světu ho poprvé představil v roce 1981 ve své knize „Karel the Robot: A Gentle Introduction to the Art of Programming“ [2]. Jeho název odkazuje na Karla Čapka, v jehož divadelní hře R.U.R. poprvé zaznělo slovo robot [2, s. 1].

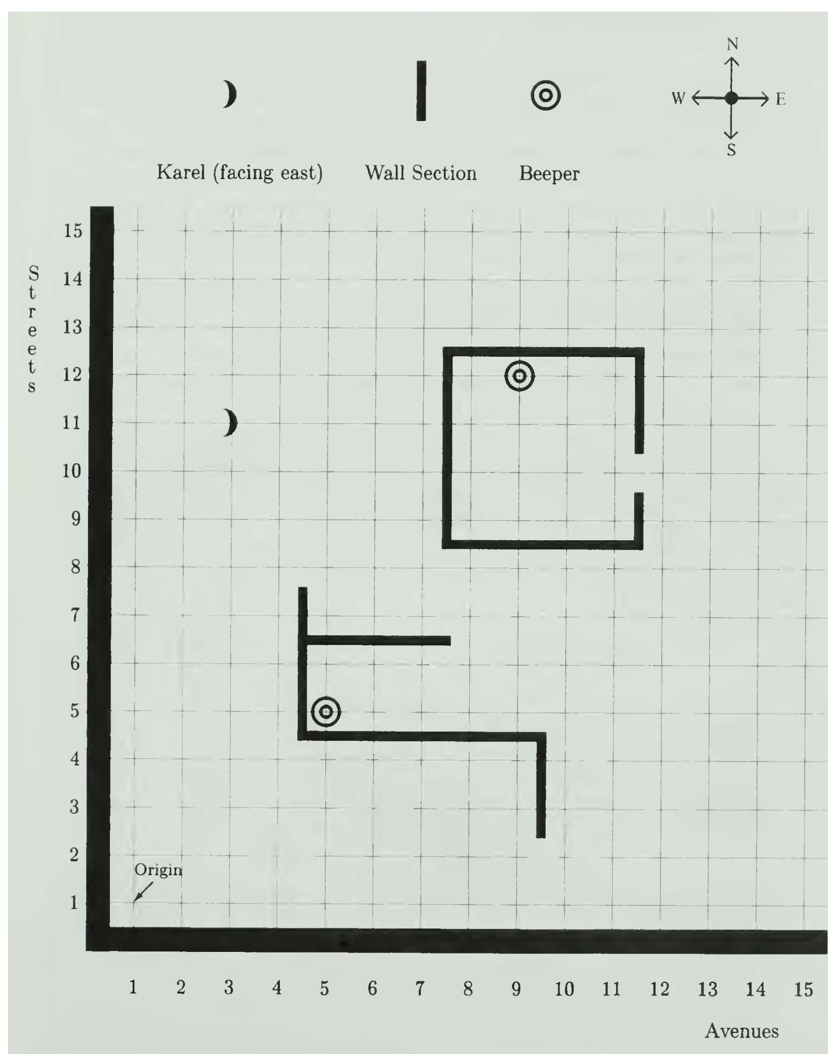
1.1 Původní verze

Podle původní Pattisovy verze popsané v jeho knize [2], byl svět, ve kterém se Karel pohyboval, tvořen mřížkovou sítí ze svislých a vodorovných čar. Svislé se v originále nazývaly „avenues“ a vodorovné „streets“. Průsečík těchto dvou typů čar se nazýval „corner“ (dále roh). Svět byl nekonečný a pouze zleva a zdola ohraničen zdmi. Zdi se mohli nacházet také na úseku mezi avenues či streets. Karel mohl být umístěn a pohybovat se pouze po rozích. Pomocí zdí mu bylo možné přesun mezi sousedními rohy blokovat. V jeho světě byly také standardním způsobem definovány světové strany, sever byl nahoře, napravo východ, dole jih a nalevo západ. Celý popsaný svět ilustruje obrázek 1.1 převzatý z Pattisovy knihy.

Co se týče ovládání robota, tak měl zabudovaných pouze 5 základních příkazů pro ovlivňování stavu prostředí. Například obsahoval příkaz pro otočení se na místě o 90 stupňů doleva, ale záměrně už ne žádný pro otočení doprava. Bylo totiž očekáváno, že si programátor tento příkaz složí ze tří po sobě jdoucích pro otočení doleva [2, s. 7]. Druhý a také poslední příkaz pro pohyb byl krok vpřed, který posunul Karla o jednu pozici ve směru, ve kterém byl natočený. Dále uměl na roh, kde se aktuálně nacházel, položit bzučák (v originále „beeper“, překlad z [3]) a nebo z něj naopak bzučák sebrat a dát si ho do batohu. Posledním příkazem byl příkaz pro vypnutí robota, který umožňoval vykonávání celého programu ukončit. Tímto příkazem dokonce muselo končit vykonávání každého programu, jinak se robot vypnul s chybou [2, s. 7, 9].

Z řídících struktur jazyk obsahoval podmínku s nepovinnou větví else, podmíněný cyklus a cyklus s předem daným počtem opakování. Bylo také možné vytvářet své vlastní příkazy.

V podmínkách a podmíněných cyklech šlo testovat, jestli je roh před Karlem volný, stejně tak pro roh vlevo a vpravo. Dokázal zjistit, zda na rohu, kde stojí, je bzučák a jestli si nese nějaký v batohu. Poznal také, jestli je otočený na předem danou světovou stranu. Všechny tyto testy byly dostupné i ve znegované variantě, aby se mohl ptát i na opak.



■ **Obrázek 1.1** Ilustrace Karlova světa v Pattisově knize [2, s. 3]

1.2 Pozdější varianty

V průběhu let vzniklo nepřeberné množství jeho dalších variant a implementací. Jednotlivé varianty se různí, co se tyče konkrétních možností a pokročilosti jazyka a často také v terminologii použité pro prvky prostředí, ve kterém se Karel pohybuje. Bzučákům se někdy říká značky a světu, kde robot žije, zase město. Většinou se také robot nepohybuje po průsečících hran čtvercové sítě, ale po dlaždicích, které jsou jimi definovány. Přesto, ale základ zůstává stejný, umožňuje pomocí kódu s několika základními příkazy pohybovat s robotem ve světě s typicky obdélníkovými rozměry, tvořeném průchozími a neprůchozími poli. Na pole pokládat a zase z nich sbírat různé prvky sloužící k jejich označení a pomocí virtuálních senzorů testovat stav okolního prostředí. Navzdory tomu, že jeho příkazy jsou typicky velmi jednoduché a většinou neobsahuje například ani proměnné, tak díky běžné podpoře rekurzivních volání je v něm možné implementovat i složitější algoritmy.

Kapitola 2

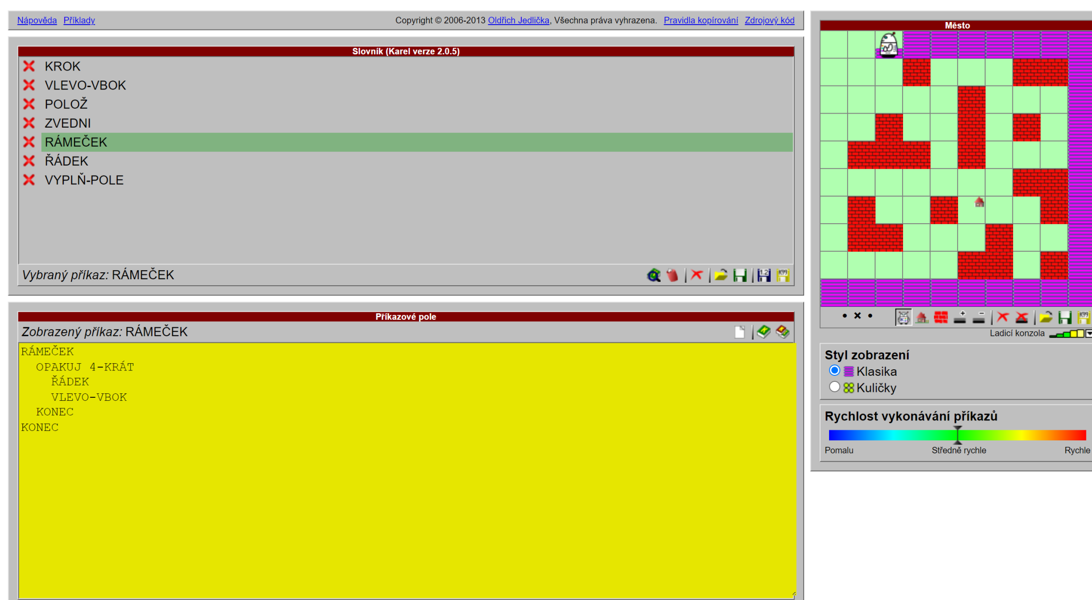
Analýza existujících řešení

Předtím, než bude možné začít s analýzou nového řešení, je třeba získat přehled o těch již existujících. Cílem této kapitoly je jejich porovnání a nalezení jejich největších nedostatků.

Implementací jazyka Karel existuje opravdu velké množství, nicméně převážná většina z nich je již velmi zastaralých, těžkopádných nebo v dnešní době už nedostupných. Z tohoto důvodu bylo vybráno jen několik těch nejzajímavějších. Z rozpočtových důvodů nebyly zahrnuty ani částečně nebo úplně placená řešení. Konkrétně tak byly vyřazeny zejména implementace jazyka Karel na vzdělávacích platformách CodeHS [4] a NCLab [5]. Jeví se ovšem jako poměrně pokročilé.

Přesto, že jednotlivá řešení používají pro prvky Karlova světa různé názvy, tak pro jednotnost pojmů v rámci celé práce bude o světě, kde se Karel pohybuje, hovořeno jako o městě, o bzučácích, které pokládá, jako o značkách a o polích, ze kterých se skládá město, jako o dlaždicích.

2.1 Robot Karel



Obrázek 2.1 Aplikace Robot Karel s programem pro vytvoření rámečku kolem města [6]

Nejjednodušším řešením, a také už poměrně zastaralým, je webová aplikace Robot Karel [6] Oldřicha Jedličky. Ta navazuje na původně desktopovou aplikaci od stejného autora. V levé části stránky se nachází editor kódu. V textovém poli lze psát zdrojový kód Karlova programu, který se po uložení objeví v seznamu programů nad textovým polem. Z tohoto seznamu je poté možné vytvořené programy spouštět nad městem tvořeným v editoru v pravé části stránky.

Prostředí je velmi jednoduché a má také mnoho nevýhod. Editor zdrojového kódu nepodporuje žádné vlastnosti pro ulehčení psaní kódu, jako je například automatické odsazování, napovídání, barevné zvýrazňování nebo kontrola chyb. Editor města neumožňuje změnu rozměrů na jiné než výchozích 10×10. Kód a město je nutné uložit zvlášť, a to pouze ručním zkopírováním zobrazené textové reprezentace kódu či města do schránky. Ani vzhled aplikace na tom není o moc lépe a působí již velmi zastarale. Celkově tedy aplikace plní základní účel spouštění programů v jazyku Karel ve webovém prostředí, nicméně uživatelsky nepřívětivým způsobem.

2.2 Stanford Karel

STANFORD KAREL Unit 12 Lesson 1 Reference

```
// Your final task is to teach
// Karel to find the midpoint
// of any world. You can assume
// that all worlds are square.
function main(){
   moveToMiddle();
   putBeeper();
   turnBack();
   moveToWall();
}

function moveToMiddle() {
   if (frontIsClear()) {
      moveDouble();
      moveToMiddle();
      move();
   }
   else
       turnBack();
}

function moveToWall() {
   while (frontIsClear())
```

Run Goal:

Obrázek 2.2 Aplikace Stanford Karel s řešením finální lekce – nalezení středu města[7]

Lépe je na tom Karel na doméně univerzity Stanford. [7] Jedná se o webovou aplikaci pro programování robota Karla pomocí syntaxe podobné jazyku JavaScript. Obrazovka je rozdělena na dvě části, v levé části se nachází editor kódu a v pravé části je zobrazeno Karlovo město.

Aplikace má dva režimy. V prvním je možné klasicky volně vytvářet a spouštět programy. Zajímavý je druhý režim, ve kterém se lze Karla, a programování obecně, postupně učit pomocí předpřipravených lekcí. V každé lekci je vždy popsáno vstupní město spolu s k němu očekávaným

výstupním městem. Uživatel vytvoří řešení dané lekce, spustí ho a aplikace mu ho automaticky ohodnotí.

První hned patrnou nevýhodou je nemožnost vlastní úpravy města, lze pouze vybírat z předem připraveného seznamu několika málo prázdných měst s různými rozměry a jedním předpřipravených bludištěm. Tato funkcionalita předpřipravených měst sama o sobě by ovšem byla dobrá, jelikož může uživateli ušetřit čas se stavbou vlastního bludiště či jiného druhu města, ale v případě, že by byla zároveň v kombinaci se svobodnějšími možnostmi editace.

Aplikace není plně responzivní, a na telefonu či jiném zařízení s malým displejem ji lze používat jen s velkými obtížemi. Nicméně je to možné, v uživatelské rozhraní není přítomen žádný problém, který by tomu zcela bránil.

2.3 Karel 1981

■ **Obrázek 2.3** Aplikace Karel 1981 s programem pro přenesení značky [3]

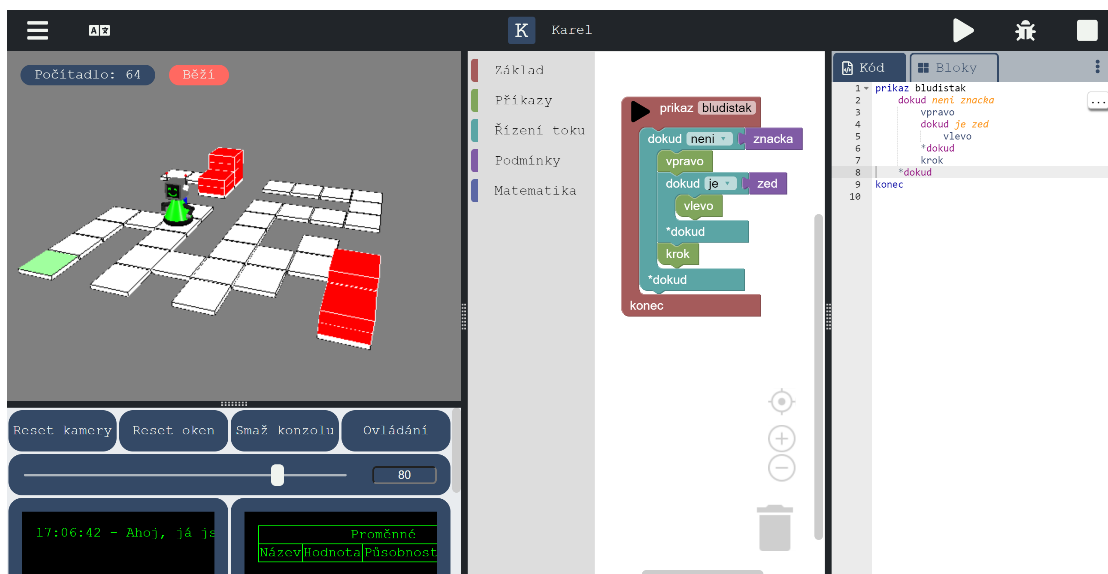
Tento Karel [3] je jako jediný z tohoto seznamu dostupný formou mobilní aplikace. Podporované jsou operační systémy Android a iOS.

V případě orientace zařízení na výšku je v horní části aplikace vidět editor města a ve spodní editor kódu. Na panelu v úplném spodku obrazovky jsou tlačítka pro spuštění a zastavení vytvořeného programu. Pro přihlášené uživatele aplikace umožňuje vytvořený kód s městem uložit na serveru. Po uložení ho lze také sdílet mezi ostatní uživatele aplikace, kteří jej poté mohou v seznamu nalézt a stáhnout do svého zařízení. V případě nedostupnosti internetového připojení se ale hodí, že aplikace bez problému funguje i offline. Je také obsažena podrobná příručka jazyka.

Co se týče možností jazyka, tak ten je z analyzovaných aplikací pravděpodobně nejblíže původní Pattisově verzi. Editor kódu umí barevně zvýraznit klíčová slova a další prvky jazyka, zvládá také automatické odsazování nového řádku. Editor města je relativně pokročilý, dají se v něm pokládat značky a zdi, přesouvat Karla a i specifikovat rozměry města. Také lze zobrazením města pohybovat v případě, že se nevejde na obrazovku.

Nevýhodou aplikace je, že je dostupná pouze na mobilní zařízení. V prohlížeči na webu lze jen zobrazovat uložená města s kódem. Dalším problémem je zvolená syntaxe jazyka používající složené závorky pro definici bloku, které je ale na klávesnici mobilního telefonu složité psát. Chybí některé vlastnosti pro pohodlnější psaní kódu, jako napovídání a podtrhávání chyb.

2.4 Karel 3D



■ **Obrázek 2.4** Aplikace Karel 3D s programem pro nalezení cesty z bludiště [8]

Nejpokročilejší z tohoto seznamu je webová aplikace Karel 3D [8] vzniklá v rámci bakalářské práce [9] Vojtěcha Čoupka. Tento Karel se ve svém pojetí od těch ostatních i nejvíce liší. Pro vykreslování města používá 3D zobrazení a kromě klasického textového programování umožňuje také skládat program graficky pomocí bloků. Vytvořený program pak i lze po jednotlivých příkazech krokovat a na jeho řádky umísťovat breakpointy. Jazyk samotný je o něco pokročilejší a to hlavně přidanou podporou proměnných.

Obrazovka editoru je tentokrát rozdělena na tři hlavní části. V levé je zobrazeno město, v té prostřední lze programovat pomocí bloků a v té pravé klasicky pomocí kódu. Vrchní lišta pak obsahuje tlačítka pro spuštění a ladění programu. V levé části pod zobrazením města jsou ještě další nástroje jako tlačítka pro manuální ovládání robota, konzole vypisující zprávy o stavu robota a tabulka aktuálních hodnot definovaných proměnných.

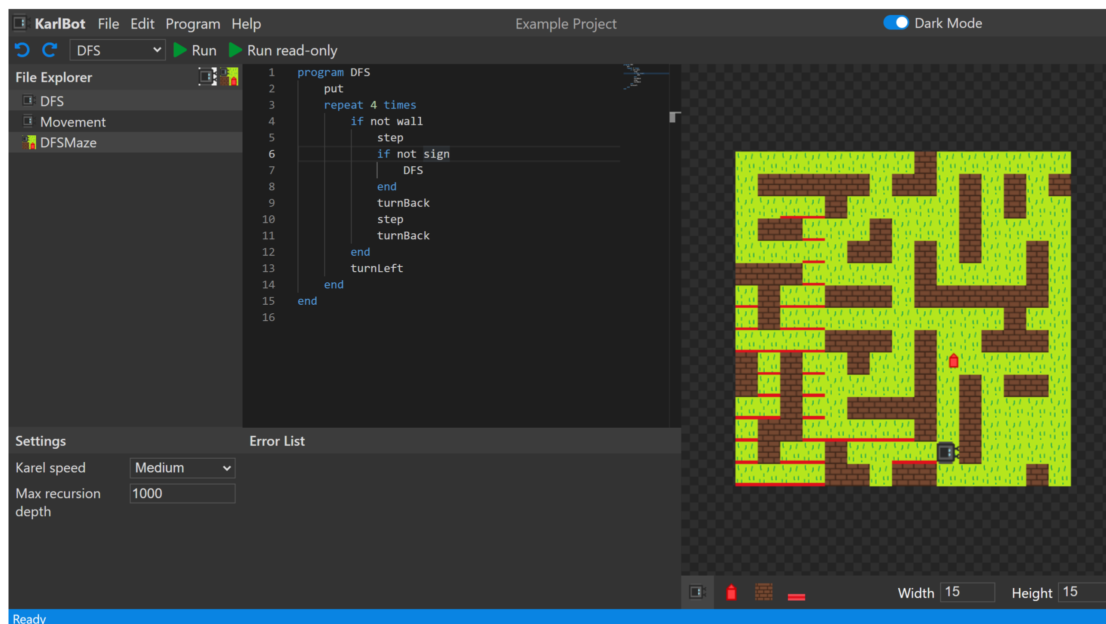
Editor kódu nabízí několik pokročilejších funkcí. Umí napovídat názvy programů a klíčových slov, samozřejmostí je pak barevné zvýrazňování a automatické odsazování. Zvládá i podtrhávání chyb, nevýhodou ale je, že ne přímo při psaní, kontrola se provádí jednorázově až před spuštěním.

Poměrně omezená je však editace města, lze sice nastavit jeho rozměry, ale samotná editace je již možná pouze manuálním ovládáním robota pomocí kláves nebo tlačítek na obrazovce. Ta je tak výrazně komplikovanější než v ostatních aplikacích, kde stačí jen klikat na dlaždice města.

Zajímavou vlastností je 3D zobrazování. Robot může před sebe na hromadu pokládat cihly a poté na ně i vystoupit. Zvládne však vystoupit vždy nejvíce o jednu cihlu výše, než aktuálně stojí. V kombinaci s tím, že cihly musí od země tvořit souvislý sloupec, se tak nejedná o 3D prostor v pravém slova smyslu, jelikož počet pozic, na kterých může být robot umístěn, závisí pouze na dvou rozměrech města.

Do určité míry je aplikace optimalizována pro mobilní zařízení. Všechny funkce aplikace, včetně ovládání pohledu na město, jsou na mobilním zařízení s dotykovou obrazovkou dostupné. Problém však je, že ovládací prvky aplikace jsou příliš malé, a je tak nutné použít zoom prohlížeče. Dále také pokud se uživateli aplikace stane, že se celou obrazovkou telefonu dostane do panelu pro zobrazení města nebo blokovou editaci, tak již není možné panel nijak opustit. Celkově tedy aplikace na telefonu použitelná není.

2.5 karlbot.cz



■ **Obrázek 2.5** Aplikace karlbot.cz s programem pro nalezení cesty z bludiště [10]

Webová aplikace karlbot.cz [10] je implementace Karla od autora této práce, na kterou zde bude navázáno. Celá aplikace se skládá z jedné stránky, jejíž hlavní části stejně jako v jiných řešeních tvoří editor kódu a editor města. Základní organizační jednotkou v aplikaci je „projekt“, který sdružuje nastavení a několik souborů s kódem a městy. To je rozdíl od ostatních řešení, kde mohlo být město a kód v editoru pouze jedno.

Editor kódu v omezené míře podporuje automatické odsazování, barevné zvýrazňování, kontrolu chyb při psaní a napovídání názvů programů. V editoru města jsou implementovány i některé pokročilejší nástroje, jako je výběr obdélníkové oblasti nebo posun a přiblížení pohledu kamery. Lze také měnit rozměry města.

Je možné kontrolovat rychlost robota a editovat město i za běhu programu. Dostupné jsou dva režimy spuštění, které se liší v tom, jestli je nebo není město po ukončení programu vráceno do původního stavu. Možnosti robota a jazyka (ne však syntaxe) do značné míry vycházejí ze zmíněného řešení Robot Karel [6] Oldřicha Jedličky.

Aplikace je zcela použitelná prakticky jen na stolním počítači. Uživatelské rozhraní není plně responzivní a neumí se přizpůsobit menším rozlišením. Některé funkcionality editoru města jsou dostupné pouze pomocí myši. Aplikace je zcela nepoužitelná na mobilních zařízeních, jednak kvůli již zmíněným problémům s responzivitou a ovládáním bez myši, ale i kvůli dalším jako jsou příliš malé ovládací prvky a editor kódu nepodporující dotykové ovládání. Malou výhodou je možnost přepínání tmavého a světlého režimu.

2.6 Shrnutí

Následuje ještě tabulka porovnávající jednotlivé funkcionality zmíněných řešení. Jejich hlavní funkcionality byli umístěny do řádků. Sloupce odpovídají jednotlivým řešením a v jejich průniku je potom informace, zda je funkcionalita v řešení obsažena či nikoliv. Na testování optimalizace webových aplikací pro telefony byl použit nástroj [11] od společnosti Google.

	Robot Karel	Stanford Karel	Karel 1981	Karel 3D	karlbot.cz
Zvýrazňování	NE	ANO	ANO	ANO	ANO
Napovídání	NE	NE	NE	ANO	ANO
Automatické odsazování	NE	ANO	ANO	ANO	?
Kontrola chyb při psaní	NE	NE	NE	NE	?
Editace pomocí bloků	NE	NE	NE	ANO	NE
Proměnné	NE	NE	NE	ANO	NE
Krokování	NE	NE	ANO	ANO	NE
Breakpointy	NE	NE	NE	ANO	NE
Editace města	ANO	NE	ANO	?	ANO
Ukládání lokálně	?	NE	ANO	ANO	ANO
Ukládání na server	NE	NE	ANO	ANO	NE
Sdílení	NE	?	ANO	NE	NE
Automaticky hodnocené úkoly	NE	ANO	NE	NE	NE
Optimalizováno pro počítač	ANO	ANO	NE	ANO	ANO
Optimalizováno pro telefon	NE	NE	ANO	NE	NE

Tabulka 2.1 Porovnání funkcionalit existujících řešení

V některých případech se stalo, že dané funkcionality šlo dosáhnout, nicméně nestandardním nebo mnohem složitějším způsobem než v ostatních aplikacích. V takovém případě je u funkcionality uveden otazník a pro posouzení čtenáře je diskutována v následujícím odstavci.

Ukládání v aplikaci Robot Karel je implementováno, ale město a kód je nutné ukládat zvlášť, a to pouze manuálním zkopírováním zobrazeného textu do schránky. V aplikaci Stanford Karel je sice zobrazeno tlačítko pro sdílení, avšak v době přístupu na web bylo nefunkční. Editace města v řešení Karel 3D bylo možné dosáhnout bez spuštění programu, ale pouze ručním vykonáváním příkazů robota pomocí tlačítek nebo kláves. To bylo tak výrazně náročnější a zdlouhavější než v ostatních aplikacích pokládání zdí a značek pouhým klikáním na dlaždice města. Aplikace karlbot.cz zvládá automatické odsazování pouze na základě odsazení předchozího řádku a ne skutečné úrovně zanoření v daném místě. Dále také při kontrole chyb tato aplikace v určitých případech nezobrazuje chybový rozsah správně.

2.6.1 Výsledek

Jak je nejlépe vidět v předchozí tabulce, tak neexistuje žádné řešení, které by obsahovalo všechny funkce ostatních. Tak základní funkcionalita, jako je editace města, je v plné míře obsažena pouze ve 3 z celkových 5. Žádné také není možné používat zároveň na počítači i mobilním telefonu.

Celkově tak neexistuje žádné řešení, které by na jednom místě umožňovalo zároveň pohodlnou editaci kódu s pokročilejšími nástroji, volnou editaci města, následné spuštění programu včetně jeho ladění, jednoduché ukládání na serveru, sdílení programů a plnění automaticky hodnocených úkolů. To dává prostor vzniknout novému řešení.

Kapitola 3
Analýza řešení karlbot.cz

Tato kapitola se zabývá analýzou starého řešení [10], ze kterého bude to nové, vytvořené v rámci této práce, vycházet. Funkčnosti aplikace z uživatelského pohledu byly již rozebrány v kapitole o existujících řešení, zde budou popsány ještě některé nezmíněné detaily a hlavně implementace z pohledu vývojáře. Na konci budou shrnuty nalezené nedostatky, které by měly být v novém řešení odstraněny. Autor této práce je zároveň i autorem tohoto řešení, a tak tato kapitola z velké části vychází z jeho znalosti o implementaci.

3.1 Varianta jazyka Karel

```
program DFS
    put
    repeat 4 times
        if not wall
            step
            if not sign
                DFS
            end
            turnBack
            step
            turnBack
        end
        turnLeft
    end
end
```

■ **Výpis kódu 3.1** Implementace DFS algoritmu ve variantě Karla karlbot.cz [10]

Implementovaná varianta jazyka Karel podporuje definici uživatelských programů, podmínky s nepovinnou větví else, cyklus s podmínkou a cyklus s předem daným počtem opakování. Obsahuje také jednořádkové a víceřádkové komentáře. Nepodporuje například proměnné, aritmetické výrazy či definici programů přijímajících parametry nebo vracejících hodnoty. Nevynucuje odsazení, ani psaní každého příkazu na samostatný řádek. Pro ulehčení psaní kódu její syntaxe neobsahuje žádné speciální znaky.

Na výpisu kódu 3.1 je ukázána implementace rekurzivního DFS algoritmu v této variantě jazyka. V ukázce byla vynechána implementace programu `turnBack`.

3.1.1 Programy pro interakci s městem

V rámci aplikace jsou do jazyka vestavěny následující programy pro interakci s městem:

- `step` – Posune Karla o jednu dlaždici ve směru, ve kterém je natočený.
- `turnLeft` – Otočí Karla o 90 stupňů doleva.
- `put` – Položí značku na dlaždici, kde se Karel nachází.
- `pick` – Zvedne značku z dlaždice, kde se Karel nachází.

3.1.2 Programy pro testování okolí

Dále také tyto programy pro testování okolí Karla v podmínkách a podmíněných cyklech:

- `north` – Otestuje, zda je Karel otočený na sever města.
- `east` – Otestuje, zda je Karel otočený na východ města.
- `west` – Otestuje, zda je Karel otočený na západ města.
- `south` – Otestuje, zda je Karel otočený na jih města.
- `sign` – Otestuje, zda se na dlaždici, kde Karel stojí, nachází značka.
- `wall` – Otestuje, zda se na dlaždici před Karlem nachází zeď.
- `home` – Otestuje, zda se na dlaždici, kde Karel stojí, nachází domov.

3.2 Neopravené chyby

Kromě problémů s použitelností má aplikace také několik chyb, které jsou i při běžném používání hned patrné. První tři zásadně komplikují práci s aplikací, čtvrtá je méně významná.

Zamrznutí při zacyklení Pokud uživatel udělá chybu a vytvoří program, který nikdy skončí, tak v aplikace v určitých případech celá zamrzne a je nutné zavřít její kartu v prohlížeči.

Větší množství chybových zpráv rozbijí rozložení stránky V případě, že program obsahuje více chyb, než se vejde do panelu s jejich seznamem, tak místo toho, aby byl v panelu zobrazen posuvník, je panel celý rozšířen. S ním pak i celá stránka a posuvník se zobrazí až u ní. To v důsledku rozbije i zobrazení města.

Neomezený pohyb zobrazením města Editor města nemá žádný limit na posun kamery a úroveň jejího přiblížení. Je tak snadné město ze zobrazení úplně ztratit a prakticky nemožné ho dostat zpět.

Špatný rozsah syntaktických chyb V některých případech není rozsah syntaktické chyby ve zdrojovém kódu zobrazen korektně. Ukazuje se větší, než by měl být.

3.3 Použité technologie a struktura projektu

Celý projekt je implementován v jazyku TypeScript [12] s pomocí frameworku Angular [13]. Aplikace se skládá ze dvou hlavních částí, implementace jazyka Karel a jádra aplikace s editorem. Obě tyto části jsou umístěny v jednom Angular projektu, jehož adresářová struktura je popsána na obrázku 3.1. Uživatelské rozhraní je vytvořeno jako single page webová aplikace. Pro ikony je použita knihovna Font Awesome [14]. Editor kódu využívá knihovnu Monaco Editor [15].

```
│ assembly.............................Instrukce bajtkódu jazyka Karel
├─ compiler.......................Starý již nepoužívaný kompilátor jazyka Karel
├─ compiler2...............................Nový kompilátor jazyka Karel
├─ components.....................................Sdílené komponenty
├─ core................................................Jádro aplikace
├─ directives......................................Sdílené direktivy
├─ interpreter.........................Interpret bajtkódu jazyka Karel
├─ pages......................................................Stránky
├─ services....................................................Služby
└─ standard-library...............Programy standardní knihovny jazyka Karel
```

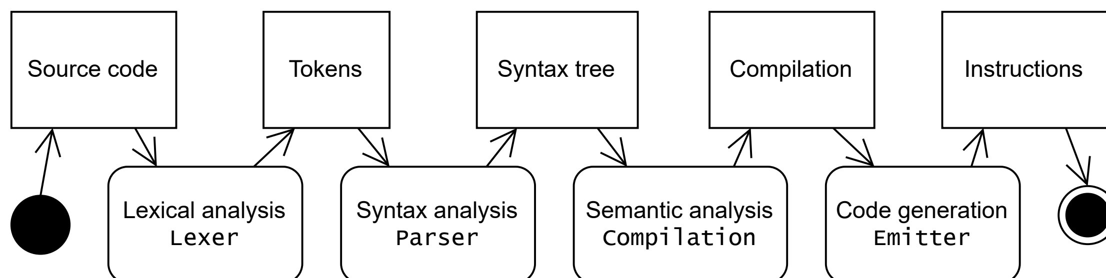
■ **Obrázek 3.1** Adresářová struktura řešení karlbot.cz

3.4 Implementace jazyka Karel

Aplikace obsahuje vlastní kompilátor a interpret jazyka Karel. Pro snazší interpretaci kódu je před spuštěním zdrojový kód jazyka kompilátorem převeden na posloupnost jednoduše vykonatelných instrukcí, které jsou poté už přímo vykonávány interpretem jazyka. Instrukce mají formu programu pro zásobníkový virtuální stroj [16], to znamená, že si mezivýsledky ukládají na sdílený zásobník. Těmto instrukcím se také říká bajtkód.

3.4.1 Fáze kompilace

Proces kompilace je podobný jako v kompilátorech běžných jazyků [17], a má tedy několik fází. V této kapitole bude popsán pouze aktuálně používaný kompilátor ve složce `compiler2`. Jeho fáze kompilace znázorňuje diagram na obrázku 3.2 a dále jsou pak popsány v textu.



■ **Obrázek 3.2** Fáze kompilace

Lexikální analýza V první fázi je kvůli zjednodušení dalšího zpracování zdrojový kód rozdělen na posloupnost tzv. tokenů. Token je například klíčové slovo, název programu nebo číslo.

Je použit ručně napsaný lexikální analyzátor implementovaný ve třídě `Lexer`. Jeho základem je metoda přijímající na svém vstupu stream znaků a vracející první nalezený token. Výhoda přijímání streamu a vracení pouze prvního tokenu je zejména to, že je lexer tak možné použít i ve scénáři inkrementálního parsování, kdy není žádoucí zpracovat celý vstup.

Kvůli předpokladu použití kompilátoru v editoru není možné přeskakovat komentáře a bílé znaky, ale je nutné je zachovat, aby byl celý syntaktický strom i po úpravě kompletně převoditelný zpět na zdrojový kód s původním formátováním. K tomu je použita technika [18] z kompilátoru Roslyn [19], kde se těmto pro vykonávání nepodstatným částem syntaxe říká „syntax trivia“ a přidávají se k jednotlivým tokenům. Každý token tak v implementaci obsahuje vlastnosti `leadingTrivia` a `trailingTrivia`, ve kterých jsou tyto části syntaxe uloženy.

Syntaktická analýza Ve druhé fázi je z posloupnosti tokenů vytvořen tzv. syntaktický strom reprezentující strukturu zdrojového kódu.

To je úkolem parseru umístěného ve třídě `Parser`. Jeho základ tentokrát tvoří metoda přijímající stream tokenů a vracející syntaktický strom. Používá techniku rekurzivního sestupu [20], parser se tak skládá z několika mezi sebou se volajících rekurzivních metod. Každá metoda odpovídá jednomu typu uzlu syntaktického stromu (neterminálu gramatiky jazyka) a volá metody pro parsování typů uzlů, které mohou být jeho dětmi.

Editory musejí parsovat kód v průběhu jeho psaní, a je tak nutné, aby byl parser rychlý. Často se proto využívání technika inkrementálního parsování, kdy jsou nezměněné části kódu při parsování přeskočeny a jim odpovídající uzly původního syntaktického stromu jsou znovupoužity v novém syntaktickém stromu. [21] Původní kompilátor ze složky `compiler` obsahoval inkrementální parser, nicméně vzhledem k tomu, že rozsah běžných programů v Karlovi to nevyžaduje a tento kompilátor tím byl zbytečně komplikovaný, tak do nové verze ve složce `compiler2` nebyl zahrnut.

Sémantická analýza Třetí fáze spočívá ve zpracování syntaktického stromu a vytvoření tabulky symbolů. Symbol je v jazyku Karel pouze program, v jiných jazycích by to však mohly být například i proměnné nebo třídy. Dále se zde také provádí kontrola chyb.

První dvě fáze se prováděly nad každým souborem odděleně, při sémantické analýze je však již třeba znát program jako celek. K zabalení všech syntaktických stromů dohromady se používá třída `Compilation`. Tato třída už obsahuje také informace o deklaracích (nikoli implementacích) externích programů, tedy například těch ze standardní knihovny Karla, jako je krok nebo otočení vlevo.

Implementace sémantické analýzy je umístěna přímo ve třídě `Compilation`. Spočívá hlavně ve vytvoření tabulky symbolů. O kontrolu chyb se pak stará třída `Checker`. Používá k tomu právě instanci třídy `Compilation` a vytvořené symboly.

Generování kódu Ve čtvrté fázi je ze syntaktického stromu a tabulky symbolů vytvořena posloupnost instrukcí bajktódu jazyka Karel.

To dělá třída `Emitter` přijímající opět instanci třídy `Compilation` s vytvořenými symboly a vracející vygenerované instrukce zabalené v instanci třídy `Assembly`. Pro každý program jsou instrukce generovány zvlášť, a `Assembly` tak obsahuje pole instancí třídy `Program`, které pak už obsahují přímo pole instrukcí.

3.4.2 Immutabilní syntaktický strom

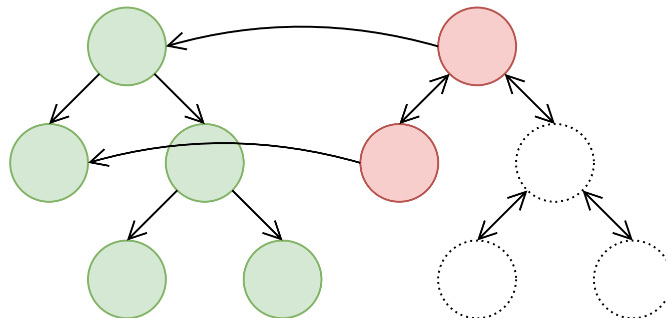
Až na pár výjimek jsou všechny třídy kompilátoru immutabilní. To znamená, že objekt takovéto třídy lze pouze číst a pro „změnu“ hodnot jeho vlastností je nutné vytvořit ho celý znovu s jejich novými hodnotami.

Výhod tohoto přístupu je hned několik. V první řadě je významně zjednodušena detekce změn v objektu z vnějšku – žádná není potřeba, jelikož se objekt změnit nemůže. Dále je mnohem snazší udržet takový objekt ve validním stavu, protože jediné místo, kde je potřeba to kontrolovat je konstruktor. Další výhoda, ovšem v jiných jazycích, je snadnost použití takových objektů při vícevláknovém programování, kdy z důvodu neměnnosti objektu není nutné k němu řídit přistup více vláken. Není to ani tak neefektivní, jak by se mohlo zdát, jelikož není potřeba provádět hlubokou kopii, ale pouze mělkou. Určitá režie navíc ale v tomto kopírování samozřejmě je, a bylo tedy potřeba uvážit, jak často se data budou měnit.

Přináší to však své vlastní výzvy. Editory zdrojového kódu a různé nástroje pro jeho analýzu potřebují procházet syntaktický strom a upravovat ho. Kvůli tomu se hodí mít u každého uzlu referenci zároveň na jeho děti i rodiče. Problém však je, že uzly immutabilního syntaktického stromu mohou obsahovat reference na své děti, ale již ne zároveň na svého rodiče. Pokud by je

totiž obsahovali, bylo by nutné při každé změně vlastnosti v uzlu vytvořit celý syntaktický strom znovu. Naproti tomu, pokud obsahují reference jen na své děti, tak při změně jednoho uzlu stačí přebudovat pouze všechny uzly na cestě do kořene. Jelikož syntaktické stromy nebývají příliš hluboké, tak se ani nejedná o zvláště nákladnou operaci.

Jako řešení byla v této implementaci použita myšlenka [22] z kompilátoru Roslyn. V něm řeší problém tak, že mají dva syntaktické stromy, červený a zelený. Zelený je klasický immutabilní strom vytvářený odspoda nahoru a jeho uzly nemají žádnou referenci na svého rodiče, pouze na své děti. Červený strom slouží jako obal pro ten zelený a je vytvářený *lazy* způsobem (tzn. uzly jsou vytvořeny až v případě potřeby) ze shora dolů. Uzly tohoto stromu již obsahují referenci i svého na rodiče, ale přesto je navenek immutabilní a musí tak být vytvářen po každé změně znovu. Díky tomu, že je vytvářený lazy a předpokladu, že ve většině případů není potřeba strom projít celý, je ale vždy takto vytvořena pouze jeho malá část. Celé to ilustruje obrázek 3.3.



■ **Obrázek 3.3** Zelený a č strom – myšlenka z kompilátoru Roslyn [22]

S červenými a zelenými stromy souvisí ještě pozice uzlů syntaktického stromu ve zdrojovém kódu a jejich textová délka. Ty je potřeba znát například pro kontrolu syntaktických chyb. Pokud by ovšem u uzlů byla ukládána jejich pozice, tak po změně kteréhokoliv uzlu by musela být u všech uzlů, které jsou ve zdrojovém kódu za ním, tato pozice přepočtena, a ze změny jednoho uzlu by se tak stala velmi nákladná operace. Řešením je zde tyto pozice vůbec neukládat a u každého uzlu zeleného stromu si pamatovat jen jeho délku a pár dalších vlastností jako počet nových řádek a délku té poslední v něm obsažené. Jelikož jsou uzly immutabilní, tak tyto informace lze spočítat z informací jeho dětí jednorázově při vytvoření uzlu. Pozice uzlů jsou pak dostupné pouze v uzlech lazy vytvářeného červeného stromu, kde jsou počítány z pozice rodiče a délky sourozenců, kteří uzlu předchází.

3.4.3 Interpret

```
 0: Push            value: 0
 1: Store           variableIndex: 0
 2: Push            value: 4
 3: Load            variableIndex: 0
 4: CompareGreater
 5: JumpIfFalse     instructionIndex: 12
 6: CallExternal    name: "step"
 7: Push            value: 1
 8: Load            variableIndex: 0
 9: Add
10: Store           variableIndex: 0
11: Jump            instructionIndex: 2
```

■ **Výpis kódu 3.2** Instrukce bajtkódu do kterých se přeloží cyklus s předem daným počtem opakování

Interpret se stará o vykonávání instrukcí bajtkódu a spouštění z nich volaných externích programů. Příklad takových instrukcí je na výpisu kódu 3.2 na předchozí stránce. Jeho implementace se nachází ve třídě `Interpreter`. Pro jeho použití se mu předají instrukce a implementace externích programů a následně se zavolá metoda `interpretAll`, která všechny instrukce vykoná. Tato metoda je asynchronní a vrátí výsledek až po skončení celé interpretace. Externí programy se předávají jako funkce, které mohou být rovněž asynchronní. Díky tomu je například možné pomocí časovače simulovat práci robota u programů, jako je krok nebo položení značky.

Interpretaci lze v jejím průběhu ukončit. Používá se kooperativní způsob ukončení. K tomu se metodě `interpretAll` a z té poté spouštěným externím programům předává instance třídy `InterpretStopToken`. Tato třída obsahuje metodu pro požádání o ukončení a zároveň také událost vyvolanou po tomto požadavku. Interpret nebo externí programy se pak k této události zaregistrují a ve chvíli požadavku na ukončení prováděnou akci co nejdříve zruší a vrátí kontrolu do volajícího.

Z pohledu implementace je v interpretu pro každou instrukci vlastní metoda, která ji vykonává. Metoda `interpretAll` ve svém základu pak pouze deleguje vykonávání na správnou metodu podle aktuální instrukce. Volání programů a vrácení se z volaného programu na správné místo je řešeno simulací zásobníku volání pomocí pole v interpretu. V každém rámci zásobníku jsou informace o indexu další instrukce k vykonání a hodnotách definovaných proměnných (jazyk Karel je sice neobsahuje, ale jeho bajtkód ano).

3.4.4 Ukázka použití

Pro lepší představu je na výpisu 3.3 ukázáno použití kompilátoru a interpretu v kódu. Kód zkompiluje program do bajtkódu a pomocí interpretu ho spustí nad vytvořeným prázdným městem.

```
const sourceCode = ...;
const extProgramRefs = getStandardLibraryProgramReferences();

const compilationUnit = Parser2.parse(sourceCode, "Some file name");
const compilation = new Compilation([compilationUnit], extProgramRefs);

const errors = Checker.check(compilation);
// ...Handle errors

const assembly = Emitter.emit(compilation);
const entryPoint = assembly.programs.find(p => p.name === "main");

const interpreter = new Interpreter();
interpreter.callStack.push(new CallStackFrame(entryPoint));

const town = Town.createEmpty(10, 10);
for (const extProgram of getStandardLibraryPrograms(town, () => 0))
    interpreter.registerExternalProgram(extProgram);

await interpreter.interpretAll(new InterpretStopToken());
```

■ **Výpis kódu 3.3** Ukázka použití knihovny jazyka Karel ve starém řešení

3.5 Implementace aplikace

Uživatelské rozhraní není příliš rozsáhlé. Jedinou veřejnou stránkou aplikace je editor projektu. Pro stránky je určena složka `pages`. Kromě editoru se v ní nachází ještě dvě testovací stránky

používané pouze při vývoji. Stránky jsou klasické Angular komponenty. Jejich načtení na základě aktuální URL adresy zajišťuje router frameworku. Uživatelské rozhraní stránky se skládá z několika dalších Angular komponent a direktiv.

Ve složce se stránkou jsou umístěny komponenty a direktivy, které jsou používané pouze na ní. Ostatní, u nichž je předpoklad použití na více stránkách, se nacházejí ve složkách `components` a `directives`. Většina komponent nemá žádný svůj stav a pouze zobrazují data předané do jejich vlastností z komponent nad nimi. Uživatelské akce zpět komunikují pomocí událostí.

Stav je umístěn přímo v komponentě stránky. Třídy, které ho tvoří, se nacházejí ve složce `core`. Podobně jako v kompilátoru jsou všechny s výjimkou třídy `Town` immutabilní. Zbylá logika je implementována ve službách ve složce `services`. Jsou v ní služby pro načítání souborů, zobrazování dialogů a správu barevného tématu.

3.5.1 Editor

Veškerá logika editoru se nachází přímo v kódu komponenty jeho stránky. Po každém napsaném znaku je znovu přeparsován kód v aktuálně otevřeném souboru. Vzniklý syntaktický strom editor poté přes rozhraní kompilátoru používá pro funkce, jako je napovídání nebo kontrola chyb.

Jak již bylo zmíněno, pro samotnou komponentu editoru kódu byla použita knihovna Monaco Editor. Jedná se o komponentu editoru kódu od společnosti Microsoft, který ji mimo jiné používá [15] ve svém editoru Visual Studio Code [23]. Tato knihovna také přichází se svým řešením pro tokenizaci kódu pomocí regulárních výrazů. Na základě vytvořených tokenů poté v editoru barevně zvýrazňuje kód.

Dále Monaco Editor poskytuje podporu pro integraci napovídání a podtrhávání chyb. Pro napovídání programátor vytvoří funkci, která na základě předané pozice v kódu vrátí napovídané položky. Editor funkci po některých napsaných znacích, či přímo po uživatelově vyžádání zavolá a vrácené položky zobrazí v popup seznamu. Mezi některé další funkcionality knihovny patří ještě například automatické odsazování nového řádku na základě toho předchozího nebo možnost zavírání jednotlivých bloků kódu.

Zobrazování města bylo implementováno pomocí nativního HTML elementu `canvas`. Element poskytuje JavaScriptové rozhraní, pomocí kterého lze na jeho plochu vykreslovat různé grafické prvky, jako třeba obrázky, obdélníky nebo úsečky. Samotné vykreslení města je poměrně jednoduché, je pouze nutné správně vypočítat pozici a velikost jednotlivých dlaždic města po aplikaci aktuálního posunu a přiblížení kamery. Kvůli optimalizaci v případě velkých měst jsou vykreslovány pouze viditelné dlaždice.

3.6 Shrnutí nalezených nedostatků

Neoddělení prezentace a logiky Tento princip je porušen na mnoha místech v kódu. Komponenta stránky s editorem se nestará pouze o své rozložení a zobrazení, ale je zde implementována i prakticky celá logika editoru. Ten samý problém se týká komponenty editoru města, kde je zase umístěna i logika pro editaci.

Neoddělení implementace jazyka Karel od zbytku aplikace Přesto, že jednotlivé třídy implementace jazyka Karel na zbytku aplikace nezávisí, tak jsou umístěny ve stejné adresářové struktuře. Mohlo by se tak snadno stát, že někdo do nich takovou závislost vnese a tím zkomplikuje použití jazyka bez zbytku aplikace.

Absence automatických testů V aplikaci nejsou přítomny žádné automatické testy. Je tak nutné po každé změně manuálně ověřit, že celá aplikace funguje jak má.

Absence dokumentace Dalším nedostatkem jsou chybějící dokumentační komentáře, minimálně u veřejných vlastností a metod v kódu. Dokumentace popisující jak a také proč některé

věci fungují tak jak fungují by mohla ušetřit mnoho času, který by jinak musel být vynaložen na získání těchto informací přímo z kódu. Není přítomna ani dokumentace pro spuštění a používání aplikace.

Nedokonalá immutabilita některých tříd Přesto, že je stav aplikace navržen jako immutabilní, tak zde tento přístup není důsledně dodržován. Například jinak immutabilní třída `Project` v sobě obsahuje třídu `File` a třída `TownFile`, která ze třídy `File` dědí, obsahuje třídu `Town`, která ale immutabilní už není. To znamená, že v konečném důsledku ani třída `Project` není plně immutabilní a neplatí pro ni tak v celém rozsahu dříve zmíněné výhody.

Vypnutý striktní režim kompilátoru Kompilátor TypeScriptu není nastaven ve striktním režimu. V tomto režimu kompilátor díky více informacím, které musí programátor přímo specifikovat, může odhalit více chyb, které by se jinak projevili až za běhu aplikace. Například je v tomto režimu povinné u typů, které mají smět obsahovat hodnotu null, to explicitně uvést. Na základě toho pak kompilátor třeba hlídá, jestli není volána metoda na hodnotě null. [24]

Zbytečné vlastní obecné UI komponenty Aplikace používá vlastní obecné UI komponenty například pro různá menu a dialogy. Ty je ale nutné ručně vytvořit, otestovat, zdokumentovat a dále udržovat. To jen přináší prostor pro chyby a zabírá čas, který by mohl být využit pro vývoj funkcí, které jsou specifičtější této aplikaci. Bylo by tak vhodné použít nějakou komponentovou knihovnu. Ty však často přichází se svým vlastním designem, který lze přizpůsobit jen do určité míry, a jejich použití tak nemusí být možné, pokud aplikace vyžaduje unikátní design. V případě této aplikace to je ale naopak výhoda, jelikož unikátní design vyžadován není a již vytvořený tak jen ulehčí práci.

Zbytečné vykreslování Město je vykreslováno při každém překreslení prohlížeče, i když se v něm od posledního vykreslení nic nezměnilo. To působí znatelnou zátěž na výkon aplikace a v důsledku i negativně ovlivňuje výdrž baterie v případě přenosných zařízení.

Nefunkčnost editoru města bez myši Většina funkcí editoru města je dostupných pouze pomocí myši, což dělá editor nepoužitelným na mobilních zařízeních. Kvůli nutnosti držet kolečko myši pro pohyb kamery není ovládání editoru plně kompatibilní ani s touchpady většiny notebooků.

Nefunkčnost editoru kódu na mobilních zařízeních Knihovna Monaco Editor využitá pro editor kódu nepodporuje mobilní zařízení.

Nepřizpůsobení menším rozlišením obrazovek Aplikace se neumí přizpůsobit menším obrazovkám jako mají například mobilní telefony. V kombinaci se dvěma předchozími body je tak na mobilních zařízeních zcela nepoužitelná.

Neopravené chyby Jak již bylo uvedeno v podkapitole 3.2, tak v aplikaci je neopraveno několik závažných chyb, které komplikují její reálné použití.

Kapitola 4
Analýza požadavků

Ještě než začne návrh nového řešení, je třeba vědět, jaká má splňovat kritéria. K tomu slouží analýza požadavků provedená v této kapitole. Požadavky jsou rozděleny do dvou kategorií. Funkční požadavky určují jednotlivé funkcionality a nefunkční požadavky ostatní hlediska, jako například vzhled nebo použitelnost. Specifikovány byly z problémů existujících řešení a ze zadání práce.

4.1 Funkční požadavky

F1: Editace zdrojového kódu Základní funkcionalitou bude editace zdrojového kódu.

Pro ulehčení psaní a orientace v kódu jsou očekávány následující vlastnosti editoru:

1. Automatické odsazování nového řádku na základě aktuální úrovně zanoření.
2. Barevné zvýrazňování klíčových slov a dalších částí kódu.
3. Napovídání názvů vestavěných i uživatelem definovaných programů.
4. Podtrhávání syntaktických chyb při psaní kódu.

F2: Editace města Bude možné vytvářet a upravovat město.

Editor musí podporovat následující možnosti úpravy:

1. Změna pozice Karla a jeho domova.
2. Umisťování a odstraňování zdí.
3. Přidávání a odebírání značek.
4. Změna rozměrů města.
5. Posun a přiblížení pohledu.
6. Obdélníkový výběr pro rychlejší umisťování zdí a značek.

F3: Spouštění programů Přímo v editoru půjde nad vytvořeným městem spustit uživatelem zvolený program. Půjde nastavit rychlost robota a to i za běhu programu. Dále také bude možné spustit program i v režimu se zálohováním města, které se v tom případě po skončení programu obnoví do původního stavu.

F4: Ladění programů Spuštěný program bude možné pozastavit. V pozastaveném stavu půjde program po jednotlivých příkazech krokovat. Dále na jednotlivé řádky zdrojového kódu bude možné umísťovat breakpointy indikující, že se program má před vykonáním příkazu na takto označené řádce pozastavit. V pozastaveném stavu bude také zobrazován aktuální stav zásobníku volání.

F5: Ukládání projektů do zařízení Celý projekt půjde jako jeden soubor uložit do zařízení uživatele a také z něho načíst.

F6: Ukládání projektů na serveru Jelikož ukládání projektů do zařízení není příliš pohodlné, tak bude umožněno i ukládání na server. To bude sloužit jako primární způsob ukládání.

F7: Správa projektů uložených na serveru Každý uživatel aplikace po přihlášení uvidí své uložené projekty. Bude mít také možnost uložený projekt odstranit.

F8: Sdílení uložených projektů Na serveru uložené projekty bude možné sdílet mezi ostatní uživatele aplikace, nehledě na to, jestli přihlášené nebo nepřihlášené.

F9: Vytváření výzev a definice pravidel splnění Administrátor aplikace bude moci vytvářet a upravovat výzvy, což jsou úkoly v jazyku Karel. U výzvy půjde specifikovat její název, popis, obtížnost a pravidla splnění. V popisu budou dostupné i možnosti formátování jako jsou nadpisy, seznamy a tučné písmo. Možnosti pravidel splnění musí být alespoň takové, aby pomocí nich bylo možné vyjádřit následující referenční výzvy:

1. Karel musí dojít k nejbližší zdi ve směru jeho otočení.
2. Karel musí dojít do středu města.
3. Karel musí udělat rámeček kolem města. To znamená položit 8 značek na každou dlaždici sousedící s jeho okrajem.
4. Karel musí dojít na dlaždici označenou jako domov.

Vždy pro více různých konfigurací města, kde je to splnitelné.

F11: Zobrazení výzev a možnost odevzdání řešení Uživatel v seznamu uvidí všechny dostupné výzvy, bude si moci zobrazit její zadání a následně odevzdat své řešení.

F10: Automatické hodnocení výzev Po odevzdání řešení výzvy ho systém neprodleně automaticky ohodnotí a zobrazí uživateli výsledek.

4.2 Nefunkční požadavky

N1: Webová aplikace Je očekáváno, že pro spuštění aplikace bude stačit webový prohlížeč a nebude nutné stahovat žádný další software.

N2: Responzivní design Aplikace musí dobře fungovat na různých rozlišeních obrazovky. Počítá se s minimální šířkou 300 pixelů a minimální výškou 500 pixelů.

N3: Podpora mobilních telefonů Aplikaci musí být možné plně používat i na mobilních telefonech a tabletech. Je tedy důležité, aby správně fungovala i s dotykovým displejem orientovaným na výšku.

N4: Tmavý a světlý režim Musí být možné přepínat barvy uživatelského rozhraní mezi světlým a tmavým režimem.

N5: Syntaxe jazyka Karel z karlbot.cz Syntaxe jazyka Karel zůstane stejná jako v původním řešení karlbot.cz.

Kapitola 5

Návrh

Nejprve bude zvolena celková architektura aplikace. Poté budou po rešerši několika alternativ vybrány vhodné technologie pro pozdější implementaci. Dále bude proveden návrh přihlašování uživatelů, hodnocení výzev, databáze a komunikačního rozhraní mezi klientem a serverem. Poslední část se bude zabývat návrhem obrazovek aplikace a uživatelského rozhraní.

5.1 Architektura aplikace

Architektura je omezena požadavkem, že se musí jednat o webovou aplikaci. Nabízela se tak a byla také zvolena klasická tříúrovňová architektura [25]. Podle té je aplikace fyzicky rozdělena do níže uvedených tří úrovní. Závislosti mezi úrovněmi jsou jednosměrné, v následujícím seznamu směrem dolů a pouze mezi těmi sousedními.

Prezentační úroveň Poskytuje data koncovému uživateli aplikace. Nejčastěji formou grafického uživatelského rozhraní. Může se tak jednat například o webovou stránku nebo desktopovou či mobilní aplikaci.

Aplikační úroveň Obsahuje hlavní logiku. Příkladem může být serverová část webové aplikace.

Datová úroveň Zajišťuje uchovávání a správu dat. V praxi je typicky implementována pomocí databáze, může se ale jednat i o nějaké vzdálené rozhraní jiné služby.

Je důležité nezaměňovat pojem úroveň s pojmem vrstva. Jednotlivé úrovně bývají umístěny na navzájem fyzicky odlišné infrastruktuře. Vrstvy se týkají pouze logického členění a mohou se všechny fyzicky nacházet na stejném místě. [25]

Obrázek 5.1 Realizace tříúrovňové architektury v této aplikaci

Jak ukazuje obrázek 5.1, tak v případě této aplikace bude prezentační úroveň reprezentovat webová single page aplikace (SPA) komunikující pomocí rozhraní se serverovou částí na aplikační úrovni, která bude dále komunikovat s databází na datové úrovni. Jedná se pouze o celkovou architekturu aplikace a architektura jednotlivých podčástí pak už bude zvolena v rámci jejich implementace.

Alternativou u složitějších aplikací by mohla být například architektura mikroslužeb [26]. V té se aplikace skládá z několika malých a autonomních služeb zodpovědných vždy za určitou část domény aplikace. Každá služba je implementována pomocí vlastních technologií nezávisle na těch ostatních a běží ve svém vlastním prostředí. Mezi sebou pak komunikují pomocí nějakého společného rozhraní. Z důvodu přílišné komplexity a nevyužití výhod pro aplikaci tohoto rozsahu byla tato architektura zamítnuta.

5.2 Výběr technologie pro klienta

Dalším krokem je volba technologie pro tvorbu uživatelského rozhraní. Vybranou technologií bude také do jisté míry určen i programovací jazyk. Kvůli požadavku na implementaci klienta jako webové aplikace jsou zde porovnávány pouze technologie pro webový frontend.

Informace o popularitě jednotlivých technologií byla v této i dalších kapitolách čerpána z průzkumu [27] platformy StackOverflow a počtu otázek [28] ohledně těchto technologií na stejné platformě.

5.2.1 React

První uvažovanou knihovnou je React [29]. Uživatelské rozhraní v této knihovně je tvořeno pomocí komponent, jejichž rozložení a vzhled však na rozdíl od běžných knihoven nejsou definovány pomocí kombinace jazyků HTML a CSS, ale přímo v kódu s využitím speciální syntaxe JSX [30] připomínající syntaxi jazyka HTML. Komponenta samotná je JavaScriptová funkce, jejíž parametry tvoří parametry komponenty a která vrací elementy výsledného uživatelského rozhraní. React se poté stará o detekci změn v těchto komponentách a v případě změny o volání jejich funkcí s novými parametry.

Nabízí se otázka, jak si komponenty drží stav, když se jedná jen o funkce. Základní možností, kterou React poskytuje je hook `useState`. Hooky jsou funkce volané na začátku těla funkce komponenty umožňující využívat různé vlastnosti, které React nabízí. Hook `useState` je právě schopný držet stav mezi jednotlivými voláními funkce komponenty. Vrací pole dvou hodnot, první z nich je aktuální stav a druhou je funkce jejímž zavoláním lze tento stav změnit.

Obecný způsob práce se stavem, který React používá, je one-way data flow. To je princip, při kterém jsou v komponentovém stromu data posílána pouze směrem dolů přes parametry komponent a uživatelské akce jsou naopak propagovány nahoru do stromu pomocí událostí. Komponenta samotná tak nemůže přímo změnit hodnoty svých parametrů, ale musí vyvolat událost, na základě které se některá z komponent výše ve stromu může rozhodnout změnit data, na kterých hodnoty těchto parametrů závisí. React poté zařídí správné předání těchto nových dat do komponent níže. Další důležitou vlastností stavu v Reactu je, že musí být immutabilní.

React je pouze knihovna pro uživatelské rozhraní, avšak existují i frameworky [31], které používají React pro vrstvu uživatelského rozhraní a obsahují i další nástroje například pro routování mezi stránkami aplikace nebo pro stahování dat. Často jsou v kombinaci s ním také používány knihovny pro správu stavu aplikace.

5.2.2 Angular

Angular [13] není jen knihovna pro uživatelské rozhraní, ale celý framework jehož součástí jsou i další nástroje například pro stahování dat ze serveru, routování stránek nebo automatické testování. Je postavený na jazyku TypeScript [12], což je nadstavba nad jazykem JavaScript přidávající statickou typovou kontrolu.

Uživatelské rozhraní je v něm také tvořeno pomocí komponent, ale klasičtějším způsobem pomocí jazyků HTML a CSS. Komponenta je třída, která obsahuje metody volané frameworkem po různých akcích, které v komponentě nastanou. Takové akce mohou být například vytvoření

komponenty či kliknutí na nějaký prvek v jejím uživatelském rozhraní. Kromě komponent obsahuje například ještě atributové direktivy. Ty nemají vlastní uživatelské rozhraní, ale přidávají se na existující komponenty nebo HTML elementy, kterým tak rozšiřují funkčnosti.

Stejně jako React také používá one-way data flow, ale na rozdíl od něho není nutné, aby stav byl immutabilní. Logika aplikace je umístěna v tzv. službách a celá aplikace je provázána pomocí vzoru Dependency Injection.

5.2.3 Vue

Tak jako React i Vue [32] je pouze knihovna pro tvorbu uživatelského rozhraní. Komponenty se v něm typicky definují v jednom souboru pomocí kombinace HTML, CSS a JavaScriptu. Pro funkční kód komponenty Vue nabízí dva způsoby. První způsob, více podobný Angularu, je Options API, v jednom objektu je definován stav komponenty a metody pro jeho změnu. Druhý způsob, naopak více podobný Reactu, je Composition API, zde je kód JavaScriptový skript a stav je umístěn ve v něm volaných funkcích podobných React hookům. I tak je ale stále na rozdíl od Reactu šablona definována pomocí klasického HTML a stav nemusí být immutabilní.

5.2.4 Rozhodnutí

Žádná z těchto technologií nemá nějakou zásadní výhodu či nevýhodu, která by některou z nich stavěla nad ostatní, nebo naopak komplikovala vytvoření požadované aplikace. Kromě vrstvy uživatelského rozhraní budou potřeba i další nástroje například pro automatické testování a komunikaci se serverem. Tato aplikace nevyžaduje použití nějakých konkrétních, a je tak pro ušetření práce s jejich výběrem výhodné použít celý framework. Tím tak odpadají samostatně použité knihovny React a Vue řešící pouze UI a v úvahu přichází frameworky [31] [33], které je obsahují.

Po jejich zvážení byl opět stejně jako v původním řešení zvolen framework Angular a z něho vyplývající jazyk TypeScript. Prvním důvodem pro jeho výběr je, že se jedná o nejpopulárnější framework, což usnadňuje hledání informací a knihoven. Druhý důvod je, že autor této práce s ním má ze zvažovaných frameworků největší zkušenosti. Není to však jednoznačná volba a dobře by pravděpodobně posloužily i jiné.

5.3 Výběr technologie pro server

Následuje výběr frameworku pro serverovou část. Ten opět bude určovat i použitý programovací jazyk.

5.3.1 ASP.NET Core

ASP.NET Core [34] je framework postavený na platformě .NET [35]. Je v něm možné vytvářet jak celé webové aplikace včetně HTML šablon se šablonovací syntaxí Razor [36], tak i pouze API, jako je třeba REST [37] nebo gRPC [38]. Používat lze jazyk C# [39] a do jisté míry také další jazyky platformy .NET, nicméně ty nejsou příliš rozšířené.

Pro tvorbu je možné využít dva přístupy. První, pravděpodobně nejrozšířenější, je návrhový vzor MVC. Akce kontrolerů odpovídají jednotlivým URL adresám aplikace a framework se stará o tzv. binding, což je naparsování vstupních dat do objektů a jejich předání těmto akcím. Druhou možností jsou Minimal APIs. To je zjednodušený přístup, kdy je URL adresa namapována pouze na předanou funkci. Framework se opět stará o binding dat do parametrů této funkce.

Další věcí, co framework poskytuje, jsou například nástroje pro Dependency Injection. V DI kontejneru se zaregistrují služby a framework je poté automatický vkládá do kontrolerů a dalších tříd frameworku. Také dodává nástroje pro validaci vstupních dat pomocí validačních atributů

umísťovaných nad jednotlivé vlastnosti datového modelu. Součástí je i podpora pro autentizaci a autorizaci, opět z velké části vyřešena deklarativně pomocí atributů.

5.3.2 Spring

Dalším známým frameworkem, tentokrát ve světě Javy, je Spring [40]. Podporovány jsou zejména jazyky Kotlin [41] a Java [42]. Kotlin má však od jeho tvůrců také vlastní framework Ktor [43].

V základních ohledech a přístupech je velmi podobný frameworku ASP.NET Core. Opět se zde využívá vzor MVC pro mapování URL adres na volání metod. Také se používá dependency injection. Validaci lze řešit pomocí anotací, což je obdoba atributů z platformy .NET.

5.3.3 Express

Zástupcem frameworků postavených na jazyku JavaScript je Express [44]. Co se týče vlastností obsažených přímo ve frameworku, tak Express je minimalistický a z předchozích dvou tak nejchudší. Pokročilejší vlastnosti jsou řešeny knihovnami třetích stran. Není zde v základu obsažena podpora pro MVC, a mapování je tak řešeno stylem podobným Minimal APIs ve frameworku ASP.NET Core.

5.3.4 Rozhodnutí

Hlavní výhoda Expressu pro tuto aplikaci by byla, že používá JavaScript a šlo by tak jednoduše sdílet části kódu mezi klientem a serverem. Z důvodu jeho minimalistického návrhu však nebyl zvolen. Jiné JavaScriptové frameworky nebyly uvažovány kvůli jejich nižší popularitě, která by ztěžovala dohledávání informací při vývoji. Mezi frameworky ASP.NET Core a Spring byla kvůli jejich podobnosti volba složitější. Nakonec byl vybrán ASP.NET Core s jazykem C#. Zejména z důvodu preference autora práce, který s ním má větší předchozí zkušenosti.

5.4 Výběr rozhraní mezi klientem a serverem

Z rozdělení aplikace na klienta a server plyne nutnost vybrat technologii pro jejich komunikaci.

5.4.1 REST API

REST je styl architektury API poprvé představený v disertační práci [37] Roye Fieldinga. Je postaven na tzv. zdrojích. Zdroj je obecně typ informace, kterou server nabízí. Může to tak být například obrázek, video, text nebo jiný typ dat. V kontextu této aplikace zdroje mohou být třeba projekty nebo výzvy. REST je pak v jeho práci definovaný pomocí šesti základních principů, třemi z nich jsou:

Bezstavovost Server zpracovává požadavky nezávisle na ostatních. Všechny informace, které server pro zpracování požadavku potřebuje, tak musí být obsaženy v požadavku samotném.

Cache Data v odpovědi musí být explicitně nebo implicitně označená jako cachovatelná.

Jednotné rozhraní Jednotné rozhraní je definováno čtyřmi omezeními (překlad autora práce):

- „*Identifikace zdrojů*“ [37]
- „*Manipulace se zdroji skrze jejich reprezentace*“ [37]
- „*Samopopisné zprávy*“ [37]
- „*Hypermedia jako základ stavu aplikace*“ [37]

Přesto, že to není explicitně vyžadováno, tak v praxi je REST API téměř výhradně implementováno pomocí protokolu HTTP. Pro odlišení typů jednotlivých požadavků se používají standardní HTTP metody, jako je GET, POST nebo DELETE. Pro signalizaci úspěchu či neúspěchu se zase vrací standardní HTTP stavové kódy. Samotná data se pak posílají v těle požadavku. Pro serializaci dat se nejčastěji využívá textový formát JSON. Pro identifikaci zdrojů se používají URL adresy. [45]

Příklad jednoduchého REST API s jedním zdrojem ukazuje tabulka 5.1.

GET	/books/{id}	Vrátí knihu podle jejího ID.
POST	/books	Vytvoří novou knihu.

■ **Tabulka 5.1** Příklad REST API

5.4.2 GraphQL

V REST API byly serverem striktně definovány zdroje a jejich reprezentace. Klient si nemohl určit, která data potřebuje, a někdy tak musel stahovat data navíc a jindy zase poslat více požadavků najednou. Tento problém řeší GraphQL [46], kde server pouze definuje schéma pro data, která poskytuje, a klient poté pomocí speciálního dotazovacího jazyka určí, která data požaduje, a přesně ta mu jsou vrácena. Kromě dotazů na data je možné i data měnit. K tomu ve schématu slouží tzv. mutations.

5.4.3 gRPC

Na rozdíl od předchozích dvou možností, které byly orientovány na data, je gRPC [38] orientováno spíše na akce. Jedná se o zástupce techniky remote procedure call, kdy klientská aplikace může pomocí RPC frameworku volat metodu v kódu serveru stejným způsobem, jako kdyby byla metoda dostupná lokálně. Vývojář pomocí speciálního jazyka specifikuje tzv. služby, které obsahují volatelné metody a nechá gRPC framework vygenerovat kód těchto služeb pro klienta a server. Jako formát přenosu dat framework ve výchozím stavu používá binární serializační formát Protocol Buffers [47].

5.4.4 Rozhodnutí

gRPC nebylo pro použití v prohlížeči primárně určeno a dříve to ani nebylo možné. Nyní to již možné je, ale pouze pomocí proxy. [48] Při jeho využití k tomuto mírně nestandardnímu účelu tak lze očekávat různé problémy. Nebylo by tedy vhodné. S GraphQL je situace jiná, pro použití v prohlížeči bylo určeno a ze všech tří je nejnovější. Jeho implementace je ovšem, ale oproti REST API složitější a řeší problémy, které tato aplikace pravděpodobně nebude mít. Bude tak stačit použití klasického REST API nad protokolem HTTP. Jeho největší výhodou je také, že je zdaleka nejrozšířenější.

5.5 Výběr databáze

Základní dělení databázi je na NoSQL a ty klasické SQL. Jedna se pouze o hrubé dělení, a tak každá databáze, která patří do jedné nebo druhé skupiny, nemusí nutně mít dále popsané vlastnosti úplně všechny.

5.5.1 SQL

Jako SQL databáze jsou označované klasické relační databáze ve kterých jsou data ukládána pomocí tabulek s předem definovanými schématy. Každý řádek tabulky tvoří jeden datový záznam a sloupce definují atributy těchto záznamů. Každý záznam má také v rámci tabulky jednoznačný identifikátor, kterému se říká primární klíč. Data tak nejsou ukládány v hierarchické struktuře a provázány jsou pouze pomocí svých klíčů. Při definici schéma dat v relačním modelu se používají principy normalizace, díky kterým se kvůli deduplikaci dat snižuje riziko porušení integrity a nároky na velikost úložiště. Jako vnější rozhraní pro práci s daty a jejich schématem tyto databáze typicky používají deklarativní jazyk SQL. [49]

Dále pro tento typ databází bývají společné vlastnosti označované zkratkou ACID [50]:

- **Atomicity** – Operace v rámci transakce jsou aplikovány všechny, nebo nejsou aplikovány vůbec.
- **Consistency** – Před i po skončení transakce jsou data v konzistentním stavu.
- **Isolation** – Stav probíhající transakce není vidět z ostatních transakcí. Změny dat tak jsou z vnějšku pozorovatelné až po jejím skončení.
- **Durability** – Po skončení transakce jsou data uloženy na persistentní úložiště. Kdyby tak byl databázový stroj po skončení transakce z nějakého důvodu náhle zastaven, tak změny budou zachovány.

5.5.2 NoSQL

Jedná se o souhrnný název více typů databázi, které ukládají a spravují data odlišně než ty klasické relační a které vznikly pro různé případy použití, kde relační databáze měly své limity. Relační databáze kvůli normalizaci dat mají nižší nároky na úložiště, a tak vznik těchto databází byl umožněn až ve chvíli, kdy cena úložišť klesla na úroveň, při které už nebylo nutné přehnaně dbát na deduplikaci dat. Jak už i název napovídá, tak pro práci s daty místo jazyka SQL používají různá vlastní rozhraní. [51]

Nejbližší NoSQL alternativou k relačním databázím jsou dokumentové databáze. Ty ukládají data v hierarchických dokumentech například ve formátu JSON. Konkrétním příkladem může být databáze MongoDB [52]. Ta jako formát dokumentů používá binární reprezentaci formátu JSON. Dokumenty seskupuje do kolekcí. Pro komunikaci s ní je dostupné velké množství oficiálních knihoven pro různé programovací jazyky.

Přesto, že NoSQL databázi je mnoho druhů, tak většinou poskytují několik společných výhod. Těch dosahují zejména jiným modelem dat než mají relační databáze. Například dokumentové databáze díky denormalizaci nabízí vyšší výkon při čtení, jelikož se data ukládají tak, jak budou požadována, aby je šlo získat najednou. Není tak potřeba je sestavovat spojováním několika tabulek pomocí klíčů jako v relačních databázích. Dosahují často také lepší horizontální škálovatelnosti. Další výhodou bývá větší flexibilita schématu na rozdíl od relačních databází, kde je nutné celé schéma určit předem. Například některé dokumentové databáze nevyžadují, aby všechny dokumenty v kolekci měly stejné atributy a jejich datové typy. [51]

5.5.3 Rozhodnutí

Pokud by byla využita NoSQL databáze, pravděpodobně by šlo o zmíněnou MongoDB. Výhoda by byla v téměř nativním ukládání projektů jazyka Karel, jejichž serializační formát je JSON.

Nakonec však byla zvolena relační databáze SQL Server [53]. Databáze je od společnosti Microsoft stejně jako zvolený framework ASP.NET Core. Díky tomu jsou tyto technologie spolu často používanou volbou, což usnadní hledání dokumentace a řešení problémů. Dále jsou také

nejčastěji společně dostupné u klasických hostingových řešení, kde by s integrací jiné databáze byl o něco větší problém.

Lepší podpora pro horizontální škálování u NoSQL databází při rozhodování nehrála téměř žádnou roli, neboť se u aplikace nepočítá s takovou zátěží, při které by to bylo nutné. Navíc i pro velké aplikace je možné s úspěchem použít relační databázi. Důkazem je například platforma StackOverflow, která podle [54] v kombinaci s cachovacími databázemi používá jako hlavní úložiště právě SQL Server databázi a mimochodem i framework ASP.NET.

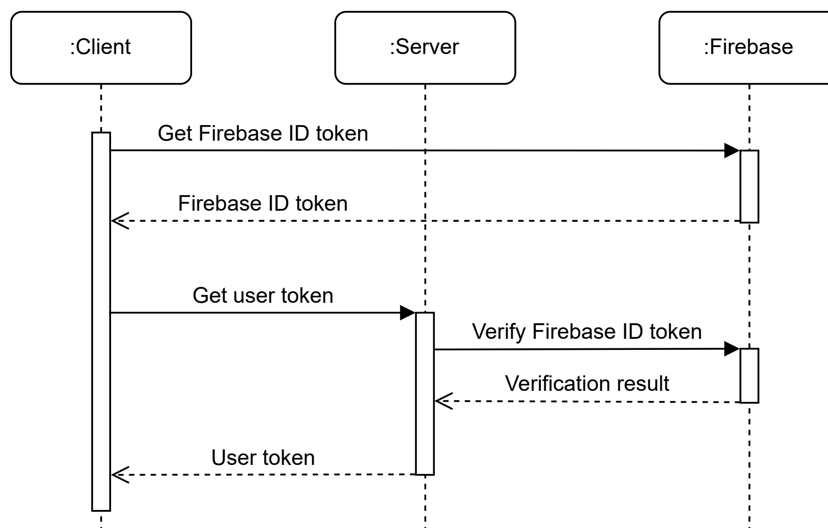
5.6 Přihlašování uživatelů

Z požadavků plyne, že aplikace pro určité funkcionality vyžaduje přihlášené uživatele. Klasickým způsobem je přihlašování pomocí emailu a hesla. Uživatel se vyplněním svých údajů nejprve zaregistruje a poté se do aplikace může přihlašovat.

Problém je však hned na začátku, kdy by nutnost registrace, ještě navíc v kombinaci s potvrzením emailu, mohla některé potenciální uživatele aplikace odradit. Navíc je tento způsob implementačně dost komplexní. Je třeba zajistit odeslání potvrzovacího emailu, vytvořit několik dialogů pro registraci, přihlášení, změnu hesla nebo profilových informací. Je nutné dbát na bezpečné uchovávání hesel a uživatelé zase musí dbát na jejich bezpečné vymýšlení. Přesto, že by v tom mohla částečně pomoci nějaká knihovna, tak byl tento způsob z těchto důvodů zamítnut.

Druhou možností je k přihlašování využít nějakou existující službu. Nabízí se třeba Google, Facebook nebo Microsoft, kde by většina uživatelů již mohla mít vytvořený účet, přes který by se mohli do aplikace přihlásit. Přes aplikaci tak vůbec neprojde uživatelovo heslo.

Nakonec byla zvolena kombinace obojího a k přihlašování uživatelů použita BaaS (Backend as a Service) platforma Firebase [55], konkrétně její řešení Firebase Authentication. Při použití takového řešení jsou účty uživatelů včetně jejich hesel (přesněji hashů hesel) uloženy na serverech platformy. Ta poté k přihlašování nabízí několik možností, včetně již zmíněných služeb třetích stran a té klasické pomocí emailu a hesla. Také poskytuje knihovny pro programovací jazyky, pomocí kterých je možné uživatele přihlásit, změnit mu heslo či vyžádat si jeho profilové informace. Není tak nutné řešit rozdíly v přihlašování mezi jednotlivými službami a starat se o bezpečné ukládání hesel. Platforma zvládá i posílání potvrzovacích emailů či emailů pro změnu hesla. V této práci bude využito pouze přihlášení pomocí Google účtu.



■ **Obrázek 5.2** Navržený proces autentizace

Jelikož je Firebase řešení typu BaaS, tak je správa uživatelů a komunikace s Firebase ovládána z klientské strany aplikace. Přesto ale server potřebuje znát přihlášeného uživatele a na základě toho autorizovat přístup ke svým zdrojům. Tokeny, které Firebase uživatelům vydává, jsou klasické JWT tokeny se zdokumentovanými parametry [56], a je tak možné na serveru ověřovat přímo ty. Nakonec ale byla zvolena cesta vydávání vlastního JWT tokenu výměnou za ten od Firebase. Výhoda tohoto přístupu je větší kontrola. Je například možné před vydáním tokenu v databázi Firebase ještě ověřit další vlastnosti uživatele jako třeba to, jestli má ověřenou emailovou adresu. Celý navržený proces autentizace je znázorněn na obrázku 5.2.

5.7 Automatické hodnocení výzev

Nejprve bylo třeba vymyslet způsob, jakým administrátoři aplikace budou definovat pravidla pro hodnocení výzev. Největším problémem se ukázalo být, jak vyvážit jednoduchost s dostatečnou obecností takového řešení.

5.7.1 Pomocí skriptu

Zpočátku byl uvažován a zkusmo implementován způsob, kdy administrátor pomocí jazyku JavaScript definuje funkci, která ve svých parametrech dostane objekt reprezentující odevzdaný projekt. Funkce projekt ohodnotí a vrátí výsledek obsahující informaci, zda projekt prošel jako úspěšný či neúspěšný, spolu se zprávou obsahující případné další informace například o chybě, která nastala při vyhodnocování. Takové řešení je prakticky nejobecnější implementovatelné. Bylo potenciálně možné v takové funkci využít přímo kompilátor jazyka a nad syntaktickým stromem provádět nejrůznější analýzy. Například kontrolovat správnost odsazení, jmenných konvencí, počtu příkazů a tak dále. Také bylo možné kód zkompilovat a spustit. Následně poté vyhodnotit, a to buďto výsledný stav města, nebo samotný průběh odevzdaného programu.

Rychle se ovšem ukázalo, že naprostá většina vymyšlených výzev takto komplexní hodnocení vůbec nepotřebuje a samotné vytváření výzev bylo příliš zdlouhavé, repetetivní a náchylné na chyby. Jako reakce na tento problém byla vytvořena pomocná JavaScriptová knihovna, která zaobalovala běžné funkce používané při hodnocení. Například snadno umožňovala spouštět odevzdaný kód nad předaným městem a bylo také plánováno v ní implementovat funkce pro generování vstupních měst.

Nicméně i s touto knihovnou zůstalo několik problémů nevyřešených. Ukázala se potřeba i ručně definovat vstupní města a to se v kódu dělalo jen velmi špatně. Neméně závažným problémem by bylo při implementaci ošetření bezpečnostní rizik plynoucích ze spouštění uživatelem zadaného kódu. Byť tato funkcionalita měla být dostupná jen pro administrátory.

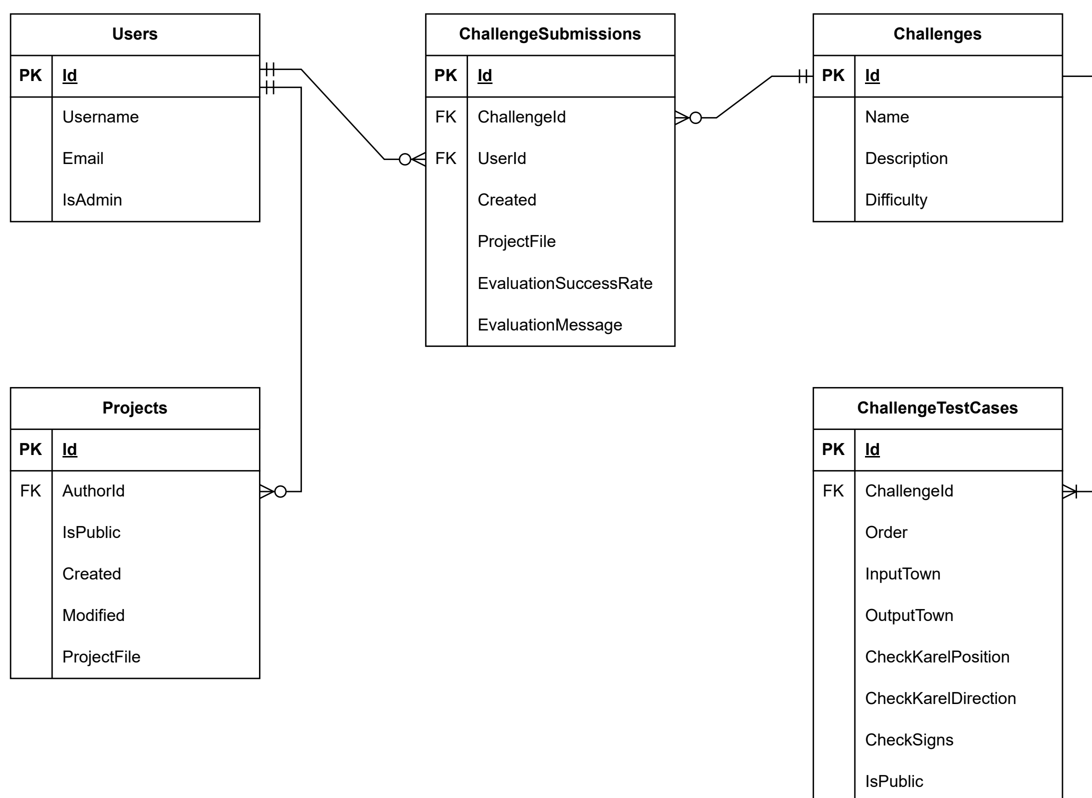
5.7.2 Pomocí testovacích případů

S přihlédnutím k těmto zkušenostem bylo navrženo nové řešení založené na testovacích případech. Testovací případ obsahoval dvě města, vstupní město a očekávané výstupní město, tedy očekávaný stav vstupního města po provedení odevzdaného programu. Ne vždy bylo žádoucí, aby se očekávané a reálné výstupní město museli shodovat přesně. Například při výzvě hledání cesty v bludišti nezáleželo na konečném otočení robota ani na položených značkách ve městě, ale pouze na jeho pozici. Jindy naopak pozice nebyla důležitá a roli hráli počty značek na jednotlivých polích. Proto byli k testovacímu případu přidány ještě tři flagy specifikující míru požadované shody. První určuje, zda se musí shodovat pozice Karla, druhý zda jeho natočení a třetí zda počty značek na polích města. Takovýchto testovacích případů mohla a typicky obsahovala výzva několik. Byl tak zajištěn požadavek na obecnost odevzdaného kódu, například aby fungoval pro různé velikosti města či různé počáteční pozice robota.

Toto řešení viditelně neumožňovalo například zmíněnou kontrolu odsazení či jmenné konvence, to ale v naprosté většině případů nebylo potřeba. Byla tak za cenu pouze malého snížení flexibility získána velká jednoduchost a rychlost při definování výzev.

5.8 Databázové schéma

Na obrázku 5.3 je pomocí ER diagramu znázorněno navržené schéma databáze. Jde o konceptuální model a nejsou tak uváděny datové typy.



■ **Obrázek 5.3** Konceptuální model databáze

Nejsložitější se při návrhu ukázalo být vymyslet, jak ukládat projekty. Ze zjevných důvodů nebylo možné je normalizovat do tabulek až na úroveň dlaždic města. Nakonec byl zvolen nejpřímočařejší způsob a JSON soubor s projektem ukládán jako celek v jednom sloupci.

Dalo by se namítnout, že pak nebude možné projekt dále analyzovat a nebo například vyhledávat podle některých jeho vlastností, jako je třeba název. Zvolená databáze SQL Server ale umožňuje s JSON daty pracovat a v její dokumentaci je i uveden způsob, jak je lze indexovat [57]. V naprosté většině případů je také projekt chápán jako běžný soubor, a bude tak i získáván jako jeden celek. Normalizace do tabulek by tak nepřinášela žádné výhody navíc, stejně by na nějaké úrovni musela končit a schéma projektu by muselo být drženo v souladu s tím definovaným na TypeScriptové straně.

5.9 REST API

Navržené REST API endpointy popisuje tabulka 5.2. Endpointem se rozumí kombinace HTTP metody a adresy, ke které se může uživatel API připojit a provést požadavek.

Zdroji v této aplikaci jsou výzva, odevzdání výzvy, projekt a uživatel. Pro jejich reprezentaci byl, tak jak je běžné, použit textový formát JSON. Každému zdroji v návrhu odpovídá jedna adresa, kterou používají jeho endpointy. Dále byl potřeba endpoint pro autentizaci přes Firebase, ten byl umístěn na adresu `/Authentication/Firebase`.

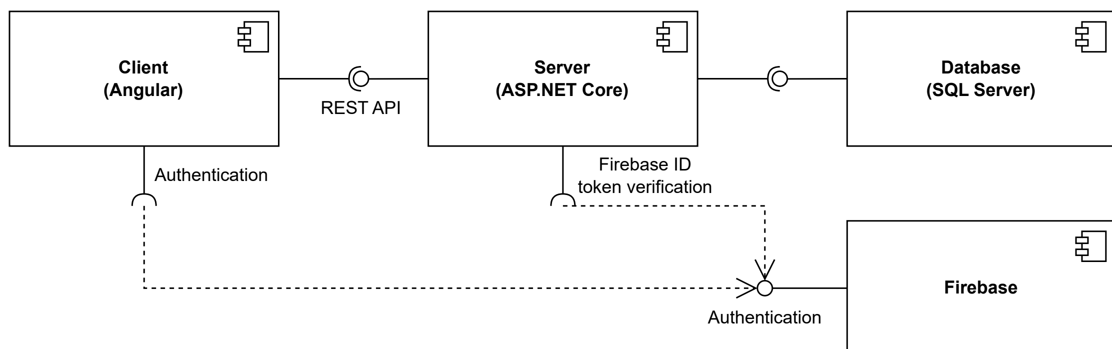
POST	/Authentication/Firebase	Vymění Firebase ID token za token aplikace.
GET	/Challenges	Vrátí všechny výzvy.
GET	/Challenges/{id}	Vrátí výzvu s daným ID.
POST	/Challenges	Přidá novou výzvu.
PUT	/Challenges/{id}	Aktualizuje výzvu s daným ID.
DELETE	/Challenges/{id}	Odstraní výzvu s daným ID.
GET	/ChallengeSubmissions	Vrátí všechny odevzdání dané výzvy.
GET	/ChallengeSubmissions/{id}	Vrátí odevzdání s daným ID.
POST	/ChallengeSubmissions	Odevzdá řešení dané výzvy.
GET	/Projects	Vrátí všechny projekty.
GET	/Projects/{id}	Vrátí projekt s daným ID.
POST	/Projects	Vytvoří nový projekt.
PUT	/Projects/{id}	Aktualizuje projekt s daným ID.
DELETE	/Projects/{id}	Odstraní projekt s daným ID.
GET	/Users/Current	Vrátí aktuálně autentizovaného uživatele.

■ **Tabulka 5.2** Návrh endpointů REST API

Jde jen o návrh z nejvyšší úrovně pohledu. Dále je samozřejmě potřeba pro každý endpoint navrhnout návratové HTTP kódy, query parametry v URL adrese a strukturu požadavku a odpovědi. Kvůli přílišnému rozsahu takové úrovně detailu se tímto text práce už nezabývá.

5.10 Shrnutí architektury

Celou navrženou architekturu aplikace i se zvolenými technologiemi shrnuje obrázek 5.4.



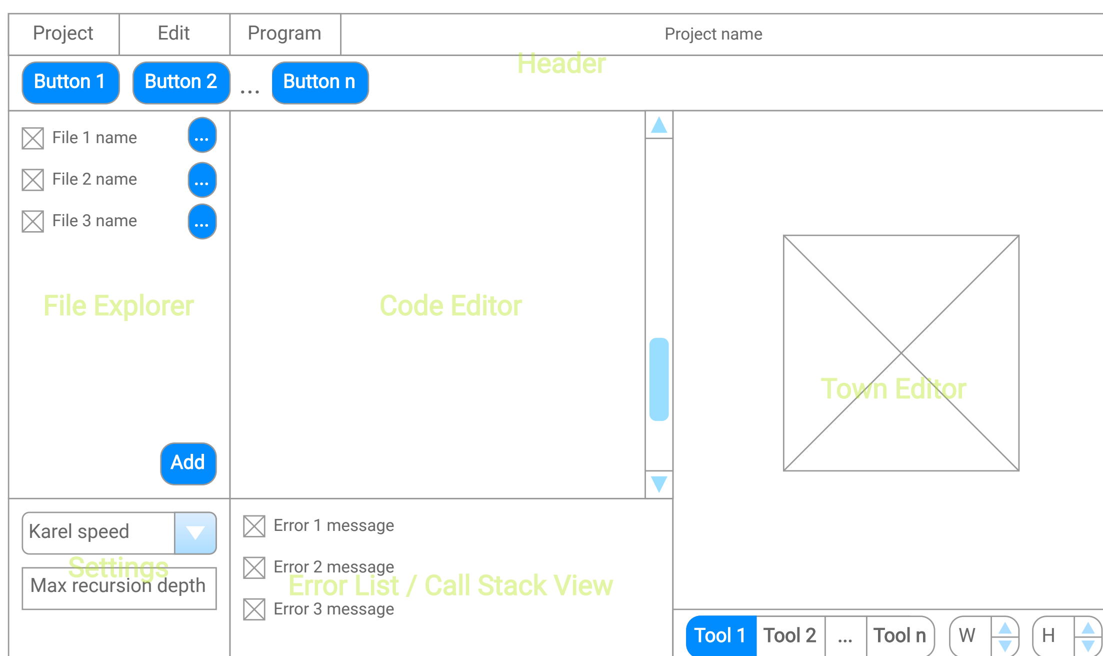
■ **Obrázek 5.4** Navržená architektura aplikace

5.11 Obrazovky

Poslední částí je návrh obrazovek a jejich uživatelského rozhraní. K tomu byl použit wireframe. To je graficky zjednodušený náhled jednotlivých obrazovek aplikace, který neřeší celkový grafický vzhled jako barvy a tvary ani finální texty, jde v něm pouze o rozložení prvků na obrazovce a popis akcí, které na nich uživatel může provádět.

5.11.1 Editor projektu

Editor je hlavní obrazovka celé aplikace. V něm uživatel tvoří kód, město a spouští vytvořený program. Obrazovka je rozdělena do několika panelů na wireframu popsaných zelenou barvou. Rozložení těchto panelů je stejné jako jejich rozložení v původním řešení, které se osvědčilo.

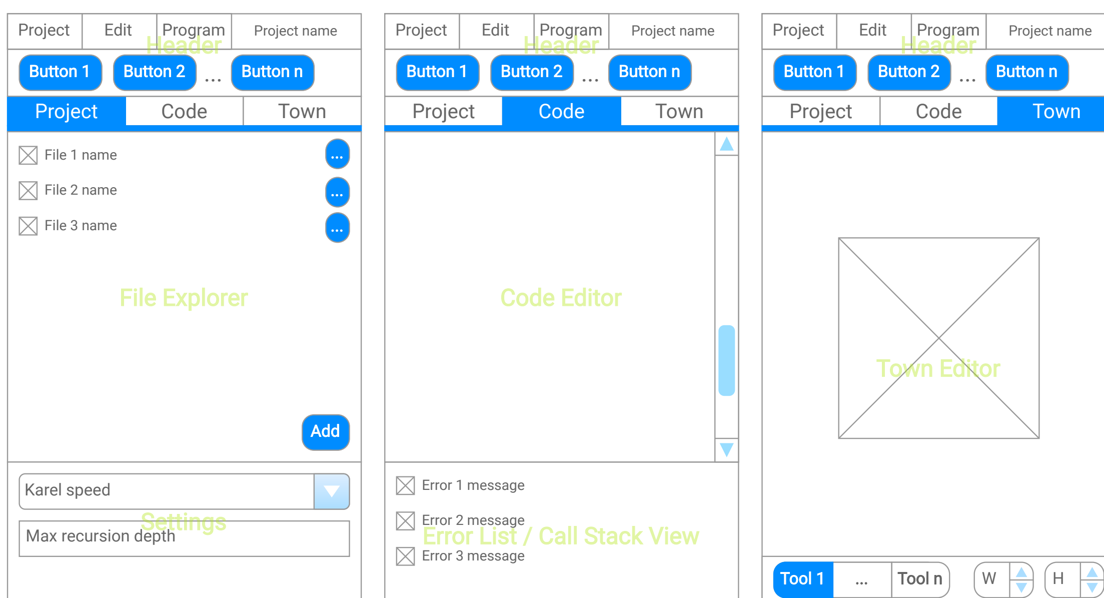


■ **Obrázek 5.5** Wireframe obrazovky editoru projektu

Na panelu „File Explorer" jsou zobrazeny soubory v aktuálně otevřeném projektu. Ty se zde dají také vytvářet a mazat. Uživatel tady může mít v jednu chvíli otevřený jeden soubor s kódem a jeden s městem. Editor kódu na panelu „Code Editor" umožňuje upravovat kód v aktuálně otevřeném souboru. Editor města („Town Editor") zase ve své spodní části poskytuje nástroje pro úpravu města z aktuálně otevřeného souboru s městem. V nastavení („Settings") lze nastavit parametry spouštění, konkrétně rychlost robota a omezení velikosti zásobníku volání. Panel pod editorem kódu („Error List / Call Stack View") při psaní zobrazuje chyby a v případě pozastaveného programu zase stav zásobníku volání.

Horní lišta („Header") obsahuje ovládací prvky celého editoru. V prvním řádku jsou tři rozbalovací menu se všemi dostupnými funkcionalitami. V druhém řádku jsou pak pro rychlejší dostupnost ty, u kterých se očekává, že budou často používány.

Vzhledem k velkému počtu panelů a ovládacích prvků nebylo jednoduché tuto obrazovku navrhnout pro užší rozlišení, zejména mobilních telefonů. Nakonec byl pro pohodlné ovládání zvolen způsob, kdy byly panely rozděleny do tří skupin a každá skupina byla schována do vlastního tabu mezi kterými je možné přepínat. Ukazuje to obrázek 5.6 na další stránce.

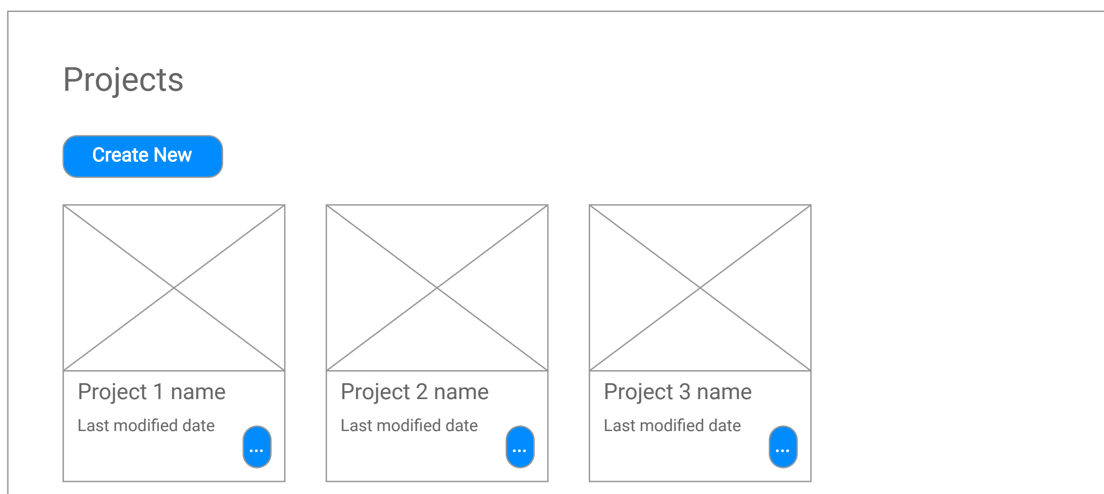


■ **Obrázek 5.6** Wireframe obrazovky editoru projektu ve verzi pro úzké rozlišení

Speciálně muselo být u mobilních zařízení myšleno také na ovládání editoru města. Počítá se s podporou dotykových gest pro posun a přiblížení. Také na rozdíl od původního řešení bude ovládání pomocí pravého tlačítka a kolečka myši jen alternativní možností, vše jinak musí být dostupné i přes levé tlačítko nebo gesta.

5.11.2 Seznam projektů

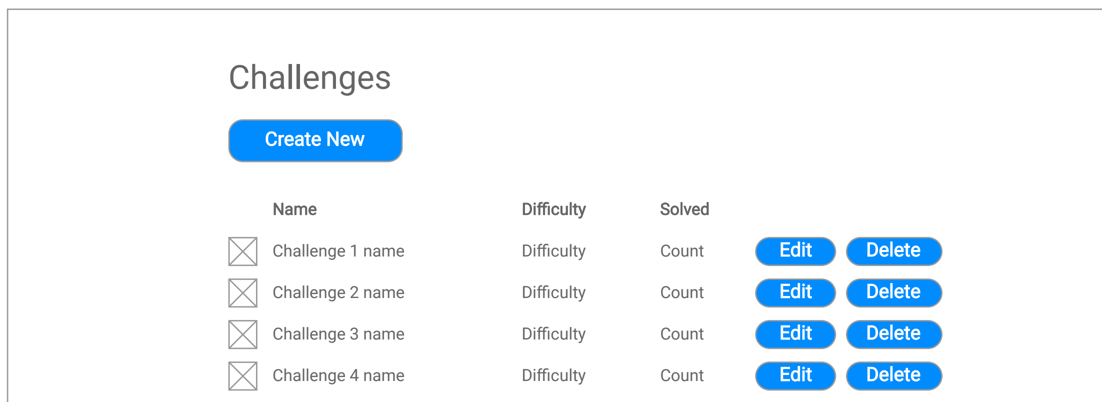
Na této obrazovce přihlášený uživatel vidí seznam svých uložených projektů spolu s jejich náhledy. Náhled je generován z některého z měst, které jsou v projektu obsaženy. Projekty jsou pro snazší orientaci seřazeny podle data a času poslední úpravy. Je zde také možnost vytvořit nový projekt nebo odstranit existující.



■ **Obrázek 5.7** Wireframe obrazovky seznamu projektů

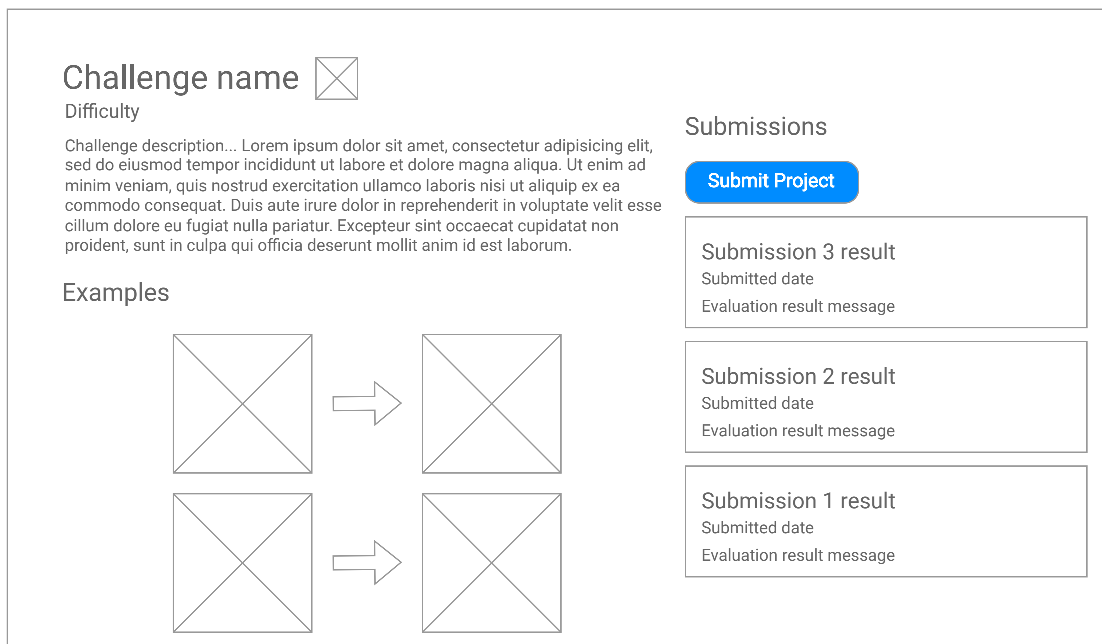
5.11.3 Seznam výzev

Tato obrazovka v tabulce zobrazuje seznam dostupných výzev. U každé výzvy je uvedena její obtížnost a počet uživatelů, kteří ji již splnili. Dále také pro lepší přehled uživatele, které má již splněné, je před názvem každé výzvy zobrazen stav jejího řešení. Rozlišují se tři stavy indikované příslušnou ikonou. Na začátku, ještě před tím, než uživatel odešle řešení výzvy, není zobrazena žádná ikona. Po odeslání řešení, které kontrolou projde jako neúspěšné, je zobrazena ikona vlaječky, informující, že je výzva rozpracovaná. Po odeslání řešení, které projde jako úspěšné, je poté již napořád zobrazena ikona fajfky, informující, že výzva je přihlášeným uživatelem již splněna.



Obrázek 5.8 Wireframe obrazovky seznamu výzev

5.11.4 Výzva



Obrázek 5.9 Wireframe obrazovky výzvy

Zde se nachází samotný text zadání výzvy spolu s ukázkovými vstupy a výstupy. Vedle názvu výzvy se opět zobrazuje ikona informující o jejím stavu. Dále v pravé části této obrazovky lze odeslat některý z dříve na serveru uložených projektů jako řešení. Po odeslání se řešení objeví v seznamu. Je u něho uvedena informace o úspěšnosti a případně chybová zpráva.

5.11.5 Editor výzvy

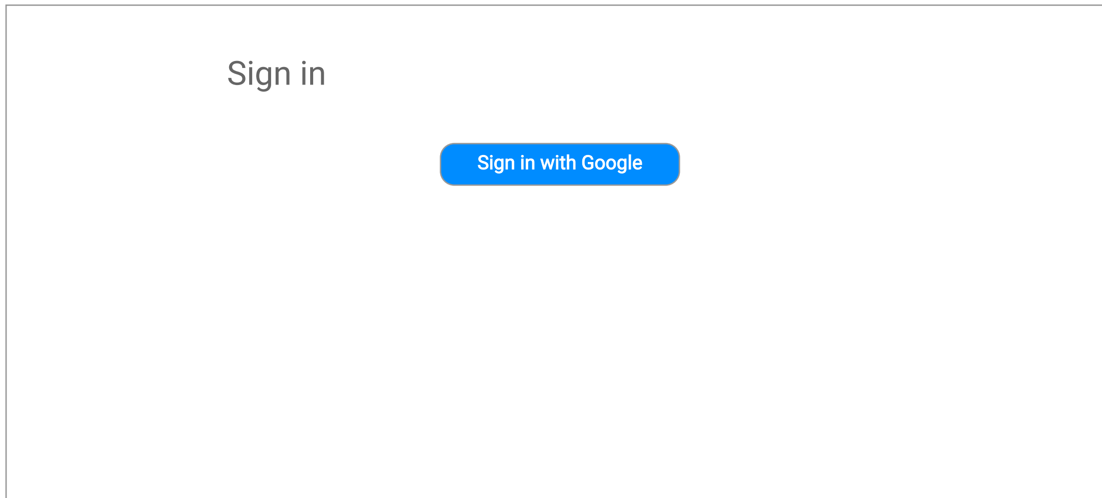
Na této obrazovce je administrátory aplikace možné vytvářet a upravovat jednotlivé výzvy. Jsou zde pole pro specifikaci názvu, obtížnosti a popisu zadání. Pro použití formátování pole pro popis podporuje textový značkovací jazyk Markdown.

Ve spodní části obrazovky lze editovat a přidávat testovací případy. Pro každý testovací případ jsou dostupné dva editory města. Vlevo pro editaci města na vstupu a vpravo pro editaci očekávaného města na výstupu. Nad těmito editory se dá pomocí zaškrtávacích boxů určit požadovaná míra shody skutečného a očekávaného výstupního města.

Obrázek 5.10 Wireframe obrazovky editoru výzvy

5.11.6 Přihlášení

Na této obrazovce je umístěno tlačítko pro přihlášení pomocí Google účtu. Je zde necháno místo i pro další možnosti přihlášení v případných budoucích verzích aplikace.



Obrázek 5.11 Wireframe obrazovky přihlášení

Kapitola 6
Implementace

Celý projekt byl při vývoji verzován pomocí nástroje Git [58]. Všechny zdrojové kódy aplikace se nacházejí v jednom repozitáři hostovaném na platformě GitHub [59]. Jednotlivé komponenty celkové architektury aplikace jsou implementovány v jim odpovídajících složkách v kořeni repozitáře. Tyto složky ukazuje obrázek 6.1.

```
.github.................................Konfigurace CI/CD pipeline
├─ client...........................Implementace klientské části aplikace
├─ firebase-emulator......................Emulátor platformy Firebase
└─ server...........................Implementace serverové části aplikace
```

Obrázek 6.1 Přehled složek v kořeni repozitáře

6.1 Implementace serverové části

Serverová část aplikace je vytvořena v jazyku C# s pomocí frameworku ASP.NET Core. Data ukládá do relační databáze SQL Server. Pro komunikaci s klientskou části poskytuje REST API.

6.1.1 Architektura

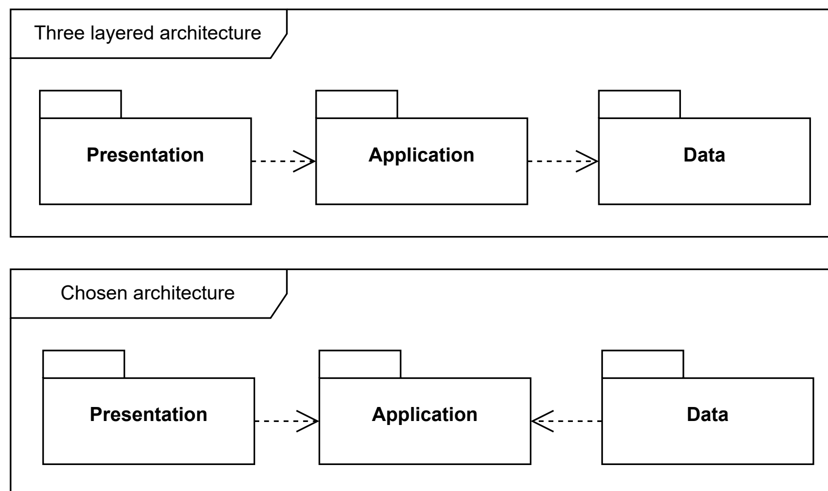
Klasická třívrstvá architektura [60] dělí aplikaci na níže popsané tří vrstvy. Závislosti mezi těmito vrstvami vedou odshora dolů. Mimo jiné tak prezentační přímo nebo ještě s mezikrokem přes aplikační závisí na datové.

Prezentační vrstva Zpracovává vstupy a prezentuje výsledná data, ať už formou uživatelského nebo programového rozhraní.

Aplikační vrstva Obsahuje business logiku aplikace, tedy typicky to hlavní, kvůli čemu aplikace byla vytvořena.

Datová vrstva Stará se o uchovávání a správu dat. Jedná se tedy například o práci s databází nebo nějakou vzdálenou službou poskytující data.

Pro architekturu serverové části byla však zvolena trochu jiná architektura podobná těm, jako je Onion Archiecture [61] nebo Clean Architecure [62], které jsou založeny na principu dependency inversion. V těchto architekturách je závislost mezi aplikační a datovou vrstvou obrácena a tak prezentační i datová vrstva závisejí pouze na už nezávislé aplikační vrstvě.

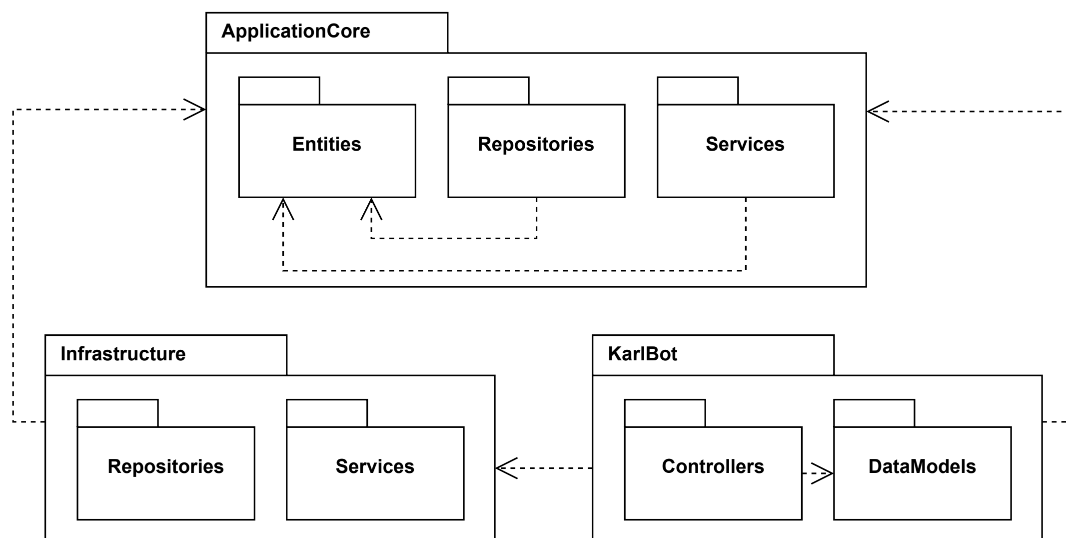


■ **Obrázek 6.2** Porovnání třívrstvé a zvolené architektury

Rozdíl ukazuje obrázek 6.2. Výhodou těchto architektur je větší flexibilita, jelikož třídy na prezentační a aplikační vrstvě mohou být vyvíjeny nezávisle na datové a lépe tak odstíněny od jejích implementačních detailů. Přináší také lepší testovatelnost, neboť už na úrovni architektury je zajištěno, že třídy aplikační vrstvy lze testovat v izolaci od ostatních.

V objektových programovacích jazycích využití této architektury typicky znamená, že aplikační vrstva pro třídy pracující s daty obsahuje jen rozhraní. Ta jsou pak implementována třídami v datové vrstě. Třídy prezentační vrstvy tak závisí pouze na těchto rozhraních.

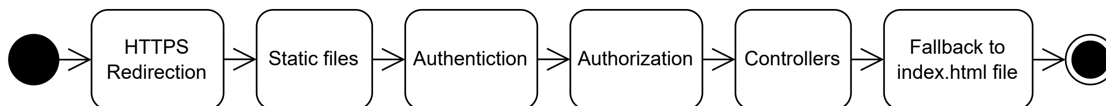
Jak je znázorněno na obrázku 6.3, tak pro implementaci této architektury byla celá aplikace rozdělena do tří .NET projektů odpovídajících jednotlivým vrstvám. Projekt `ApplicationCore` odpovídá aplikační vrstvě, projekt `Infrastructure` datové a projekt `KarlBot` prezentační. Toto pojmenování vychází z architektonické příručky frameworku ASP.NET Core [63]. Rozdělení do projektů umožňuje vynutit správné závislosti mezi vrstvami a lépe je oddělit. Každá vrstva také díky tomu může používat své vlastní knihovny bez ohledu na ostatní.



■ **Obrázek 6.3** .NET projekty serverové části aplikace

6.1.2 Celkový pohled

Vstupním bodem této a i jiných aplikací ve frameworku ASP.NET Core je C# soubor `Startup.cs`. V něm je definována sekvence zpracování HTTP požadavku (obrázek 6.4), na jejímž konci jsou namapovány MVC kontrolery, jejichž metody tvoří jednotlivé REST API endpointy. Při přijetí požadavku ho tyto metody kontrolerů zpracují a pro provedení samotné akce už zavolají vhodné služby, nebo repozitáře, v případě jednoduchých CRUD operací. Zjednodušeně řečeno, služby obsahují business logiku a repozitáře se starají o persistenci dat. Jako datové modely si mezi sebou předávají třídy reprezentující doménové entity. Po provedení příslušné akce vrátí výsledek zpět kontroleru a ten se postará o jeho převedení do tvaru odpovědi daného REST API endpointu.



Obrázek 6.4 Middleware pipeline zpracovávající HTTP požadavky

6.1.3 Entity

Doménové entity aplikace jsou umístěny na aplikační vrstvě. V této aplikaci je entitou například projekt, výzva nebo uživatel. Jsou reprezentovány běžnými C# třídami bez jakékoliv závislosti na databázi (persistence ignorance princip) nebo složitější logiku. V tuto chvíli entity prakticky jedna ku jedné odpovídají tabulkám v databázi, ale nemuselo by tomu nutně tak být.

Každá entita má svůj jednoznačný identifikátor. Jako jeho datový typ byl zvolen GUID. Hlavní výhodou je, že je tak možné stejný identifikátor použít i pro vnější reprezentaci entity například v REST API. Protože díky svému velkému rozsahu a náhodném generování neposkytuje o datech prakticky žádné informace navíc, které by mohli být zneužity. Není ani dost dobře možné ho sekvenčně procházet například při použití v URL adrese.

6.1.4 Repozitáře

S entitami úzce souvisí repozitáře pro jejich ukládání. Kvůli testovatelnosti tříd používajících databázi a oddělení zodpovědností je práce s databází v aplikaci zapouzdřena do tříd nazývaných repozitáře. Je tak také o mnoho snazší vyměnit implementaci, například použít jinou relační nebo i nerelační databázi.

Každá entita má vlastní rozhraní repozitáře, které obsahuje metody pro základní CRUD operace a případně ještě některé další rozšiřující. Například pro vyhledání na základě dalších kritérií. Třeba v případě výzev jejich repozitář obsahuje metodu pro získání informací o tom, které má uživatel splněny. Každé rozhraní repozitáře dědí z jednoho společného generického rozhraní `IRepository<TEntity, TKey>`, které definuje základní CRUD operace (výpis kódu 6.1).

```
public interface IRepository<TEntity, TKey>
{
    Task<List<TEntity>> GetAsync();
    Task<TEntity?> GetByIdAsync(TKey id);
    Task<bool> ExistsByIdAsync(TKey id);
    Task AddAsync(TEntity entity);
    Task UpdateAsync(TEntity entity);
    Task RemoveAsync(TEntity entity);
}
```

Výpis kódu 6.1 Společné generické rozhraní repozitáře ze kterého dědí všechny ostatní

Rozhraní jsou definována na aplikační vrstvě a datová vrstva pak obsahuje jejich implementace. Tyto implementace opět kvůli neduplikování kódu sdílejí jednu společnou nadtřídu `DbContextRepository<TEntity, TKey>` implementující zmíněné `IRepository<TEntity, TKey>`.

S databází není komunikováno přímo, ale pomocí ORM knihovny Entity Framework Core [64]. Ta díky použití C# dotazovacího jazyka LINQ místo SQL částečně skrývá rozdíly mezi různými databázemi a hlavně poskytuje mapování databázových tabulek na C# třídy. Použití knihovny ukazuje výpis kódu 6.2.

```
public async Task<IList<Project>> GetAsync(Guid? authorId)
{
    IQueryable<Project> query = DbSet;
    if (authorId != null)
        query = query.Where(p => p.AuthorId == authorId);

    return await query.ToListAsync();
}
```

■ **Výpis kódu 6.2** Ukázka použití knihovny Entity Framework Core v repozitáři projektů

6.1.5 Služby

Služby jsou třídy, které obsahují business logiku aplikace. Tou je například hodnocení výzev. Je použit stejný přístup jako u repozitářů, kdy jejich rozhraní jsou definována v projektu ApplicationCore a implementace v projektu Infrastructure. Jsou tak získány i obdobné výhody.

6.1.6 Kontrolery

Prezentační vrstvu aplikace tvoří MVC kontrolery. Starají se pouze o zpracování REST požadavků, jejich delegování na aplikační vrstvu a následné vrácení odpovědi zpět. To zde znamená, že kontrolery buďto volají repozitáře a nebo služby. V těch je pak umístěna samotná logika aplikace.

Každý REST API endpoint je namapován na jednu metodu kontroleru. Pro definici formátu požadavků a odpovědí kontrolery používají immutabilní třídy nazvané data modely. U vlastností těchto tříd jsou také pomocí atributů specifikovaná validační pravidla, jejichž dodržení je automaticky hlídáno frameworkem. Jednoduchou metodu kontroleru ukazuje výpis kódu 6.3.

```
[HttpGet("current")]
public async Task<ActionResult<UserDataModel>> GetCurrentAsync()
{
    var user = await _userManager.GetUserAsync(User);

    if (user == null)
        throw new UnreachableException();

    return new UserDataModel
    {
        Id = user.Id,
        Email = user.Email!,
        IsAdmin = User.IsInRole(RoleNames.Admin)
    };
}
```

■ **Výpis kódu 6.3** Ukázka metody kontroleru pro získání aktuálního uživatele

6.1.7 Uživatelé a jejich autentizace

V navrženém řešení autentizace je jediná zodpovědnost serveru vydání JWT tokenu a na jeho základě pak autorizace svých zdrojů. K provedení autentizace v REST API slouží endpoint `/Authentication/Firebase`, který výměnnou za Firebase ID token vydá uživateli vlastní JWT token. Pro ověřování ID tokenů lze využít knihovnu Admin SDK [65] od Firebase. Ta je v kódu umístěna do služby `AdminSdkFirebaseAuthenticationService`. Vydání vlastního tokenu pak zajišťuje služba `UserTokenService`. Příklad obsahu takového tokenu ukazuje výpis kódu 6.4. Autorizaci provádí kontrolery, zčásti deklarativně pomocí autorizačních atributů umístěných u jejich metod.

Pro správu uživatelů je použito ASP.NET Core Identity. To je část frameworku, která se stará o správu uživatelů, jejich rolí a přihlašovacích metod. Pomocí napojení na Entity Framework Core řeší také jejich persistenci v databázi.

```
{
  "nameid": "5e2f0319-a4d2-4c26-df1d-08db33b3e9c8",
  "role": "Admin",
  "nbf": 1682509664,
  "exp": 1682524064,
  "iat": 1682509664,
  "iss": "https://karlbot.dev",
  "aud": "https://karlbot.dev"
}
```

■ **Výpis kódu 6.4** Ukázka obsahu JWT tokenu vydaného serverem

6.1.8 Hodnocení výzev

Aby byly výsledky hodnocení objektivní a nebylo je možné falšovat, je nutné řešení výzev hodnotit na serveru. S tím však přichází komplikace, jelikož kompilátor i interpret je napsán v jazyku TypeScript, kdežto server v jazyku C#. Udržovat byť jen interpret ve dvou jazycích by bylo příliš složité, takže bylo nutné nějak využít stávající TypeScriptovou implementaci.

První možností bylo vytvořit komponentu pro hodnocení výzev jako externí službu v samostatném procesu implementovanou pomocí nějakého JavaScriptového (TypeScript se překládá do JavaScriptu) frameworku. Použitím vhodného protokolu by pak komunikovala se serverem. Druhou možností bylo využít některou z knihoven pro spouštění JavaScriptu přímo z prostředí .NET. Při použití takového řešení skript typicky běží ve stejném procesu jako .NET kód.

První možnost je o něco svobodnější, její velkou nevýhodou je však vyšší složitost implementace a zejména nasazení, jelikož by bylo nutné nasazovat o jednu komponentu navíc. Hlavně kvůli složitosti nasazení tak byla zvolena druhá možnost. Pro spouštění JavaScriptu byla vybrána knihovna ClearScript [66] od společnosti Microsoft. Tato knihovna poskytuje .NET rozhraní pro JavaScriptový engine V8 [67], který mimo jiné pro spouštění JavaScriptu používá například webový prohlížeč Google Chrome.

Hodnocení odevzdaných řešení výzvy provádí třída `ClearScriptChallengeEvaluationService`. Obsahuje jednu veřejnou metodu `EvaluateAsync`, které se předá soubor s odevzdaným projektem a seznam testovacích případů dané výzvy. Samotné hodnocení probíhá na TypeScriptové straně a tato metoda se tak pomocí knihovny ClearScript pouze stará o správné předání jejích vstupů do TypeScriptového kódu, spuštění hodnocení a následné vrácení výsledku zpět (výpis kódu 6.5).

TypeScriptový hodnotící kód je pro snadné spuštění zabalen nástrojem webpack [68] do jednoho souboru. Jeho rozhraní pro C# stranu tvoří funkce `evaluate`. Tento kód již má přímý přístup ke knihovně jazyka Karel a může tak odevzdaný projekt načíst, zkompilovat a spustit. Poté stačí jen porovnat výsledné a očekávané město podle předaných testovacích případů. Z důvodu bezpečnosti bylo nutné ještě zajistit ochranu před zacyklením a přetečením zásobníku volání. To bylo vyřešeno jednoduše limitem na vykonaný počet instrukcí a hloubku zásobníku volání.

```
engine.Evaluate(_options.KarelLibrarySource);
engine.Evaluate(_options.KarelEvaluationLibrarySource);

var taskCompletionSource =
    new TaskCompletionSource<ChallengeEvaluationResult>();

Action<dynamic> resolveHandler = r =>
{
    var successRate = r.successRate;
    var message = r.message;
    var result = new ChallengeEvaluationResult(successRate, message);
    taskCompletionSource.SetResult(result);
};
Action<dynamic> rejectHandler = e =>
{
    taskCompletionSource.SetException(
        CreateEvaluationScriptErrorException(e));
};

var evaluationScriptTestCases = testCases
    .Select(tc => new { ... })
    .ToArray();

try
{
    var resultPromise = engine.Script.karelEvaluation.evaluate(
        projectFile,
        JsonSerializer.Serialize(evaluationScriptTestCases)
    );
    resultPromise.then(resolveHandler, rejectHandler);
}
catch (RuntimeBinderException exception)
{
    throw new Exception("Can not find a function ...", exception);
}

return taskCompletionSource.Task;
```

■ **Výpis kódu 6.5** Ukázka komunikace s TypeScript stranou pomocí knihovny ClearScript

6.1.9 Dependency injection

Celá aplikace je propojena pomocí návrhového vzoru dependency injection (dále DI). Při startu aplikace se v DI kontejneru zaregistrují služby, repozitáře, konfigurace a další komponenty aplikace (ukazuje výpis kódu 6.6). Zde se také k rozhraním služeb a repozitářů určí jejich implementace. Framework se poté stará o jejich automatické vytváření a vkládání do konstruktorů. Například konstruktory kontrolerů mají parametry s rozhraními požadovaných repozitářů a DI frameworkem jsou pak do nich automaticky předány konkrétní instance. Jako DI framework je využit výchozí v ASP.NET Core z namespace `Microsoft.Extensions.DependencyInjection`.

Jak bylo možné si všimnout na obrázku 6.3 popisujícím jednotlivé .NET projekty, tak navíc oproti zvolené architektuře projekt odpovídající prezentační vrstvě závisí na projektu datové vrstvy. To je právě z důvodu propojení aplikace, aby byla možná registrace implementací repozitářů z datové vrstvy do DI kontejneru. Jelikož právě projekt prezentační vrstvy je ten, který je spouštěný, tak je tato logika umístěna v něm. Šlo by sice vytvořit ještě jeden projekt, který

by byl spouštěn místo něj a staral se o sestavení aplikace, ale bylo rozhodnuto, že nevýhoda udržovat další projekt převažuje získané výhody a je tak jen potřeba si dát pozor, aby z jiných míst prezentační vrstvy než je kód registrace repozitářů nebyla datová vrstva referencována.

```
b.Services.AddControllers(o => ...);
b.Services.ConfigureOptions<ChallengeEvaluationOptionsConfiguration>();
b.Services.AddTransient<IUserTokenService, UserTokenService>();
b.Services.AddTransient<IUserRepository, DbContextUserRepository>();
```

■ **Výpis kódu 6.6** Ukázka části konfigurace DI kontejneru

6.1.10 Konfigurace aplikace

Konfigurací je v případě této aplikace například připojovací řetězec k databázi, nastavení JWT tokenů nebo přístupové údaje k Firebase. Hodí se, aby byla na jednom místě, a může být potřeba ji změnit i po kompilaci aplikace, je tak vhodné ji extrahovat ven ze zdrojového kódu. K tomu byly použity konfigurační možnosti frameworku .NET. Ty zajišťují načtení konfigurace z různých zdrojů jako jsou konfigurační soubory nebo proměnné prostředí a její namapování do C# tříd. Ty jsou pak po aplikaci distribuovány pomocí dependency injection.

6.2 Implementace klientské části

Klientská část je stejně jako v původním řešení implementována formou single page webové aplikace ve frameworku Angular. Jako programovací jazyk je použit TypeScript.

6.2.1 Architektura

Projekt je rozdělen do 5 modulů podle funkcionalit. Ty jsou pak dále rozděleny na prezentační a aplikační vrstvu. Každý modul je umístěn ve své vlastní složce, ty popisuje obrázek 6.5.

- `challenges` .... Vytváření a správa výzev
- `editor` .... Editor projektu
- `projects` .... Vytváření a správa projektů
- `shared` .... Sdílené komponenty, direktivy, služby a modely
- `user` .... Přihlašování uživatele

■ **Obrázek 6.5** Složky s moduly nového řeření

Alternativou rozdělení tříd na moduly podle funkcionality by mohlo být rozdělení podle typu třídy, tedy například na služby, komponenty, direktivy, atd. Rozdělení podle funkcionality je však lepší v tom, že častěji budou společně používány třídy odpovídající jedné funkcionalitě než jednomu typu a mít je blízko u sebe je tak výhodné. Dále také typů tříd je menší množství než potenciálně funkcionalit a rychle by tak jejich složky mohli začít být nepřehledné.

6.2.2 Celkový pohled

Vstupní bod celé aplikace je soubor `main.ts`, zde je zavedena hlavní komponenta `AppComponent`, která existuje po celou dobu spuštění aplikace a tvoří layout pro všechny stránky. Každá stránka je pak implementována jako Angular komponenta v příslušném modulu. Uživatelské rozhraní stránky je tvořeno pomocí Angular komponent, direktiv a pipe umístěných buďto ve stejném modulu a nebo v modulu `shared`. Business logika je umístěna ve službách, v případě složitějších stránek může stránka mít i svoji vlastní službu, která se stará o její stav a logiku. Tyto služby pak

mohou komunikovat s dalšími službami z modulu `shared` například pro komunikaci s REST API serveru a nebo třeba přímo s kompilátorem či interpretem jazyka Karel, které jsou v odděleném projektu. Modul `shared` pak také obsahuje modely reprezentující doménové entity aplikace a další logiku, jako je autorizace nebo zpracování chyb.

6.2.3 Správa stavu

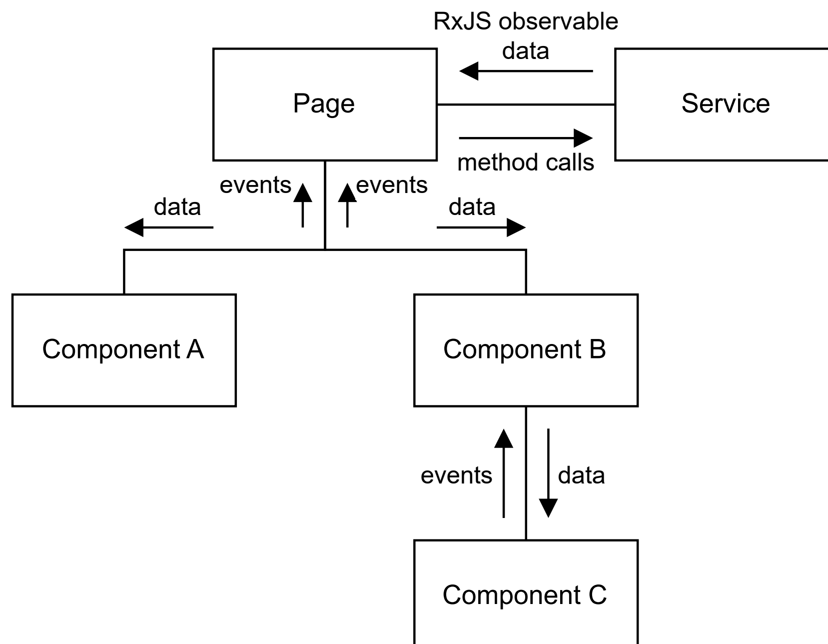
Jako u každé takovéto aplikace je základem její stav. Na základě stavu je vykreslováno uživatelské rozhraní a uživatelé svými akcemi tento stav mění. Ohledně stavu je třeba při návrhu zodpovědět otázky, jako kde bude stav uchováván, jakým způsobem bude měněn a jak bude předáván do ostatních částí aplikace.

Při návrhu této aplikace byly jako odpovědi na tyto otázky uplatněny následující principy:

Single source of truth Každá část stavu bude mít pouze jedno autoritativní místo, které za ni bude zodpovídat a kde bude uložena. To bude také jediné místo, kde tento stav půjde změnit. Díky tomuto principu se předejde problémům se synchronizací stavu.

One-way data flow Ve stromu komponent bude stav předáván směrem dolů. Nahoru budou naopak předávány události, na základě kterých pak bude vlastník stavu stav aktualizovat. Tento princip zjednodušuje tok stavu v aplikaci a v důsledku jeho pochopení, neboť data, která určují výsledný stav komponenty, mohou v komponentovém stromu přijít pouze ze shora. Při určení stavu komponenty v jeden okamžik tak lze ignorovat komponenty níže ve stromu. Přesto však i komponenty níže musí mít možnost stav ovlivňovat, proto směrem nahoru prostupují události. Tento princip ilustruje obrázek 6.6.

Immutabilita Stav bude neměnný. Bude tak možné pouze vyměnit starý stav za nový jako jeden celek. To ulehčuje detekci změn a dále také pomáhá zajistit, aby komponenty nemohli data, která do nich přijdou, přímo měnit (one-way data flow).



■ **Obrázek 6.6** One-way data flow

Aby byl zachován princip single source of truth a one-way data flow, je přirozené stav umístit nad všechny komponenty, které ho používají. V tomto případě tak bude stav umístěn ve službách, které se starají zároveň o logiku, a nebo v jednoduchých případech v komponentě stránky.

Pro tok dat mezi službami a do komponenty stránky byla použita knihovna pro reaktivní programování RxJS [69]. Jejím základem jsou tzv. observables. Observable reprezentuje synchronní nebo asynchronní proud dat. Tyto observables je pak možné pomocí různých operátorů skládat a reagovat na nová data, která z nich přijdou. Použití této knihovny je izolováno do služeb a stránek, běžné komponenty už na ní nezávisí. Celkový navržený způsob práce se stavem znázorňuje obrázek 6.6.

Ještě je dobré zmínit, že pro správu stavu existují i specializované knihovny. Jejich společným znakem je, že jsou poměrně robustní a stanovují jasné postupy a doporučení tykající se práce se stavem. S tím ale jde ruku v ruce i jejich velká komplexita. Pro aplikaci tohoto rozsahu bylo však zváženo, že nevýhody převyšují výhody, a byl použit výše popsaný jednodušší způsob. Nicméně například one-way data flow je principem používaným u obou těchto možností.

6.2.4 Routování stránek

Jelikož se jedná o single page aplikaci, tak není možné pro přepínání stránek na základě URL adresy spoléhat na prohlížeč a je potřeba použít jiné routovací řešení. Konkrétně byla použita část Angular frameworku Angular Router.

V HTML šabloně hlavní komponenty `AppComponent` je umístěn element `<router-outlet>`, který označuje, kam má router komponentu s aktuální stránkou umístit. Hlavní komponenta pak ještě obsahuje společné ovládací prvky každé stránky, jako je například horní lišta s navigací nebo postranní vysouvací menu, v případě rozložení pro úzké rozlišení.

Pravidla pro to, které URL adrese odpovídá jaká stránka, jsou umístěna v hierarchických souborech `routes.ts`. Kořen tvoří soubor `app.routes.ts` a z něho jsou pak odkazovány soubory s pravidly pro konkrétní moduly. Této hierarchické struktuře odpovídá i struktura URL adresy. Každé pravidlo se v základu skládá ze vzoru URL adresy a komponenty stránky, která se na ni má namapovat. Pravidla jsou testována postupně, na konec je tak umístěno pravidlo se vzorem pro jakoukoliv URL adresu se stránkou oznamující uživateli, že stránka, kterou hledal, neexistuje.

```
export const appRoutes: Routes = [
    {
        path: "",
        redirectTo: "/editor",
        pathMatch: "full"
    },
    {
        path: "editor",
        loadChildren: () => import("./editor/editor.routes")
            .then(m => m.editorRoutes)
    },
    ...
    {
        path: "**",
        component: NotFoundPageComponent
    }
];
```

■ **Výpis kódu 6.7** Ukázka routovacích pravidel ze souboru `app.routes.ts`

V těchto pravidlech je také řešena autorizace stránek. U každého pravidla lze specifikovat tzv. route guards, což jsou funkce, kterými k němu lze omezit přístup. Jedním z typů takových route guards je „CanActivate“, který určuje, jestli pravidlo aktuální uživatel může použít.

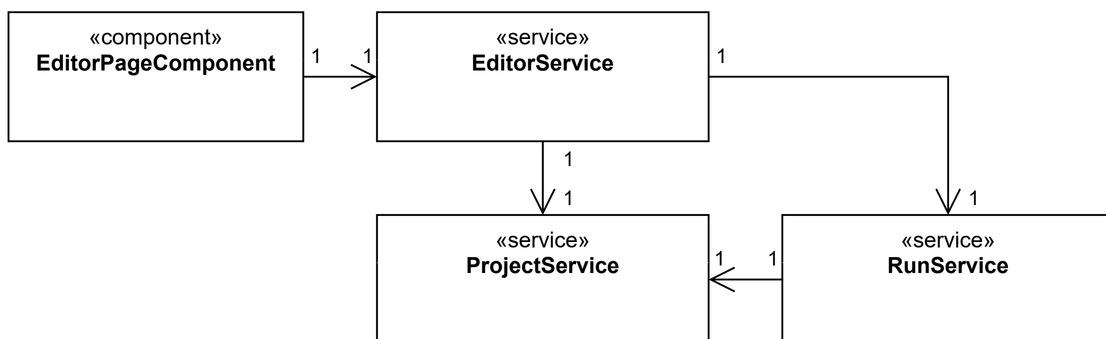
Ty byly v této aplikaci implementovány dva, `anonymousCanActivate` umožňuje přístup pouze nepřihlášeným uživatelům a `authenticatedCanActivate` zase jen těm přihlášeným, u toho lze také specifikovat navíc ještě požadování administrátorských oprávnění.

Aby všechny části aplikace nemusely být do prohlížeče staženy najednou, tak je v těchto souborech využita technika lazy-loadingu. To znamená, že jednotlivé moduly jsou routerem načteny až ve chvíli, kdy jsou skutečně potřeba. Konfiguraci routovacích pravidel ukazuje výpis kódu 6.7.

6.2.5 Editor

Nejkomplexnější stránkou aplikace je editor. Nachází se ve vlastním modulu ve složce `editor`. Na rozdíl od původního řešení, kde byla veškerá funkcionalita editoru umístěna přímo v komponentě jeho stránky, jsou nově funkce editoru rozděleny do tří služeb mimo kód komponenty. Každá z těchto služeb si spravuje svůj vlastní stav a dohromady jejich stav tvoří celkový stav editoru. Pro snadnou reakci na jeho změny je z nich, jako v každé jiné službě, zveřejněn pomocí observables knihovny RxJS. Všechny služby a jejich závislosti popisuje diagram na obrázku 6.7.

- Služba `ProjectEditorService` se stará o samotný projekt a jeho editaci. To zahrnuje například vytváření souborů a úpravu kódu. Tvoří také rozhraní pro funkce kompilátoru, jako je kontrola chyb nebo napovídání.
- Služba `RunService` má na starost spouštění programu. Ta zase tvoří rozhraní pro interpret a spravuje jeho stav. Řeší také nastavování breakpointů a poskytuje vysokoúrovňové rozhraní pro debugger v interpretu.
- Služba `EditorService` zaobaluje obě předchozí služby a navíc přes služby modulu `shared` komunikuje se serverem. Tvoří tak fasádu pro použití v komponentě editoru (facade pattern).



Obrázek 6.7 Služby používané editorem

Aby mohl editor při psaní napovídat symboly a poskytovat další editační služby, potřebuje mít neustále k dispozici aktuální syntaktický strom. Po každém napsaném znaku je tak kód v otevřeném souboru znovu přeparsován. To se ukázalo pro všechna zamýšlená použití jazyka Karel jako dostatečně rychlé. Pokud by nebylo, dal by se implementovat inkrementální parser, jako často musejí mít editory běžných jazyků. Kontrola syntaktických chyb by nicméně po každém napsaném znaku působila příliš rušivě, a je tak prováděna s určitou časovou prodlevou.

Z hlediska uživatelského rozhraní se komponenta stránky stará jen o celkové rozložení, uživatelská rozhraní jednotlivých panelů, jako je prohlížeč souborů, editor kódu nebo editor města, jsou už umístěna ve vlastních komponentách. Implementace posledních dvou je popsána v následujících kapitolách.

6.2.6 Editor kódu

Pro samotný editor kódu byla v původním řešení použita knihovna Monaco Editor [15]. Kvůli její zásadní nevýhodě, kterou je nepoužitelnost na mobilních zařízeních, bylo však nutné se poohlédnout po jiném řešení. Tím se stala knihovna CodeMirror [70].

První překážkou bylo, že architektura této knihovny je velmi odlišná od většiny takovýchto editorů, včetně toho použitého původně. Stav editoru je zcela určen immutabilním objektem, který je možné aktualizovat pouze pomocí tzv. transakcí. Editor je také navržen s velkým důrazem na rozšiřitelnost a modularitu. V základu není o mnoho chytřejší než běžné textové pole a až mnoho malých rozšíření dohromady z něj dělá pokročilý editor kódu. Pomocí rozšíření se přidávají funkcionality od těch základních, jako je undo & redo, číslování řádků nebo grafický styl editoru, až k těm pokročilejším jako barevné zvýrazňování, napovídání a podpora jazyků.

Pro integraci knihovny do aplikace byla vytvořena komponenta `CodeEditorComponent`, která knihovnu zaobaluje a vytváří její rozhraní pro použití v Angularu. Stará se tak o inicializaci editoru a synchronizaci svých vstupů s jeho stavem. Jednotlivé funkce editoru specifické této aplikaci byly implementovány jako rozšíření v samostatných TypeScriptových souborech. Všechny implementovaná rozšíření shrnuje tabulka 6.1, dále je pak popsána implementace těch nejzajímavějších.

`karel-language`	Barevné zvýrazňování a automatické odsazování.
`code-completion`	Napovídání.
`error-highlighting`	Podtrhávání chyb.
`current-range-highlighting`	Zvýrazňování dalšího vykonaného příkazu.
`breakpoints`	Umisťování breakpointů.
`application-theme`	Rozložení a tvary v souladu s designem aplikace.
`application-theme-light`	Barvy světlého režimu v souladu s designem aplikace.
`application-theme-dark`	Barvy tmavého režimu v souladu s designem aplikace.
`tab-keymap`	Odsazování pomocí klávesy Tab.

■ **Tabulka 6.1** Implementovaná rozšíření CodeMirror editoru

Barevné zvýrazňování První implementovanou funkcí je barevné zvýrazňování. K tomu je nutné editoru nějak poskytnout informaci o významech jednotlivých částí zdrojového kódu. Je tedy potřeba zdrojový kód průběžně parsovat a ze syntaktického stromu předávat tyto informace editoru. Přesto, že jazyk Karel už parser má, tak bylo vhodnější pro tento účel vytvořit nový pomocí knihovny Lezer [71] od autora CodeMirroru. Výhoda je, že v ní vytvořený parser je přímo uzpůsoben pro použití v editoru. Vytvoření parseru v této knihovně spočívalo v definici gramatiky pomocí speciálního jazyka a její předání nástroji knihovny, který z ní vygeneroval data pro vytvoření samotného parseru.

Při vytváření gramatiky se nejprve definují tokeny (výpis kódu 6.8). To jsou například klíčová slova, identifikátory nebo čísla. Využívají se k tomu regulární výrazy.

```
@tokens {
  Number { "0" | $[1-9] $[0-9]* }
  Operator { "is" | "not" }
  identifier { $[a-zA-Z] $[a-zA-Z0-9]* }
  SingleLineComment { "//" ![\n\r]* }
  endOfLine { "\r\n" | "\r" | "\n" }
  space { $[ ]+ }
}
```

■ **Výpis kódu 6.8** Ukázka tokenů Lezer gramatiky pro jazyk Karel

Z tokenů jsou poté poskládány neterminální symboly reprezentující jednotlivé konstrukce jazyka, jako jsou programy, podmínky a cykly. To ukazuje výpis kódu 6.9.

```
@top CompilationUnit {
    Program*
}

Program {
  kw<"program"> identifier
  Block
}

Block {
  statement*
  kw<"end">
}

statement {
  If | While | Repeat | Call
}

If {
  kw<"if"> Operator Call
  Block
  (kw<"else"> Block)?
}
```

■ **Výpis kódu 6.9** Ukázka neterminálních symbolů Lezer gramatiky pro jazyk Karel

Pro integraci do editoru bylo ještě potřeba namapovat jednotlivé symboly v gramatice na tagy pro stylování, kterým rozumí editor.

Automatické odsazování Druhou funkcí bylo automatické odsazování. Zde se ukázala další výhoda použití Lezeru, kdy stačilo pro vybrané symboly gramatiky jen definovat způsob odsazení a o zbytek se již editor postaral sám. Tato funkce spolu s barevným zvýrazňováním byla zabalena jako vlastní CodeMirror rozšíření a umístěna do TypeScriptového souboru `karel-language.ts`.

Napovídání Dále bylo třeba přidat napovídání. Logika pro získání seznamu nabízených položek na daném umístění v kódu je součástí `LanguageService` v kompilátoru jazyka, šlo tak hlavně o její propojení s editorem. Editor kódu nezná celý kontext kompilace, ale pouze obsah otevřeného souboru, napovídané položky tedy musí přijít zvenčí.

Do rozhraní komponenty editoru tak byla přidána vlastnost `completionItemsProvider`. Ta umožňuje zvnějšku předat funkci, která na základě pozice v kódu položky vrátí. CodeMirror pak nabízí rozšíření `autocompletion`, které přijímá podobnou funkci a volá ji po napsání určitých znaků, nebo po uživatelově explicitním vyžádání. Zároveň zajišťuje uživatelské rozhraní seznamu, do kterého vrácené položky vkládá. Propojení těchto dvou funkcí je opět realizováno jako rozšíření a umístěno v souboru `code-completion.ts`. Jeho hlavním úkolem je převést pozice v kódu z jednodimenzionálního indexu znaku, který používá editor, na systém řádek a sloupec, který používá kompilátor.

Podtrhávání chyb Podobná byla implementace podtrhávání chyb. Základ zobrazování je opět již součástí knihovny jako rozšíření. Pro zobrazení chyb pak stačí jen editoru odeslat transakci s jejich seznamem. Chyby opět musí přijít zvnějšku komponenty, do jejího rozhraní tak byla přidána vlastnost `errors` umožňující je editoru předat. V souboru `error-highlighting.ts` bylo vytvořeno rozšíření, jehož úkolem je mapovat chyby ve formátu kompilátoru Karla na formát, který používá zmíněné rozšíření. Opět šlo hlavně o převod pozic v kódu.

■ **Obrázek 6.8** Výsledný editor kódu

Zvýraznění aktuálního příkazu Poslední dvě funkce souvisí s požadavkem na ladění programů. V první řadě editor musí umět zvýrazňovat další vykonaný příkaz. K tomu byly použity tzv. mark dekorace editoru, pomocí kterých je možné přiřadit určitým rozsahům kódu vlastní CSS třídu.

Breakpointy Pro ladění je také potřeba umisťování breakpointů na jednotlivé řádky. Knihovna nabízí obecnější funkcionalitu, která umožňuje vedle řádků umisťovat jakékoliv HTML elementy. Dokumentace editoru ukazuje jak pomocí ní implementovat breakpointy. Zjednodušeně řečeno se jednalo jen o přidání pole do stavu editoru pro uchování zobrazovaných breakpointů a jeho změnu ve chvíli, kdy uživatel klikne vedle řádku s cílem umístit breakpoint.

Výsledné rozhraní editoru je vidět na výpisu kódu 6.10. Celý editor pak ukazuje obrázek 6.8.

```
@Input()
code = "";
@Input()
readonly = false;
@Input()
errors: readonly Error[] = [];
@Input()
currentRange: LineTextRange | null = null;
@Input()
breakpoints: readonly number[] = [];
@Input()
completionItemsProvider: (line: ..., column: ...) => CompletionItem[]...
@Output()
codeChange = new EventEmitter<string>();
@Output()
breakpointsChange = new EventEmitter<readonly number[]>();
```

■ **Výpis kódu 6.10** Výsledné veřejné rozhraní komponenty editoru

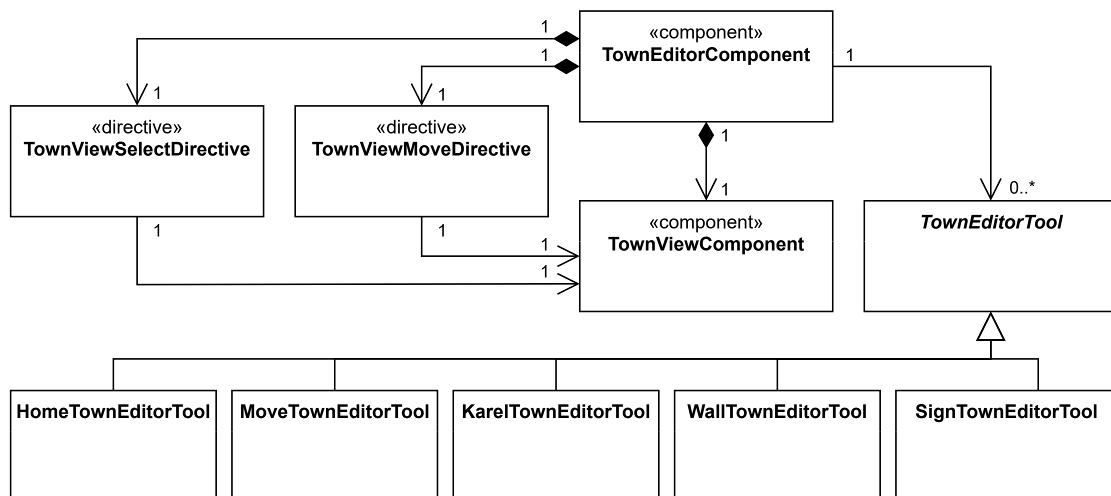
6.2.7 Editor města

Tak jako v původním řešení, i zde je pro vykreslování města použit HTML element `canvas`, přes jehož JavaScriptové rozhraní jsou vykreslovány jednotlivé dlaždice města a další objekty.

Uživatelské rozhraní editoru je implementováno modulárně. Skládá se ze dvou komponent a direktiv. Komponenta `TownViewComponent` pouze zobrazuje město podle předané kamery. Pohyb kamerou a přiblížení či oddálení jejího pohledu je vyčleněn do direktivy `TownViewMoveDirective`. V direktivě `TownViewSelectDirective` je pak umístěna funkcionalita obdélníkového výběru dlaždic města. Celý editor je sestaven v komponentě `TownEditorComponent`, která ke komponentě pro zobrazení města připojuje obě direktivy a navíc přidává uživatelské rozhraní pro editaci.

Stejně bylo UI rozděleno i v původním řešení. Protože se to ukázalo být velmi flexibilní, tak bylo takto rozděleno i v tomto novém. Samotná implementace těchto tříd ovšem byla nedostatečná. Byly tedy vytvořeny znovu.

Hlavním nedostatkem bylo, že implementace nástrojů pro editaci se nacházela přímo v kódu komponenty editoru a použití správného nástroje řídila sekvence podmínek. Nově byla pro každý nástroj vytvořena vlastní třída dědící ze třídy `TownEditorTool`. Ta obsahuje jednu veřejnou metodu určenou pro volání z editoru v případě použití daného nástroje. V parametrech přijímá editované město a vybrané dlaždice. Editor tak nově obsahuje jen pole dostupných nástrojů a referenci na ten, který je aktuálně vybraný. Ve chvíli, kdy z direktivy `TownViewSelectDirective` přijde událost o uživatelském výběru dlaždic, editor jen zavolá popsanou metodu na aktuálním nástroji. Díky polymorfismu je zajištěno volání správné implementace. Pro úplnost je ještě dobré zmínit, že tento jednoduchý výběr chování na základě aktuálního stavu má i svůj vlastní název, říká se mu state pattern.



■ **Obrázek 6.9** Diagram tříd uživatelského rozhraní editoru města

Výhodou vyčlenění logiky editoru do těchto několika tříd, komponent a direktiv je její nezávislost na komponentě editoru, z té opět plyne lepší oddělení odpovědností a testovatelnost. Celé řešení je včetně závislostí jednotlivých tříd ještě popsáno diagramem na obrázku 6.9.

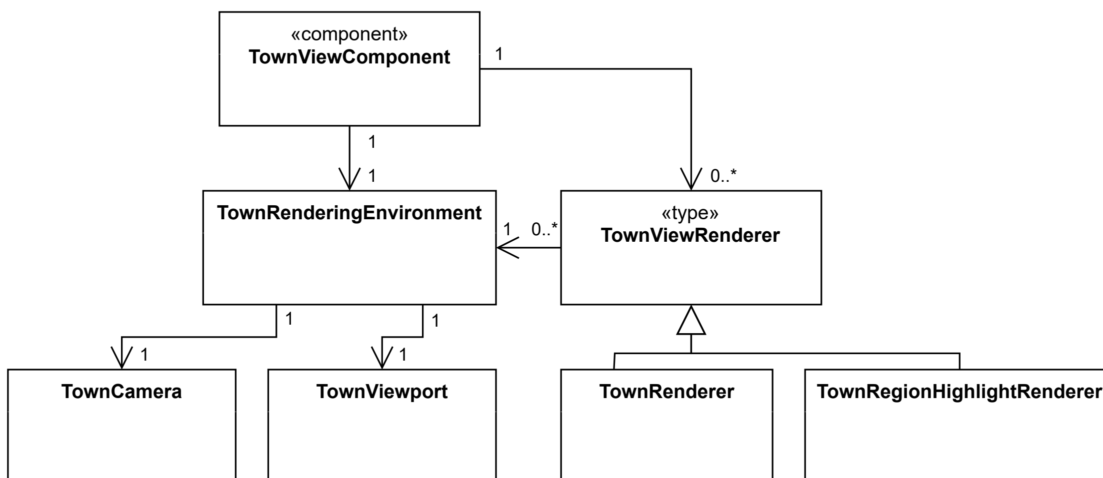
Implementace vykreslování a vlastnosti kamery byly původně opět neoptimálně umístěny přímo v kódu komponenty `TownViewComponent`. Ta tak měla zodpovědnost nejen za zobrazení vykresleného města, ale i za jeho samotné vykreslování, načítání obrázků a správu kamery. To bylo tak zase odděleno do vlastních tříd. O vykreslování města a načítání obrázků se stará `TownRenderer`. K tomu používá předanou instanci třídy `TownRenderingEnvironment`, která reprezentuje prostředí, ve kterém je město vykreslováno, a poskytuje metody pro převod mezi souřadnicemi města v dlaždicích a souřadnicemi komponenty na obrazovce v pixelech. Toto prostředí

je složeno ze tříd `TownCamera`, která reprezentuje virtuální kameru na město, a `TownViewport` definující obdélník na obrazovce, do kterého je město vykreslováno. Dále je potřeba vykreslovat také poloprůhledný obdélník zvýrazňující aktuální výběr, o to se stará `TownRegionHighlightRenderer`. Jeho vykreslení ukazuje výpis kódu 6.11, je na něm vidět i použití `TownRenderingEnvironment`.

```
const pixelX = environment.tileXToPixelX(region.x);
const pixelY = environment.tileYToPixelY(region.y);
const pixelWidth = region.width * environment.tilePixelSize;
const pixelHeight = region.height * environment.tilePixelSize;
context.fillStyle = "rgb(0, 0, 0, 0.3)";
context.fillRect(pixelX, pixelY, pixelWidth, pixelHeight);
```

■ **Výpis kódu 6.11** Vykreslení obdélníku pro zvýraznění vybraných dlaždic města

Dalším nedostatkem nalezeným v části analýzy bylo příliš časté vykreslování. Jak bude vysvětleno později, tak město není immutabilní a pro informaci o jeho změně tak k němu musela být přidána událost. K té se komponenta pro vykreslení zaregistruje a zajistí překreslení jen v nutných případech. Všechny třídy pro vykreslování shrnuje diagram na obrázku 6.10.



■ **Obrázek 6.10** Diagram tříd pro vykreslování města

Uživatelské rozhraní bylo původně navrženo pouze pro použití s myší a některé funkcionality, jako je pohyb kamery, přiblížení či výběr, nebyly na dotykové obrazovce vůbec dostupné. Jako řešení byla mimo jiné v části návrhu navržena dotyková gesta pro posun a přiblížení. Ta byla implementována ručně pomocí pointer events. To jsou události prohlížeče, které zobecňují a sjednocují události pro myš a dotyk. Implementaci toho pro přiblížení ukazuje výpis kódu 6.12.

```
if (this.pointersDown.size === 2) {
    const [[, prevFirst], [, prevSecond]] = this.pointersDown.entries();
    const prevDist = Vector.calculateDistance(prevFirst, prevSecond);
    this.pointersDown.set(event.pointerId, new Vector(..., ...));
    const [[, first], [, second]] = this.pointersDown.entries();
    const dist = Vector.calculateDistance(first, second);

    const centerX = (first.x + second.x) / 2;
    const centerY = (first.y + second.y) / 2;
    const zoom = dist / prevDist;
    this.zoom(centerX, centerY, zoom);
}
```

■ **Výpis kódu 6.12** Ukázka implementace dotykového gesta pro přiblížení

Výsledné uživatelské rozhraní editoru města ukazuje obrázek 6.11.

Obrázek 6.11 Výsledný editor města

6.2.8 Markdown

Pro umožnění formátování textu v popisu zadání výzvy byl v návrhové části zvolen značkovací jazyk Markdown. Prohlížeče však Markdown neumí zobrazit, a je tak potřeba ho převést do jazyka HTML. K tomu byla použita knihovna markdown-it [72]. Té stačí jen předat Markdown kód a ona vrátí odpovídající HTML.

Volání této knihovny bylo zapouzdřeno ve službě `MarkdownService`. Pro snazší použití v HTML šablonách byla ještě vytvořena komponenta `MarkdownViewComponent`. Té se předá Markdown kód a ona se postará o jeho vykreslení i se správnými styly. Použití této komponenty ukazuje výpis kódu 6.13.

```
<app-markdown-view [source]="challenge.description"></app-markdown-view>
```

Výpis kódu 6.13 Ukázka použití vytvořené Markdown komponenty

6.2.9 Přihlašování uživatelů

Jak již bylo uvedeno při návrhu v kapitole 5.6, tak pro přihlašování uživatelů byla použita služba Firebase Authentication. Ta pro implementaci poskytuje JavaScriptovou knihovnu řešící několik problémů souvisejících s přihlašováním uživatelů. V první řadě knihovna zaobaluje samotnou komunikaci s Firebase servery. Dále se také stará o relaci aktuálně přihlášeného uživatele a její persistenci při zavření karty prohlížeče. Její další výhodou je, že obsahuje jednotné rozhraní pro přihlašování pomocí různých externích služeb.

V aplikaci není tato knihovna použita přímo, ale využívá se její wrapper AngularFire [73], který zjednodušuje její použití v prostředí Angularu. Tato knihovna se pak používá ve třídě `SignInService`, která ji odstiňuje od zbytku aplikace. Její rozhraní ukazuje výpis kódu 6.14.

Při vývoji by nebylo rozumné připojovat se přímo k Firebase platformě s reálnými daty. Platforma jako řešení nabízí emulátor a při vývoji je tak místo ní použit ten.

```
export class SignInService {
    readonly currentUserToken$ = ...;
    readonly currentUser$ = ...;
    async signInWithGoogle(): Promise<boolean> { ... }
    async signOut(): Promise<void> { ... }
}
```

■ **Výpis kódu 6.14** Veřejné rozhraní služby pro přihlašování

6.2.10 Komunikace se serverem

Komunikaci se serverem zajišťují služby odpovídající jednotlivým zdrojům REST API umístěné v modulu `shared`. Každý požadavek (kromě autentizace) musí být autorizován. K tomu je využíván tzv. HTTP Interceptor z frameworku Angular. To je funkce, která může zachytávat a upravovat všechny požadavky odeslané přes Angular HTTP klient. Byl tak vytvořen interceptor `TokenInterceptor` (výpis kódu 6.15), který do každého požadavku automaticky přidá autorizační token přihlášeného uživatele získaný od služby `SignInService`.

Při otevření aplikace je Firebase knihovnou obnoven dříve přihlášený uživatel. Problém však je, že toto obnovení je asynchronní a požadavky jsou tak často odeslány dříve, než je známý uživatelův token. Toto je jeden z problémů, který elegantně řeší použitá knihovna RxJS. Token uživatele je ze služby `SignInService` zveřejněn jako asynchronní RxJS observable. Objeví se v něm tak až ve chvíli, kdy je získán, a není třeba používat nějakou dočasnou hodnotu. Díky tomu je požadavek v interceptoru, který také vrací observable, automaticky pozdržen do doby, než je token znám.

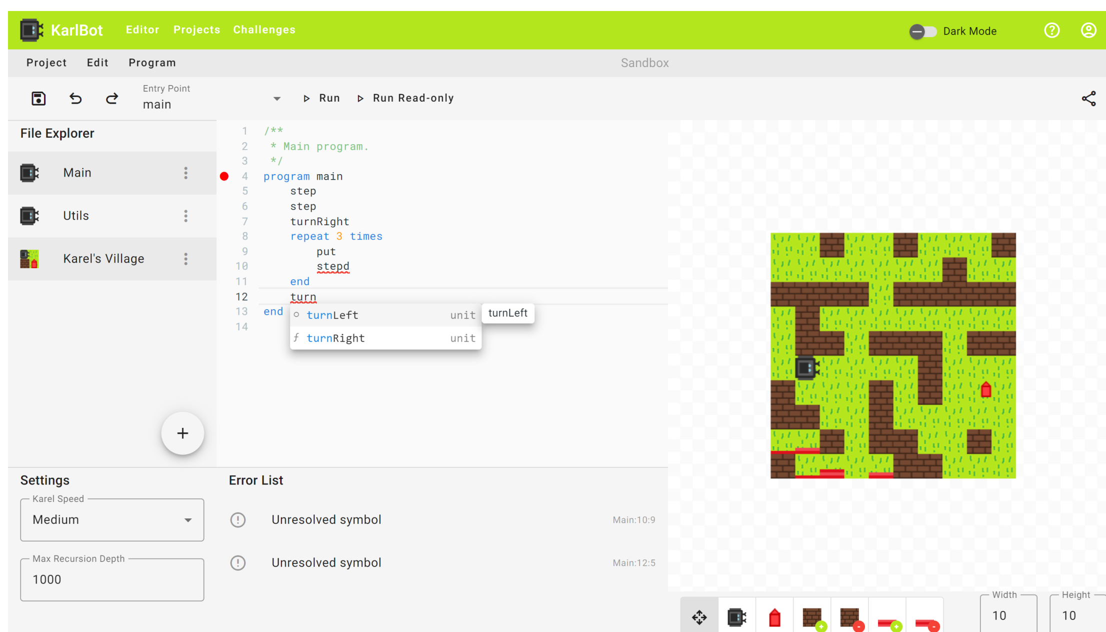
```
export function TokenInterceptor(request: ..., next: ...) {
    if (request.context.get(IS_ANONYMOUS_ENDPOINT))
        return next(request);

    const signInService = inject(SignInService);
    return signInService.currentUserToken$.pipe(
        take(1),
        switchMap(t => {
            if (t === null)
                return next(request);
            else {
                const requestWithToken = request.clone({
                    headers: request.headers
                        .set("Authorization", `Bearer ${t}`)
                });
                return next(requestWithToken);
            }
        })
    );
}
```

■ **Výpis kódu 6.15** HTTP interceptor přidávající do požadavků autorizační token

Odeslání požadavku na server a získání odpovědi nějakou dobu trvá, je to tedy dobré uživateli vizuálně indikovat. Aby to každá komponenta nemusela řešit zvlášť, tak byl opět vytvořen interceptor. Při odeslání požadavku interceptor zaznamená, že má být zobrazen načítací indikátor a po jeho skončení ho zase odvolá. Pro správu stavu načítání byla vytvořena služba `LoadingService`. Na základě té pak komponenta `GlobalLoaderComponent` přes obrazovku zobrazuje načítací indikátor.

6.2.11 Vzhled

Pro vytvoření uživatelského rozhraní byl nově zvolen Material Design [74]. To je návrhový systém od společnosti Google, který poskytuje pravidla a doporučení pro různé aspekty uživatelského rozhraní jako jsou rozložení, barvy, tvary nebo typografie. Pro jeho implementaci aplikace používá komponentovou knihovnu Angular Material [75]. Její součástí je velké množství Angular komponent vytvořených podle Material Design specifikace verze 2. Díky tomu tak také na rozdíl od starého řešení není nutné ručně implementovat obecné UI komponenty, jako třeba tlačítka, menu nebo dialogy. Jsou používány i ikony, které jsou součástí Material Designu. Jedna z výsledných stránek aplikace je ukázána na obrázku 6.12.



■ **Obrázek 6.12** Stránka editoru projektu výsledné aplikace ve světlém režimu

Podle požadavků musí mít aplikace i tmavý režim. Knihovna Angular Material už přichází s jeho podporou, a tak implementace nebyla příliš obtížná. Zvlášť musel být nastylován jen editor kódu a pár dalších prvků UI. U editoru kódu to bylo provedeno opět pomocí CodeMirror rozšíření. Ostatním prvkům UI byly barvy aktuálního režimu předány přes CSS proměnné. Správu aktuálního barevného režimu zajišťuje služba `ColorThemeService`. Ta se stará také o jeho ukládání do local storage a získání preferovaného režimu z nastavení prohlížeče. Na základě této služby nakonec hlavní komponenta nastaví HTML elementu `body` správnou CSS třídu s barvami.

Uživatelské rozhraní je plně responzivní a použitelné na dotykových zařízeních. Pro aplikování různých stylů na základě aktuálního rozlišení byly použity CSS media queries. V některých případech to však nestačilo. Například na stránce s editorem je pro menší rozlišení potřeba měnit nejen styly, ale i strukturu stromu komponent. K tomu byl využit `BreakpointObserver` z knihovny Angular CDK [76], který umožňuje reagovat na změnu rozlišení i v kódu.

6.2.12 Dependency injection

Stejně jako na serverové části i zde se pro propojení tříd aplikace používá vzor dependency injection. Angular využívá hierarchickou dependency injection. To znamená, že lze závislosti konfigurovat jinak na úrovni jednotlivých modulů a HTML elementů. Lze tím také řídit jejich

životnost. To se zde využívá pro registraci služeb stránky editoru. Většina služeb je však zaregistrovaná v kořeni (ukazuje výpis kódu 6.16) a jsou tak dostupné všem třídám.

```
providers: [
    provideRouter(appRoutes),
    provideHttpClient(
        withInterceptors([TokenInterceptor, LoadingInterceptor])
    ),
    provideAnimations(),
    importProvidersFrom(
        provideFirebaseApp(() => initializeApp(environment.firebase)),
        provideAuth(() => ...),
        MatDialogModule,
        MatBottomSheetModule,
        MatSnackBarModule
    ),
    { provide: API_BASE_URL, useValue: environment.apiBaseURL },
    { provide: ErrorHandler, useClass: ApplicationErrorHandler }
]
```

■ **Výpis kódu 6.16** Ukázka konfigurace kořenového injectoru

6.3 Implementace jazyka Karel

Implementací jazyka Karel se rozumí kompilátor, interpret, standardní knihovna, město a formát souboru projektu jazyka Karel. Kromě absence testů a dokumentace v ní byly v průběhu analýzy nalezeny pouze menší nedostatky. Po jejich odstranění byla téměř celá převzata z původního řešení. O jejím celkovém fungování tedy stále platí to, co je uvedeno v analýze starého řešení. Bylo však třeba přidat některé nové funkcionality týkající se zejména ladění programů. O tom a o refaktoringu pojednává tato kapitola, o testování a dokumentaci pak kapitoly 7.4 a 8.1.

```
│ assembly............................Instrukce bajtkódu
├─ compiler...........................Kompilátor
│  ├─ code-generation.................Generování instrukcí
│  ├─ errors..........................Syntaktické chyby
│  ├─ language-service................Rozhraní kompilátoru pro editory
│  ├─ semantic-analysis...............Sémantická analýza a kontrola chyb
│  ├─ symbols.........................Symboly ze sémantické analýzy
│  ├─ syntax-analysis.................Syntaktická a lexikální analýza
│  └─ syntax-tree.....................Syntaktický strom
│     ├─ nodes........................Uzly syntaktického stromu
│     ├─ tokens.......................Tokeny syntaktického stromu
│     └─ trivia.......................Syntax trivia tokenů
├─ interpreter........................Interpret bajtkódu
├─ math...............................Matematické struktury
├─ project............................Projekt, jeho soubory a serializace
├─ standard-library...................Programy standardní knihovny
├─ interpreter........................Interpret bajtkódu
├─ text...............................Práce s textem
├─ town...............................Město robota
└─ utils..............................Privátní pomocné třídy a metody
```

■ **Obrázek 6.13** Nová adresářová struktura knihovny jazyka Karel

Jedním z problémů ve starém řešení bylo neoddělení knihovny jazyka Karel od zbytku aplikace. Nově tak byla celá umístěna do svého vlastního projektu nazvaného `karel`. Díky tomu je možné ji používat i samostatně. Kromě lepší struktury kódu se to v tuto chvíli hodí zejména pro hodnocení výzev na serveru. V rámci přesunu byla také kompletně upravena adresářová struktura tak, aby více odpovídala jednotlivým fázím kompilace a celkově byl snížen počet závislostí mezi adresáři. Novou adresářovou strukturu ukazuje obrázek 6.13 na předešlé stránce.

6.3.1 Kompilátor

Kvůli nově požadované podpoře krokování je nutné u spuštěného programu vědět, kde ve zdrojovém kódu se vykonávání právě nachází. To je však zkomplikováno překladem do bajtkódu. Je potřeba tedy mít nějaké mapování vygenerovaných instrukcí bajtkódu na textové rozsahy ve zdrojovém kódu. Stejně tak kvůli umisťování breakpointů na jednotlivé řádky je třeba mít i opačné mapování, minimálně z jednotlivých řádek na jim odpovídající instrukce.

Pro potřeby kontroly syntaktických chyb už je u uzlů syntaktického stromu snadno dostupná informace, v jakém rozsahu zdrojového kódu se nachází. Stačí tedy tuto informaci ještě nějakým způsobem předat k samotným instrukcím. To bylo přirozené udělat ve třídě `Emitter`, která tyto instrukce a celé sestavení obsahující všechny programy generuje. K jeho výstupu tak byla přidána nově vytvořená třída `SourceMap` (výpis kódu 6.17). To je jednoduchá datová struktura poskytující obousměrné mapování mezi instrukcemi a jejich textovými rozsahy. Instance této datové struktury je umístěna do výsledné třídy `Assembly` reprezentující vygenerované sestavení.

```
export class SourceMap {
    static create(instructionsWithRange: ...) { ... }
    getRangeByInstruction(instruction: Instruction): ... { ... }
    getInstructionsByLine(filePath: string, line: number): ... { ... }
}
```

■ **Výpis kódu 6.17** Veřejné rozhraní třídy `SourceMap`

Ne každé části zdrojového kódu odpovídá nějaká instrukce. Například hlavička programu nebo token „end“ označující konec bloku žádné instrukce negenerují. Přesto by se hodilo moci i na tyto části kódu umístit breakpoint a krokovat přes ně. To bylo vyřešeno stejně jako v mnoha jiných instrukčních sadách přidáním instrukce „no operation“ [77]. To je instrukce, která nic nedělá a slouží mimo jiné pro potřeby ladění. K těmto instrukcím jsou pak zmíněné části kódu přiřazeny.

6.3.2 Interpret

Interpret byl přepracován výrazně více. Oproti staré verzi musel navíc podporovat krokování a breakpointy. Kvůli krokování do něj byly přidány tři nové veřejné interpretační metody. Nyní tak pro interpretaci instrukcí obsahuje následující metody:

- `interpretAll` – Interpretuje všechny instrukce.
- `interpretSingle` – Interpretuje jednu instrukci.
- `interpretStepInto` – Interpretuje všechny instrukce odpovídající stejnému rozsahu ve zdrojovém kódu.
- `interpretStepOver` – Jako předchozí avšak s tím rozdílem, že pokud některá instrukce obsahuje volání programu, tak se v něm nezastaví a vykoná ho najednou.
- `interpretStepOut` – Interpretuje instrukce až do vrácení se z aktuálního programu.

Všechny tyto metody přijímají token pro zastavení a vracejí třídu typu `InterpretResult` reprezentující výsledek. Implementace nových metod byla přímočará, jelikož interpret má referenci na celé sestavení, stačilo pouze využít v něm dostupnou třídu `SourceMap` pro získání rozsahu zdrojového kódu odpovídajícího aktuální instrukci.

Dále byla do interpretu přidána metoda pro nastavení breakpointů na konkrétní instrukce. Pro oznámení, že program narazil na breakpoint, byl vytvořen nový typ výsledku. Ty byly navíc také refaktorovány a z enumů převedeny na třídy. Možné výsledky tak nyní jsou:

- `NormalInterpretResult` – Interpretace skončila standardním způsobem.
- `StopInterpretResult` – Interpretace byla zastavena pomocí předaného tokenu.
- `ExceptionInterpretResult` – Při interpretaci nastala výjimka jazyka Karel.
- `BreakpointInterpretResult` – Interpretace narazila na breakpoint.

Vlastností původního interpretu bylo, že pokud z instrukcí nebyly volány žádné asynchronní externí programy, tak metody pro interpretaci vykonaly všechny instrukce najednou. To bylo nejen problém u programů, které trvají dlouho, ale zásadní problém to byl u programů, které nikdy neskončí. V takovém případě tak nikdy neskončily ani metody pro interpretaci, což vzhledem k jednovláknovému vykonávání JavaScriptu v prohlížeči vedlo k zaseknutí celé aplikace a nakonec až k jejímu ukončení ze strany prohlížeče.

To bylo vyřešeno pomocí kooperativního multitaskingu, kdy interpret při vykonávání Karlova programu jednou za čas předá kontrolu zpět prohlížeči, nechá ho zpracovat uživatelské akce a poté opět pokračuje v interpretaci instrukcí. Technicky je to řešeno pomocí dvou vlastností přidaných do interpretu. Vlastnost `interpretedInstructionCountBeforeYield` určuje zda a po kolika vykonaných instrukcích má být vrácení kontroly uskutečněno. Vlastnost `yieldFunction` potom určuje samotnou asynchronní funkci, která předání zajistí. V prohlížeči se může jednat například o nastavení časovače na nějakou malou nebo dokonce nulovou dobu (výpis kódu 6.18), čímž se efektivně interpretace Karla v event loop prohlížeče [78] přesune za uživatelské akce čekající na zpracování.

```
interpreter.yieldFunction = (stopToken: InterpretStopToken) => {
    return new Promise(resolve => {
        window.setTimeout(() => resolve(), 0);
    });
};
```

■ **Výpis kódu 6.18** Příklad funkce předávající kontrolu zpět prohlížeči

Bylo také zjednodušeno rozhraní třídy interpretu a zapouzdřen její stav, aby nemohl být nekontrolovaně měněn zvenčí. Stavem interpretu je například zásobník volání. Pro jeho zapouzdření mu bylo vytvořeno read-only rozhraní pod kterým je z interpretu veřejně vystaven.

6.3.3 Město

Na rozdíl od téměř všech ostatních datových tříd město ve starém řešení nebylo immutabilní. To bylo však z dobrých důvodů, neboť město je programem robota velmi často měněno a za účelem co nejlepšího výkonu spuštěných programů tak v tomto případě není rozumné při každé změně vytvářet nové. Bylo by sice možné některé části sdílet, jako to dělají jiné persistentní datové struktury, ale stále bude samozřejmě prostá mutace jedné proměnné o mnoho rychlejší a v tomto případě je tak toto nesystémové porušení immutability opodstatněné.

Nově byl však jako kompromis zvolen přístup dvou měst, jedno immutabilní a druhé mutabilní s metodami pro jednoduchý převod mezi nimi. V rámci neduplikování kódu bylo využito kompozice a implementace immutabilního města tak interně obsahuje instanci mutabilního, na kterou deleguje volání všech svých metod (obráceně to ze zjevných důvodů nešlo). V projektu je pak uloženo immutabilní město a programy standardní knihovny používají zase to mutabilní.

6.3.4 Ostatní změny

V ostatních částech knihovny byl proveden hlavně refaktoring a funkčnost zůstala stejná. Následuje ještě seznam těch nejdůležitějších změn:

- Kompilátor TypeScriptu byl přepnut do striktního režimu. V něm je programátor nucen specifikovat některé věci, které by jinak kompilátor odvodil automaticky. Díky tomu má kompilátor více informací a může tak být odhaleno více chyb už ve fázi kompilace.
- Některé třídy, které spolu úzce souvisely, byly přesunuty do jednoho modulu. Hlavní motivací pro tuto změnu bylo odstranění cyklických závislostí mezi moduly.
- Programy v sestavení byly udělány immutabilní.
- Sémantická analýza byla oddělena do vlastní třídy.

6.3.5 Ukázka použití

Pro shrnutí je tato kapitola opět zakončena ukázkou použití knihovny na výpisu kódu 6.19. Možno srovnat s ukázkou ze starého řešení na výpisu 3.3.

```
const sourceCode = ...;
const extProgramRefs = StandardLibrary.getProgramReferences();

const compilationUnit = CompilationUnitParser.parse(sourceCode, "File");
const compilation = new Compilation([compilationUnit], extProgramRefs);

const errors = Checker.check(compilation);
// ...Handle errors

const assembly = Emitter.emit(compilation);
const entryPoint = assembly.programs.find(p => p.name === "main")!;

const town = MutableTown.createEmpty(10, 10);
const extPrograms = StandardLibrary.getPrograms(town, () => 0);
const interpreter = new Interpreter(assembly, entryPoint, extPrograms);

await interpreter.interpretAll(new InterpretStopToken());
```

■ **Výpis kódu 6.19** Ukázka použití knihovny jazyka Karel v novém řešení

Kapitola 7
Testování

Nejběžnějším způsobem testování softwaru jsou automatizované testy. Jejich účelem je nejen zjistit, jestli aplikace funguje tak jak má ve chvíli, kdy je vytvořena, ale hlavně zajistit, aby fungovala i v budoucnu po dalším vývoji. Umožňují jednoduše ověřit, zda se některá již dříve funkční část aplikace nerozbila změnou kódu některé jiné části.

7.1 Typy testů

Testy se rozlišují zejména na základě toho, jakou část aplikace zahrnují. Při testování této aplikace byly použity následující druhy automatizovaných testů:

Unit testy Testují malé jednotky v kódu, jako jsou třeba třídy, metody nebo funkce, nezávisle na zbytku aplikace. Závislosti těchto jednotek se tak nahrazují za jejich testovací verzi. Jejich výhodou je malý rozsah kódu pro hledání chyb a rychlost spouštění.

Integrační testy Integrační testy už testují několik jednotek najednou a hlavně to, že fungují všechny dohromady.

End-to-end testy Testují software od začátku do konce tak, jak by s ním pracoval jeho uživatel. Dokáží z pohledu uživatele ověřit, že výslednou aplikaci lze skutečně používat.

Statická analýza Testuje na úrovni zdrojového kódu softwaru ještě před jeho spuštěním. Umí například odhalit problémy, jako jsou nepoužité proměnné nebo nedostupný kód.

7.2 Testování serverové části

Pro testování serveru byla použita knihovna NUnit [79]. Kde to bylo možné, tak byly využity unit testy a závislosti tříd nahrazeny pomocí knihovny Moq [80]. U testování repozitářů by však nebylo vhodné databázi něčím nahrazovat a byly tak místo unit testů použity integrační testy komunikující přímo s reálnou databází (ovšem ne produkční, ale testovací). Existují sice možnosti, jak by se šlo použití reálné databáze vyhnout, například In-Memory databáze knihovny Entity Framework Core nebo použít jinou jednodušší databázi, která ukládá data třeba do souboru. Problém u takového řešení ale je, že mezi těmito databázemi a tou použitou v produkci budou vždy nějaké rozdíly a testy by tak v mnoha případech nefungovali správně.

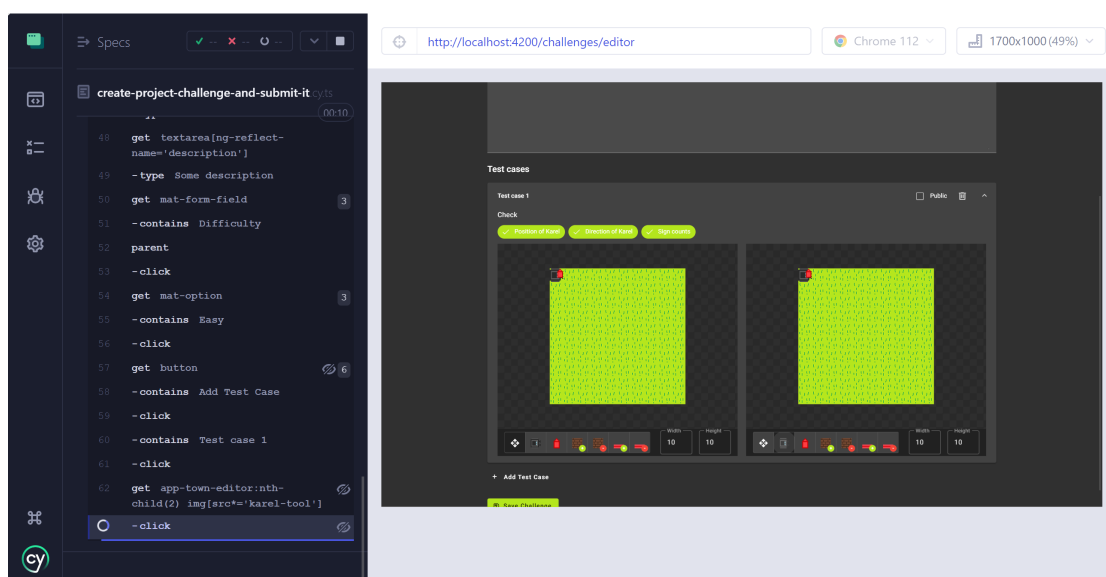
Z časových důvodů bylo vynecháno unit testování prezentační vrstvy, nicméně její funkčnost do jisté míry ověřuje jeden dále zmíněný druh testů.

7.3 Testování klientské části

Klientská část je specifická v tom, že obsahuje množství složitějších UI komponent, které nelze dostatečně spolehlivě otestovat jen za pomoci unit testů. To, že jednotlivé metody v kódu komponenty fungují správně, totiž ještě nezaručuje, že se ve výsledku celá komponenta vykreslí tak jak má, její ovládací prvky budou klikatelné a nebudou překryté jinými prvky. Unit testy ani nedokáží ověřit, že aplikace bude funkční jako celek i se serverovou částí a databází.

Navíc bylo tedy ještě sáhnuto k end-to-end testům v prohlížeči. Pro jejich tvorbu a spouštění byl zvolen testovací framework Cypress [81]. Serverová část aplikace ani databáze nejsou v testech pro co nejvěrohodnější výsledky nijak odstíněny a používají se jen jejich lokální testovací instance. Za testovací verzi muselo být kompletně nahrazeno pouze Firebase přihlášení Google účtem, neboť jeho automatizované použití bylo příliš složité a náchylné k chybám. Ani Cypress dokumentace nedoporučuje testovat přihlašování pomocí třetích stran z podobných důvodů.

Bylo zjištěno, že Firebase knihovna pro obnovení relace přihlášeného uživatele ukládá její stav do databáze IndexedDB v prohlížeči, a tak pro automatické přihlašování stačilo správně nastavit její obsah a nástrojem frameworku nahradit data vracená autentizačními HTTP endpointy za statická testovací data.



■ **Obrázek 7.1** Ukázka z průběhu end-to-end testu v uživatelském rozhraní frameworku Cypress

Byl vytvořen jeden testovací scénář pokrývající ty nejvíce zásadní funkce aplikace. Ukázka z jeho průběhu ve frameworku je na obrázku 7.1. Skládá se z následujících kroků:

1. Otevři pomocí URL adresy stránku aplikace s editorem projektu.
2. Napiš do editoru zdrojový kód programu, ovládající robota, aby došel ke své protější zdi.
3. Spusť vytvořený program.
4. Počkej než skončí.
5. Změň název projektu.
6. Ulož projekt na server.
7. Zkontroluj, že je projekt dostupný na stránce „Seznam projektů“.

8. Vytvoř novou výzvu s názvem, obtížností a popisem.
9. Přidej k výzvě testovací případ ověřující, že robot dojde k protější zdi.
10. Ulož výzvu.
11. Odešli vytvořený projekt jako řešení výzvy.
12. Zkontroluj, že je výzva úspěšně splněna.

Samotných unit testů je v klientské části z důvodu nedostatku času malé množství a jejich další doplnění je tak necháno pro budoucí rozšíření. Vytvořeny byly ve frameworku Jasmine [82].

V projektu byl také nastaven nástroj pro statickou analýzu ESLint [83]. Jednak pro odhalení běžných chyb, tak i pro udržení jednotnějšího stylu kódu. Umožňuje problémy nejen nalézt, ale v mnoha případech i automaticky opravit.

7.4 Testování knihovny jazyka Karel

Zde mimo jiné bylo nutné otestovat hlavně jednotlivé fáze kompilace. Hlavní část testů tvoří unit testy. Ty se však v některých případech ukázali jako ne úplně vhodné, zejména kvůli inherentní komplexnosti vstupních dat u některých tříd. Například pro třídu `Checker`, která provádí kontrolu chyb, by musel být v kódu vytvořen celý syntaktický strom, to znamená i se všemi bílými znaky, novými řádky a odsazeními. To se ukázalo jako velmi zdlouhavé a hlavně nepřehledné. Pro tyto třídy tak byly použity integrační testy. Místo syntaktického stromu je na vstup předáván už zdrojový kód a částečně testován i lexer a parser.

Unit a integrační testy ověřují, že jsou funkční jednotlivé části knihovny. To ještě nemusí znamenat, že bude fungovat i jako celek. Z toho důvodu bylo vytvořeno ještě několik end-to-end testů z pohledu uživatele knihovny – vývojáře, které testují celý proces od kompilace zdrojového kódu až po jeho spuštění. Tyto testy jsou nejblíže tomu, jak bude knihovna využívána. Skládají se vždy z kódu programu v jazyku Karel, vstupního města a očekávaného výstupního města po skončení programu. Pro všechny typy testů byl opět využit framework Jasmine.

7.5 Continuous integration

Běh testů nějakou dobu trvá a navíc je jejich spouštění často opomíjeno. Je tak vhodné automatizovat nejen testy samotné, ale i jejich spouštění. K tomu byl použit nástroj GitHub Actions platformy GitHub, kde je zdrojový kód projektu hostován. Pomocí tohoto nástroje byla vytvořena tzv. CI/CD pipeline mající za úkol po každém commitu aplikaci sestavit a otestovat. Díky tomu je možné chybný kód odhalit co nejdříve a hlavně zamezit jeho nasazení do produkce.

Mírnou komplikaci při spouštění testů v prostředí CI/CD pipeline přinesly integrační testy používající databázi. Pomohlo, že použitá databáze SQL Server je distribuována i jako Docker [84] image a k jejímu spuštění tak šlo použít GitHub service containers, díky kterým je možné v prostředí GitHub pipeline spustit docker kontejner a připojit se k němu (výpis kódu 7.1). End-to-end testy vyžadují ještě o něco komplexnější prostředí a jejich začlenění do pipeline se nestihlo.

```
services:
  sqlserver:
    image: mcr.microsoft.com/mssql/server:2022-latest
    env:
      SA_PASSWORD: '29Z#0*P63g!5'
      ACCEPT_EULA: 'Y'
    ports:
      - 1433:1433
```

Výpis kódu 7.1 Ukázka konfigurace SQL Server kontejneru v nastavení CI/CD pipeline

7.6 Uživatelské testování

Před odevzdáním práce byla testovací verze aplikace měsíc nasazena a mohli ji tak už vyzkoušet první uživatelé. Celkově jich aplikaci vyzkoušelo 7, z nichž 5 mělo předchozí zkušenosti s programováním. Každému byl odeslán odkaz na aplikaci a dostal za úkol zkusit vyřešit výzvu či několik výzev dle svého výběru.

Ohlasy byly veskrze pozitivní, ukázalo se však několik problémů s občasnou nejasností uživatelského rozhraní. Největší problémy dělalo odeslání řešení výzvy. Nebylo dostatečně jasné, že je nejdříve potřeba projekt s řešením vytvořit a až poté použít tlačítko pro jeho odeslání. Uživatelé očekávali, že po stisknutí tohoto tlačítka budou moci řešení teprve začít vytvářet. Po vysvětlení už to nedělalo žádný problém a tyto nedostatky tak vyřeší stručná uživatelská příručka.

Kapitola 8

Dokumentace

K aplikaci byla napsána také dokumentace. Jednak byly vytvořeny dokumenty z vyšší úrovně pohledu popisující aplikaci jako celek a také byla napsána dokumentace i k jednotlivým částem kódu, jako jsou třídy a metody. Dokumentace byla rozdělena podle toho, komu je určena, na uživatelskou a vývojářskou. Pro základní informace o aplikaci byly vytvořeny dva dokumenty:

Uživatelská příručka Tento dokument je určen uživatelům aplikace. Popisuje jak aplikaci používat a také je zde zdokumentována implementovaná varianta jazyka Karel. Dostupná je přímo v aplikaci. S ohledem na cílovou skupinu není psána striktně formálním jazykem

Vývojářská příručka V té je popsáno zejména jak projekt sestavit, spustit a začít vyvíjet. Dále je zde popsána základní struktura projektu a hlavní použité technologie. Je vytvořena ve formátu Markdown a umístěna v souboru `README.md` v kořeni repozitáře.

8.1 Třídy, metody a vlastnosti

U každé veřejné třídy, metody a vlastnosti klienta i serveru, kde to mělo alespoň trochu smysl, byly napsány dokumentační komentáře. Z nich je pak pomocí různých nástrojů pro konkrétní programovací jazyk generována HTML dokumentace zobrazitelná ve webovém prohlížeči.

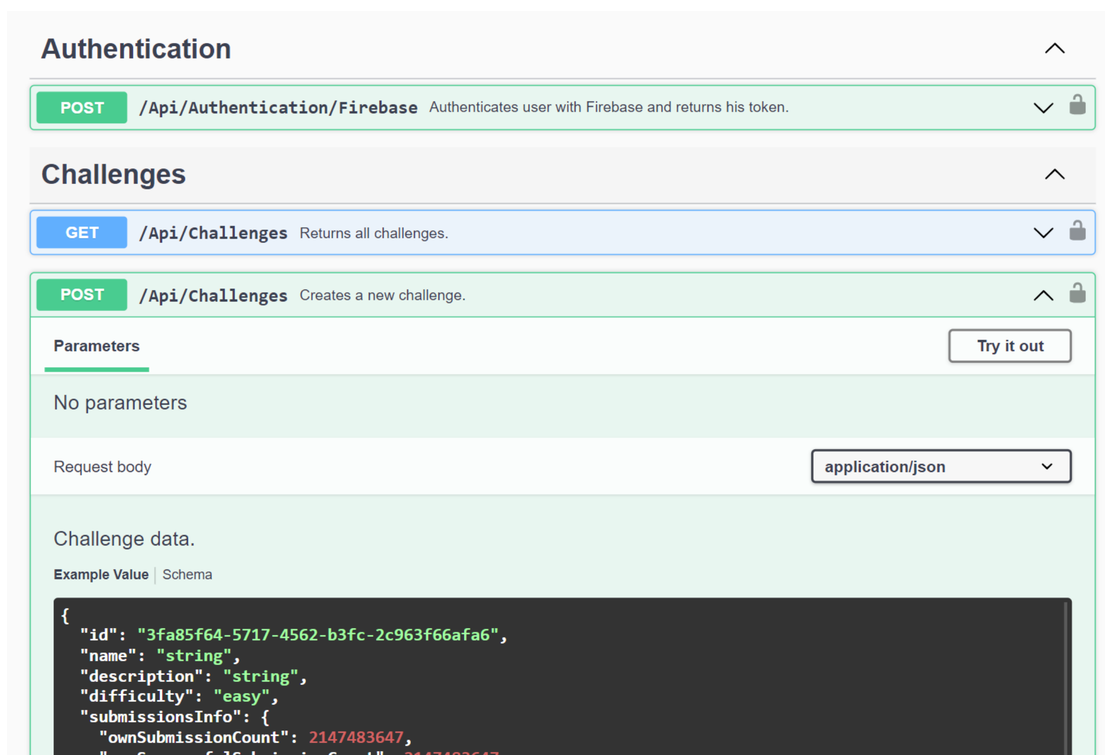
Serverová část Pro generování dokumentace serveru byl použit nástroj DocFX [85] od společnosti Microsoft. Tento nástroj do svého výstupu kromě generované dokumentace z kódu umožňuje přidat i vlastní textovou dokumentaci ve formátu Markdown. Tato možnost však zatím nebyla využita.

Klientská část Přesto, že by pro dokumentaci bylo možné použít nástroj pro generování dokumentace TypeScriptu, ve kterém je klientská část vytvořena, tak lepší volbou byl nástroj přímo specializovaný na framework Angular. Byl tak použit nástroj compodoc [86], který díky tomu, že rozumí Anguluru, umí odlišit i jednotlivé typy tříd, jako jsou komponenty, direktivy nebo služby. Zvládá také například vedle třídy komponenty zobrazit i její HTML šablonu. Nevýhodou však je, že v době psaní této práce ještě není plně kompatibilní s použitou verzí Angularu a některé věci tak nefungují úplně korektně.

Knihovna jazyka Karel Pro dokumentaci knihovny jazyka Karel byl použit nástroj pro generování TypeScriptové dokumentace TypeDoc [87].

8.2 REST API

K dokumentaci REST API byl využit nástroj Swagger UI [88]. Dokumentace je díky tomu interaktivní a lze z ní přímo zkoušet posílání požadavků na server. Je automaticky generována z kódu a dokumentačních komentářů kontrolerů pomocí knihovny Swashbuckle [89]. Ve vývojovém režimu je po spuštění serveru dostupná přímo ve webovém prohlížeči. Ukázka z této dokumentace je na obrázku 8.1.



■ **Obrázek 8.1** Ukázka části dokumentace REST API v nástroji Swagger UI

Kapitola 9
Nasazení

Aplikace byla v rámci bakalářské práce také nasazena a je tak už dostupná na internetu. Pro nasazení webové aplikace jsou zpravidla potřeba minimálně dvě věci. Webový server, kde bude serverová část aplikace hostována. A také doména, která bude na tento server nasměrována, a přes kterou budou moci uživatelé aplikaci nalézt.

9.1 Server

Aplikace je hostována u klasické webhostingové společnosti poskytující prostředí Windows serveru s IIS [90] webovým serverem a SQL Server databází. Nasazení aplikace bylo automatizováno pomocí CI/CD pipeline na platformě GitHub. Po každém merge do větve `master` je aplikace v případě, že úspěšně projdou její testy, sestavena a přes protokol FTPS odeslána na hosting.

9.2 Doména

Staré řešení bylo dostupné na doméně karlbot.cz. To má však pro aplikaci nevýhodu, že cz je česká národní doména. Přesto, že aplikace samotná je v anglickém jazyce, tak díky této doméně působí, že je určena pouze pro Českou republiku. Bylo tedy potřeba vybrat nějakou novou obecnější doménu, která by vystihovala účel aplikace.

Byla zakoupena doména karlbot.dev. Doména dev [91] je generická doména nejvyššího řádu spravovaná společností Google. Je určena například pro vývojové platformy, nástroje a programovací jazyky. Zajímavou vlastností této domény je, že je součástí „HSTS preload" seznamu [92] a prohlížeče tak neumožňují její použití bez šifrovaného protokolu HTTPS, což zvyšuje bezpečnost aplikací na této doméně.

Pro samotné zprovoznění bylo po zakoupení potřeba doménu přesměrovat na webový server hostingu, kde je aplikace umístěna. K tomu stačilo k doméně přidat DNS záznam typu A s IP adresou webového serveru uvedenou v administraci hostingu.

9.3 Certifikát

Jak už bylo zmíněno, tak k doméně dev je možné přistupovat pouze přes protokol HTTPS. K použití tohoto protokolu je nutné, aby stránka měla TLS certifikát. Ten byl získán u neziskové certifikační autority Let's Encrypt [93] pomocí nástroje v administraci hostingu.

Závěr

Vývojové prostředí pro jazyk Karel karlbot.cz umožňovalo ve webovém prohlížeči editaci kódu, úpravu města, ve kterém se robot Karel pohybuje, a spouštění vytvořených programů. Z technického i uživatelského hlediska bylo však nevyhovující. Hlavním cílem této práce bylo jeho kompletní přepracování a doplnění nových funkcionalit. S tím souvisel dílčí cíl analyzovat současný stav tohoto řešení a zároveň i ostatních aplikací podobného zaměření.

Začátek práce se věnoval úvodu do problematiky jazyka Karel. Hned poté bylo porovnáno 5 existujících vývojových prostředí tohoto jazyka a zmíněny jejich hlavní nedostatky. Speciálně pak byla rozebrána a zhodnocena implementace řešení karlbot.cz. Další část práce se už věnovala jeho úplnému přepracování, jinými slovy vytvoření nového řešení, které na něj navázalo. Na základě nedostatků existujících řešení a zadání práce byly definovány požadavky, které musí splňovat. Poté následoval jeho návrh a rešerše technologií pro realizaci. Té se věnovala další kapitola. Na konci práce byl popsán proces jeho testování, zdokumentování a nasazení.

Byly tak naplněny všechny body zadání a vzniklo samostatně použitelné webové vývojové prostředí pro jazyk Karel. Spolu s pokročilou editací města, kódu a jeho laděním nabízí i automaticky hodnocené výzvy nebo sdílení a ukládání projektů na serveru. Nad rámec zadání je aplikace také responzivní, použitelná i na mobilních zařízeních a nasazena do produkce. Na doméně karlbot.dev je dostupná komukoliv přímo ve webovém prohlížeči.

Pro implementaci byl použit framework Angular na klientské části a ASP.NET Core na serverové. Editor kódu využívá knihovnu CodeMirror a město je vykreslováno pomocí HTML elementu `canvas`. Jelikož obsažený kompilátor jazyka Karel je v jazyku TypeScript a server v jazyku C#, tak byla pro hodnocení výzev na serveru využita knihovna pro spouštění JavaScriptu ClearScript.

Možná budoucí rozšíření

V průběhu vývoje vzniklo několik nápadů, kam by se aplikace mohla dále ubírat. Shrnuty jsou v této kapitole.

Proměnné a matematické výrazy Stejné jako ostatní běžné varianty jazyka Karel ani tato neobsahuje proměnné a matematické výrazy. Problém pro začátečníky s programováním tak je, že mnoho typicky iterací řešených problému je nutné řešit složitěji s využitím rekurze. Logickým rozšířením by tak mohlo být jejich přidání. Na druhou stranu tato vlastnost dělá jazyk Karel unikátní a je otázkou, do jaké míry by se pak jednalo o vývojové prostředí pro jazyk Karel, a ne jen další aplikaci pro výuku programování, kterých jsou desítky.

Knihovny programů Některé programy například pro otočení vpravo nebo pohyb ke zdi je po čase otravné psát neustále znovu. Dala by se tak implementovat možnost referencovat z projektu některé jiné existující projekty a poté moci volat v nich definované programy. Oproti

proměnným podpora knihoven mnohem méně narušuje principy jazyka Karel, nepřidává mu nové možnosti, ale pouze zjednodušuje použití stávajících.

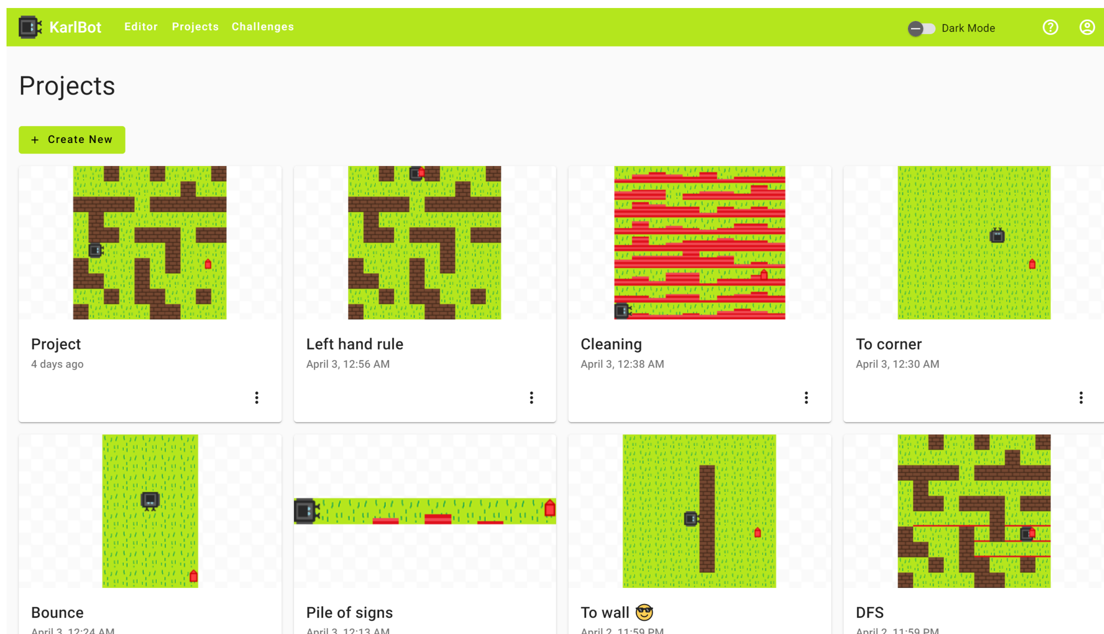
Dokumentační komentáře programů Další zajímavou vlastností by mohlo být zobrazování dokumentačních komentářů programů při napovídání v editoru.

Překlad do JavaScriptu nebo WebAssembly V tuto chvíli je pro všechny potřeby aplikace výkon interpretu naprosto dostačující, pokud by z nějakého důvodu někdy nebyl, bylo by možné sáhnout po kompilaci jazyka Karel do JavaScriptu nebo WebAssembly.

Docker Sestavení aplikace je kvůli jejím mnoha součástem a nutnosti stažení různých závislostí poměrně komplikované. Pomoci by mohl například nástroj jako je Docker. Jeho další výhodou by bylo zmenšení rozdílu mezi testovacím a produkčním prostředím.

Příloha A

Snímky obrazovek



Obrázek A.1 Obrazovka seznamu projektů

KarlBot Editor Projects Challenges Dark Mode

General maze

Hard

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Nullam justo enim, consectetuer nec, ullamcorper ac, vestibulum in, elit. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Integer lacinia. Mauris dolor felis, sagittis at, luctus sed, aliquam non, tellus. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Vestibulum fermentum tortor id mi. Praesent id justo in neque elementum ultrices. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

- Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Fusce wisi.
- Praesent in mauris eu tortor porttitor accumsan. In convallis.
- Nulla accumsan, elit sit amet varius semper, nulla mauris mollis quam, tempor suscipit diam nulla vel leo.

Examples

Submissions

+ Submit Project

Success
Just now

Failure (0 %)
Just now
Exception was thrown: Karel hit the wall.

Failure (75 %)
April 3, 2:33 AM

Failure (0 %)
April 2, 11:59 PM
Invalid output for some test cases.

■ **Obrázek A.2** Obrazovka výzvy

Test case 2 Public

Test case 3 Public

Check

Position of Karel Direction of Karel Sign counts

Width 12 Height 1 Width 12 Height 1

+ Add Test Case

Save Challenge

■ **Obrázek A.3** Obrazovka editoru výzvy (její část)

Bibliografie

1. PIECH, Chris; ROBERTS, Eric. *Chapter 1: Introducing karel the robot* [online]. 2019. [cit. 2023-04-05]. Dostupné z: `https://compedu.stanford.edu/karel-reader/docs/python/en/chapter1.html`.
2. PATTIS, Richard E. *Karel the Robot: A Gentle Introduction to the Art of Programming* [online]. 1. vyd. USA: John Wiley & Sons, Inc., 1981 [cit. 2023-04-18]. ISBN 0471089281. Dostupné z: `https://archive.org/details/karelrobotgentle00patt`.
3. IHPCI.ORG, Z.S. *Robot Karel* [online]. [cit. 2023-05-11]. Dostupné z: `https://karel1981.com/`.
4. CODEHS. *Intro to Programming with Karel the Dog (Ace)* [online]. [cit. 2023-04-16]. Dostupné z: `https://codehs.com/course/719/overview`.
5. NCLAB INC. *NCLab Karel HOC* [online]. 2021. [cit. 2023-04-16]. Dostupné z: `https://hoc.nclab.com/karel/`.
6. JEDLIČKA, Oldřich. *Robot Karel: vývojové prostředí* [online]. 2006. [cit. 2023-05-11]. Dostupné z: `http://karel.oldium.net/`.
7. STANFORD UNIVERSITY. *Karel IDE* [online]. [cit. 2023-05-11]. Dostupné z: `https://stanford.edu/~cpiech/karel/ide.html`.
8. ČOUPEK, Vojtěch. *Karel 3D* [online]. 2021. [cit. 2023-05-11]. Dostupné z: `http://karelrobot.cz/`.
9. ČOUPEK, Vojtěch. *Karel 3D – aplikace pro výuku programování* [online]. Brno, 2021 [cit. 2023-04-18]. Dostupné z: `https://theses.cz/id/uzu6i3/23969.pdf`. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií.
10. JÖRKA, Jan. *KarlBot* [online]. 2020. [cit. 2023-05-11]. Dostupné z: `http://karlbot.cz/`.
11. GOOGLE LLC. *Test použitelnosti v mobilech* [online]. [cit. 2023-04-16]. Dostupné z: `https://search.google.com/test/mobile-friendly`.
12. MICROSOFT CORPORATION. *TypeScript* [online]. 2012. [cit. 2023-04-18]. Dostupné z: `https://www.typescriptlang.org/`.
13. GOOGLE LLC. *Angular* [online]. 2010. [cit. 2023-04-16]. Dostupné z: `https://angular.io/`.
14. FONTICONS, INC. *Font Awesome* [online]. [cit. 2023-04-16]. Dostupné z: `https://fontawesome.com/`.
15. MICROSOFT CORPORATION. *Monaco Editor* [online]. 2023. [cit. 2023-04-16]. Dostupné z: `https://microsoft.github.io/monaco-editor/`.

16. SINNATHAMBY, Mark Vinod. *Stack Based vs Register Based Virtual Machine Architecture, and the Dalvik VM* [online]. 2012. [cit. 2023-04-18]. Dostupné z: https://www.codeproject.com/Articles/461052/Stack-Based-vs-Register-Based-Virtual-Machine-Arch.
17. STEC, Albert. *How Compilers Work* [online]. 2023. [cit. 2023-04-18]. Dostupné z: https://www.baeldung.com/cs/how-compilers-work.
18. MICROSOFT CORPORATION. *How are comments stored in the syntax tree (and how to use the Syntax Visualizer)?* [Online]. 2016. [cit. 2023-04-18]. Dostupné z: https://github.com/jasonmalinowski/roslyn-wiki/blob/master/FAQ.md#how-are-comments-stored-in-the-syntax-tree-and-how-to-use-the-syntax-visualizer.
19. MICROSOFT CORPORATION. *Roslyn* [online]. [cit. 2023-04-18]. Dostupné z: https://github.com/dotnet/roslyn.
20. MAKKAR, Jaglike. *Recursive Descent Parser* [online]. 2023. [cit. 2023-05-11]. Dostupné z: https://www.codingninjas.com/codestudio/library/recursive-descent-parser.
21. BRUNSFELD, Max. *Atom understands your code better than ever before* [online]. 2018. [cit. 2023-05-11]. Dostupné z: https://github.blog/2018-10-31-atoms-new-parsing-system/.
22. LIPPERT, Eric. *Persistence, façades and Roslyn's red-green trees* [online]. 2012. [cit. 2023-04-05]. Dostupné z: https://ericlippert.com/2012/06/08/red-green-trees/.
23. MICROSOFT CORPORATION. *Visual Studio Code* [online]. 2023. [cit. 2023-04-18]. Dostupné z: https://code.visualstudio.com/.
24. MICROSOFT CORPORATION. *TSConfig Reference – strict* [online]. 2012. [cit. 2023-05-11]. Dostupné z: https://www.typescriptlang.org/tsconfig#strict.
25. INTERNATIONAL BUSINESS MACHINES CORP. *What is three-tier architecture?* [Online]. [cit. 2023-04-16]. Dostupné z: https://www.ibm.com/topics/three-tier-architecture.
26. MICROSOFT CORPORATION. *Microservice architecture style* [online]. 2023. [cit. 2023-04-16]. Dostupné z: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices.
27. STACK EXCHANGE INC. *Stack Overflow Developer Survey 2022* [online]. 2022. [cit. 2023-05-03]. Dostupné z: https://survey.stackoverflow.co/2022/.
28. STACK EXCHANGE INC. *Tags – StackOverflow* [online]. 2023. [cit. 2023-05-03]. Dostupné z: https://stackoverflow.com/tags.
29. FACEBOOK, INC. *React* [online]. 2023. [cit. 2023-04-16]. Dostupné z: https://react.dev/.
30. META PLATFORMS, INC. *JSX* [online]. 2022. [cit. 2023-05-03]. Dostupné z: https://facebook.github.io/jsx/.
31. VERCEL, INC. *Next.js* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://nextjs.org/.
32. YOU, Evan. *Vue.js* [online]. 2014. [cit. 2023-04-16]. Dostupné z: https://vuejs.org/.
33. NUXT. *Nuxt* [online]. 2016. [cit. 2023-04-23]. Dostupné z: https://nuxt.com/.
34. MICROSOFT CORPORATION. *ASP.NET* [online]. 2023. [cit. 2023-04-16]. Dostupné z: https://dotnet.microsoft.com/en-us/apps/aspnet.
35. MICROSOFT CORPORATION. *.NET* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://dotnet.microsoft.com/.
36. MICROSOFT CORPORATION. *Razor syntax reference for ASP.NET Core* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/mvc/views/razor.

37. FIELDING, Roy Thomas. *Architectural styles and the design of network-based software architectures* [online]. Irvine, 2000 [cit. 2023-04-18]. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf. Disertační práce. University of California.
38. GOOGLE LLC. *gRPC* [online]. 2023. [cit. 2023-04-16]. Dostupné z: https://grpc.io/.
39. MICROSOFT CORPORATION. *C#* [online]. 2023. [cit. 2023-04-18]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/csharp/.
40. VMWARE, INC. *Spring* [online]. 2023. [cit. 2023-04-16]. Dostupné z: https://spring.io/.
41. JETBRAINS S.R.O. *Kotlin programming language* [online]. [cit. 2023-05-11]. Dostupné z: https://kotlinlang.org/.
42. ORACLE. *Java* [online]. 2023. [cit. 2023-05-11]. Dostupné z: https://www.java.com/.
43. JETBRAINS S.R.O. *Ktor* [online]. [cit. 2023-04-19]. Dostupné z: https://ktor.io/.
44. OPENJS FOUNDATION. *Express* [online]. 2017. [cit. 2023-04-16]. Dostupné z: https://expressjs.com/.
45. AMAZON.COM, INC. *What Is A RESTful API?* [Online]. 2023. [cit. 2023-04-16]. Dostupné z: https://aws.amazon.com/what-is/restful-api/.
46. GRAPHQL FOUNDATION. *GraphQL* [online]. 2023. [cit. 2023-04-16]. Dostupné z: https://graphql.org/.
47. GOOGLE LLC. *Protocol Buffers* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://protobuf.dev/.
48. BRANDHORST, Johan. *The state of gRPC in the browser* [online]. 2019. [cit. 2023-05-03]. Dostupné z: https://grpc.io/blog/state-of-grpc-web/.
49. GOOGLE LLC. *What is a relational database?* [Online]. [cit. 2023-04-16]. Dostupné z: https://cloud.google.com/learn/what-is-a-relational-database.
50. INTERNATIONAL BUSINESS MACHINES CORP. *ACID properties of transactions* [online]. 2023. [cit. 2023-04-16]. Dostupné z: https://www.ibm.com/docs/en/cics-ts/5.4?topic=processing-acid-properties-transactions.
51. MONGODB, INC. *What is NoSQL?* [Online]. 2023. [cit. 2023-04-16]. Dostupné z: https://www.mongodb.com/nosql-explained.
52. MONGODB, INC. *MongoDB* [online]. 2023. [cit. 2023-05-03]. Dostupné z: https://www.mongodb.com/.
53. MICROSOFT CORPORATION. *SQL Server* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://www.microsoft.com/en-us/sql-server.
54. STACK EXCHANGE INC. *Performance – Stack Exchange* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://stackexchange.com/performance.
55. GOOGLE LLC. *Firebase* [online]. [cit. 2023-04-16]. Dostupné z: https://firebase.google.com/.
56. GOOGLE LLC. *Verify ID Tokens* [online]. [cit. 2023-05-03]. Dostupné z: https://firebase.google.com/docs/auth/admin/verify-id-tokens.
57. MICROSOFT CORPORATION. *Index JSON data* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://learn.microsoft.com/en-us/sql/relational-databases/json/index-json-data?view=sql-server-ver16.
58. SOFTWARE FREEDOM CONSERVANCY. *Git* [online]. [cit. 2023-04-16]. Dostupné z: https://git-scm.com/.
59. GITHUB, INC. *GitHub* [online]. 2023. [cit. 2023-04-16]. Dostupné z: https://github.com/.

60. RUBIN, Dean. *The Three Layered Architecture* [online]. 2021. [cit. 2023-04-18]. Dostupné z: https://medium.com/@deanrubin/the-three-layered-architecture-fe30cb0e4a6.
61. PALERMO, Jeffrey. *The Onion Architecture : part 1* [online]. 2008. [cit. 2023-04-18]. Dostupné z: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/.
62. MARTIN, Robert C. *The Clean Architecture* [online]. 2012. [cit. 2023-04-18]. Dostupné z: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.
63. SMITH, Steve. *Architecting Modern Web Applications with ASP.NET Core and Microsoft Azure* [online]. 6. vyd. One Microsoft Way, Redmond, Washington 98052-6399: Microsoft Developer Division, .NET, a Visual Studio product teams, 2022 [cit. 2023-05-11]. Dostupné z: https://dotnet.microsoft.com/en-us/download/e-book/aspnet/pdf.
64. MICROSOFT CORPORATION. *Entity Framework Core* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://learn.microsoft.com/en-us/ef/core/.
65. GOOGLE LLC. *Firebase Admin .NET SDK* [online]. [cit. 2023-05-03]. Dostupné z: https://github.com/firebase/firebase-admin-dotnet.
66. MICROSOFT CORPORATION. *ClearScript* [online]. [cit. 2023-04-23]. Dostupné z: https://github.com/microsoft/ClearScript.
67. GOOGLE LLC. *V8 JavaScript engine* [online]. [cit. 2023-04-23]. Dostupné z: https://v8.dev/.
68. OPENJS FOUNDATION. *webpack* [online]. [cit. 2023-05-03]. Dostupné z: https://webpack.js.org/.
69. *RxJS* [online]. [cit. 2023-05-03]. Dostupné z: https://rxjs.dev/.
70. HAVERBEKE, Marijn. *CodeMirror* [online]. [cit. 2023-04-16]. Dostupné z: https://codemirror.net/.
71. HAVERBEKE, Marijn. *The Lezer Parser System* [online]. [cit. 2023-04-23]. Dostupné z: https://lezer.codemirror.net/.
72. KOCHARIN, Alex; PUZRIN, Vitaly. *markdown-it* [online]. [cit. 2023-04-23]. Dostupné z: https://github.com/markdown-it/markdown-it.
73. GOOGLE LLC. *AngularFire* [online]. [cit. 2023-04-23]. Dostupné z: https://github.com/angular/angularfire.
74. GOOGLE LLC. *Material Design* [online]. [cit. 2023-04-16]. Dostupné z: https://m2.material.io/.
75. GOOGLE LLC. *Angular Material UI component library* [online]. 2010. [cit. 2023-04-16]. Dostupné z: https://material.angular.io/.
76. GOOGLE LLC. *CDK | Angular Material* [online]. 2010. [cit. 2023-05-03]. Dostupné z: https://material.angular.io/cdk/categories.
77. CHEN, Raymond. *What are these spurious nop instructions doing in my C# code?* [Online]. 2007. [cit. 2023-05-11]. Dostupné z: https://devblogs.microsoft.com/oldnewthing/20070817-00/?p=25533.
78. WHATWG. *HTML standard – Event loops* [online]. 2023. [cit. 2023-04-23]. Dostupné z: https://html.spec.whatwg.org/multipage/webappapis.html#event-loops.
79. POOLE, Charlie; PROUSE, Rob. *NUnit* [online]. 2023. [cit. 2023-05-03]. Dostupné z: https://nunit.org/.
80. CLARIUS; MANAS; INSTEDD. *moq* [online]. [cit. 2023-05-03]. Dostupné z: https://github.com/moq/moq4.
81. CYPRESS. *Cypress* [online]. [cit. 2023-05-11]. Dostupné z: https://www.cypress.io/.

82. HOVE, Gwendolyn Van; GRAVROCK, Steve. *Jasmine* [online]. [cit. 2023-05-03]. Dostupné z: https://jasmine.github.io/.
83. OPENJS FOUNDATION. *ESLint* [online]. [cit. 2023-05-03]. Dostupné z: https://eslint.org/.
84. DOCKER INC. *Docker* [online]. 2023. [cit. 2023-05-03]. Dostupné z: https://www.docker.com/.
85. MICROSOFT CORPORATION. *docfx* [online]. [cit. 2023-04-18]. Dostupné z: https://dotnet.github.io/docfx/.
86. OGLOBLINSKY, Vincent. *compodoc* [online]. 2016. [cit. 2023-04-18]. Dostupné z: https://compodoc.app/.
87. TYPESTRONG. *TypeDoc* [online]. [cit. 2023-04-18]. Dostupné z: https://typedoc.org/.
88. SMARTBEAR SOFTWARE. *Swagger UI* [online]. 2023. [cit. 2023-04-18]. Dostupné z: https://swagger.io/tools/swagger-ui/.
89. DOMAINDRIVENDEV. *Swashbuckle.AspNetCore* [online]. [cit. 2023-04-18]. Dostupné z: https://github.com/domaindrivendev/Swashbuckle.AspNetCore.
90. MICROSOFT CORPORATION. *Internet Information Services* [online]. 2023. [cit. 2023-04-18]. Dostupné z: https://www.iis.net/.
91. GOOGLE LLC. *.dev* [online]. [cit. 2023-04-18]. Dostupné z: https://get.dev/.
92. GOOGLE LLC. *HSTS Preload List* [online]. [cit. 2023-04-18]. Dostupné z: https://hstspreload.org/.
93. INTERNET SECURITY RESEARCH GROUP. *Let's Encrypt* [online]. [cit. 2023-04-18]. Dostupné z: https://letsencrypt.org/.

Obsah přiloženého archivu

```
readme.txt ........................ stručný popis obsahu archivu
src
├── application ................... zdrojové kódy aplikace
└── thesis ........................ zdrojová forma práce ve formátu LaTeX
text .............................. text práce
└── thesis.pdf .................... text práce ve formátu PDF
```