VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ROZHRANÍ PRO AUTOMATIZOVANÉ TESTOVÁNÍ KNIHOVNY GEOVISTO

INTERFACE FOR AUTOMATIC TESTS OF GEOVISTO LIBRARY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. LUKÁŠ DOHNAL**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JIŘÍ HYNEK, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2021/2022

Zadání diplomové práce

Student:	**Dohnal Lukáš, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Informační systémy a databáze
Název:	**Rozhraní pro automatizované testování knihovny Geovisto** **Interface for Automatic Tests of Geovisto Library**
Kategorie:	Informační systémy

Zadání:

1. Prostudujte problematiku zpracování a vizualizace geografických dat na webu a prozkoumejte existující technologie určené pro tento účel (např. Leaflet, d3-geo, Geovisto, apod.).
2. Prostudujte problematiku testování a profilování projektů v jazyce JavaScript/TypeScript. Proveďte průzkum existujících nástrojů určených pro tento účel (např. Jest, Mocha, Cypress, Selenium, apod.).
3. Prozkoumejte existující informační systém pro správu dat a konfigurací mapových instancí vytvořených pomocí knihovny Geovisto. Analyzujte současné možnosti testování a profilování Geovisto. Proveďte průzkum dostupných otevřených datových sad vhodných pro testování geovizulačních nástrojů a vybraných tematických map.
4. Navrhněte rozšíření informačního systému z bodu 4 o rozhraní, které umožní provádět automatizované testy nad jednotlivými součástmi knihovny Geovisto.
5. Navržené rozšíření implementujte.
6. Pomocí implementovaného rozšíření proveďte testování knihovny Geovisto a navrhněte možná vylepšení kvality a výkonnosti knihovny.

Literatura:

- Hynek, J., Kachlík, J. a Rusňák, V.: Geovisto: *A Toolkit for Generic Geospatial Data Visualization*. In *VISIGRAPP (3: IVAPP)* (pp. 101-111).
- Grossmann, J.: *Informační systém pro správu vizualizací geografických dat*. Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií.
- Dent, B., D., a spol.: *Cartography: Thematic Map Design*. McGraw-Hill Higher Education 2009, ISBN 978-128-3388-023.
- Cypress: *Cypress Documentation* [online]. 2021 [cit. 2021-10-09]. Dostupné z: https://docs.cypress.io/
- Jest: *Getting Started* [online]. 2021 [cit. 2021-10-09]. Dostupné z: https://jestjs.io/docs/getting-started
- Leaflet: *Leaflet API reference* [online]. 2021 [cit. 2021-10-09]. Dostupné z: https://leafletjs.com/reference-1.7.1.html

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 4.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Hynek Jiří, Ing., Ph.D.**
Vedoucí ústavu:	Kolář Dušan, doc. Dr. Ing.
Datum zadání:	1. listopadu 2021
Datum odevzdání:	18. května 2022
Datum schválení:	11. října 2021

Abstrakt

Cílem této diplomové práce je navrhnout a implementovat rozhraní pro automatizované testování knihovny Geovisto. Rozhraní by mělo umožnit efektivní a uživatelsky přívětivé testování knihovny při jejím vývoji. V rámci práce bude proveden průzkum knihoven k testování projektů v jazycích JavaScript a TypeScript. Práce zahrnuje vývoj testů aktuálních modulů knihovny Geovisto a provedení profilování knihovny, pomocí kterého budou navrženy možné změny v implementaci ke zlepšení výkonnosti knihovny.

Abstract

The aim of this master thesis is to design and implement an interface for automated testing the Geovisto library. The interface should allow efficient and user-friendly testing the library during its development. The thesis includes a survey of libraries to test projects in programming languages JavaScript and TypeScript. The thesis includes the development of tests of current modules of the Geovisto library and the library profiling, which will be used to propose possible changes in the implementation to improve library performance.

Klíčová slova

automatizované testování, geovizualizace, geografická data, testovací framework, JavaScript, TypeScript, Geovisto

Keywords

automated testing, geovisualization, geographical data, testing framework, JavaScript, TypeScript, Geovisto

Citace

DOHNAL, Lukáš. *Rozhraní pro automatizované testování knihovny Geovisto.* Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jiří Hynek, Ph.D.

Rozhraní pro automatizované testování knihovny Geovisto

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jiřího Hynka, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Lukáš Dohnal
16. května 2022

Poděkování

Rád bych poděkoval panu Ing. Jiřímu Hynkovi, Ph.D za odborné vedení mé diplomové práce, ochotu, cenné rady, čas strávený konzultacemi a za poskytnutí zpětné vazby k práci.

Obsah

Kapitola 1

Úvod

Testování je nedílnou součástí vývoje software, které nám umožňuje snížit riziko neplánovaných výdajů. Testování má za cíl nejen nalézt chyby, ale zároveň i ověřit, že software se chová tak, jak je od něj požadováno. Existuje velké množství způsobů a technik, jak software otestovat. Výběr dané metody záleží především na typu aplikace, kterou chceme testovat. Tato práce se zaměřuje na automatizované testování geografické knihovny, především pak na správnou funkcionalitu jednotlivých nástrojů knihovny.

Cílem této práce je vytvořit rozhraní pro automatizované testování knihovny Geovisto. Tato knihovna slouží pro práci s geografickými daty. Knihovna se skládá z několika nástrojů, které umožňují vytvářet tematické mapy nad geografickými daty. V aktuální době ale neexistuje žádný způsob, jak tuto knihovnu efektivně testovat. Při úpravách knihovny a jejich nástrojů je tak nutné ji testovat ručně, což je způsob značně zdlouhavý a zároveň neefektivní. Práce se rovněž zabývá průzkumem možností, jakými lze provádět jednotkové a end-to-end testování ve webových aplikacích a jak je možné tyto aplikace profilovat za účelem optimalizace výkonnosti.

Kapitola 2 se zabývá definicí geografických datových sad a geografických objektů, ze kterých se mohou skládat či na ně odkazovat. Dále popisuje možnosti referencí na geografické objekty a různé druhy formátů, v jakých mohou být geografické objekty zpracovány. Kapitola 3 se zaměřuje na různé způsoby vizualizace geografických dat a na nástroje, které můžeme k tvorbě těchto vizualizací využít. Kapitola 4 se zaměřuje na definici testování a pohled na testování z hlediska životního cyklu vývoje software. Dále rozebírá jednotlivé druhy testů a nástroje, které můžeme využít pro testování projektů v jazyku JavaScript/TypeScript. Tyto nástroje jsou v této kapitole navzájem porovnány a následně se kapitola zaměřuje na možnosti profilování webových aplikací. Kapitola 5 definuje cílovou skupinu uživatelů a jejich potřeby. Dále je zde definován samotný problém a popis aktuálních možností pro řešení daného problému.

V kapitole 6 je popsána architektura implementace, která řeší definovaný problém. Dále je zde uveden výčet testů, které budou implementovány, s jakými datovými sadami a konfiguracemi budou provozovány a které nástroje knihovny nad nimi budou testovány. Tato kapitola taktéž obsahuje návrhy scénářů pro profilování knihovny, návrh uživatelského rozhraní a datového modelu. V kapitole 7 je popsána architektura informačního systému, do kterého bude rozhraní pro automatizované testování knihovny Geovisto implementováno. Dále je zde popsána práce s testy a jakým způsobem probíhal proces profilování. V kapitole 8 je provedeno testování funkcionality rozhraní pro automatizované testování. Dále jsou zde popsány výsledky profilování a provedené optimalizace plynoucí z těchto výsledků.

Kapitola 2

Geografická data a jejich zpracování

Tato kapitola se zaměřuje na definici geografických datových sad a geografických objektů, ze kterých se mohou skládat, či na ně odkazovat. Dále se zabývá možnostmi referencí na geografické objekty a popisuje možnosti, v jakých formátech mohou být zpracovány.

2.1 Geografická data

Dle [29] je geografická datová sada definována jako kolekce geoprostorových dat. Geografická data se od ostatních druhů dat liší v tom, že jedna z datových domén geografických dat reprezentuje geografickou polohu. Ta může být reprezentována buď přímo popisem geografické polohy (souřadnice zeměpisné délky a šířky) či referencí na nějaký geografický objekt. Díky této vlastnosti mohou být geografická data následně promítnuta přímo na mapu.

Jak již bylo výše zmíněno, geografická data se odkazují na geografické objekty. Reference probíhá pomocí identifikátorů, které obsahují jednotlivé geografické objekty. Tyto identifikátory mohou být standardizované, ale mohou být také námi zvolené a vytvořené. Geografické objekty následně mohou být reprezentovány jako body, čáry či polygony.

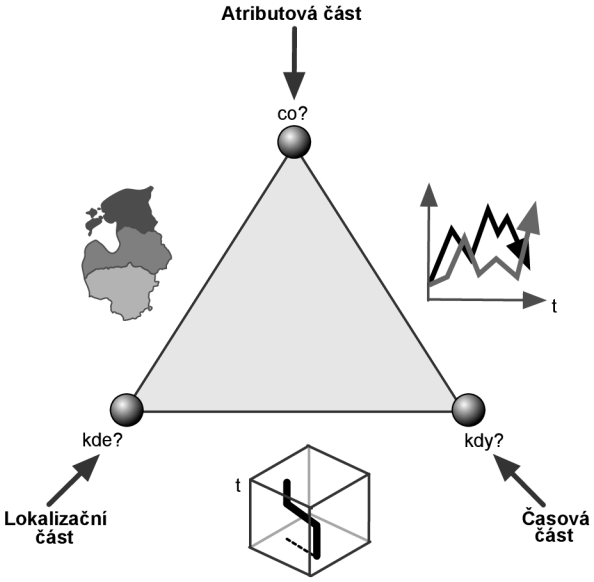
Menno-Jan Kraak a Ferjan Ormeling [29] rozdělují geoprostorová data do tří částí – lokalizační, atributová a časová. Lokalizační část odkazuje na geometrické aspekty – poloha a rozměry, zatímco atributová část odkazuje na jiné, negeometrické charakteristiky objektu. Časová složka odkazuje na okamžik v čase, pro který je lokalizační a atributová část platná. Pomocí těchto částí jsme následně schopni odpovědět na tři základní otázky ohledně geoprostorových dat, které stanovily – Co?, Kde? a Kdy?, viz. obrázek 2.1.

Kdykoliv se rozhodujeme, jak zobrazit model dat reálného světa pomocí geografického informačního systému, musíme rozhodnout, jakou charakteristiku mají geografické objekty, které se snažíme zobrazit. Existují dva typy těchto objektů.

Prvním typem jsou vektorová data. S pomocí těchto dat svět zobrazujeme jako kolekci objektů, které mají pevné umístění (bod) či pevný počáteční a koncový bod (úsečka) nebo nějaký druh pevné hranice (mnohoúhelníky). Druhým typem dat jsou rastrová data. Pomocí rastrových dat svět zobrazujeme jako mřížku obrazových dat [40].

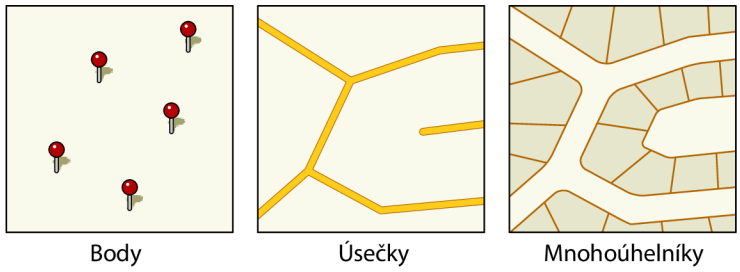
2.1.1 Zobrazení vektorových dat

Jak již bylo zmíněno výše, při zobrazení vektorových dat využíváme tři základní typy objektů: body, úsečky a mnohoúhelníky (obrázek 2.2).



Obrázek 2.1: Tři části geoprostorových dat [29]

- Body: Jsou to objekty vyjadřující polohu objektu. Pomocí bodů tak můžeme vyjádřit například polohu stromů či požárních hydrantů.
- Úsečky: Jsou to objekty, které mají počáteční a koncový bod. Pomocí úseček můžeme vyjádřit jednotlivé silnice či místa vedení elektrického vedení.
- Mnohoúhelníky: Jsou to objekty, které se skládají z kolekce úseček, jež určují jeho hranice. Pomocí mnohoúhelníků můžeme vyjádřit hranice pozemků či států.



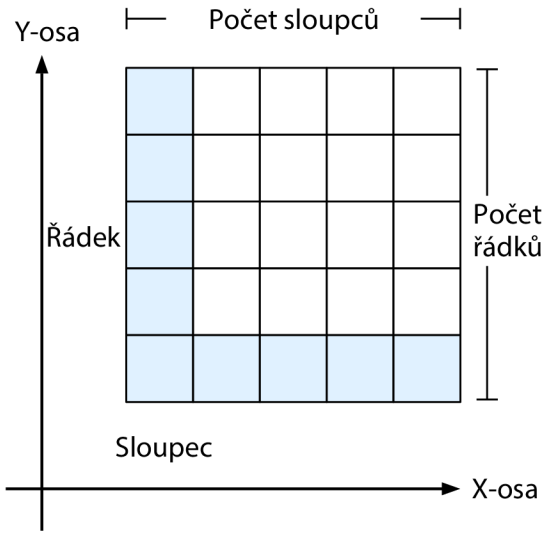
Obrázek 2.2: Základní objekty vektorového zobrazení dat [40]

2.1.2 Zobrazení rastrových dat

Pro zobrazení obrazových dat v geografických informačních systémech se používají rastrová data. Ty jsou reprezentována pomocí matice bodů (někdy taktéž nazývané jako buňky), které obsahují jednotlivé hodnoty (obrázek 2.3). Každý bod (buňka) představuje určitou geografickou oblast a hodnota v tomto bodu vyjadřuje určitou charakteristiku dané oblasti.

Například pokud bychom reprezentovali data obsahující teplotu vzduchu na daném území, tak každá buňka mřížky by mohla obsahovat barevnou intenzitu vyjadřující tep-

lotu vzduchu v dané oblasti na zemi. Pomocí rastrových dat se běžně zobrazují datové sady obsahující například teploty či půdu [40].



Obrázek 2.3: Mřížka buněk obsahující data rastrového modelu [40]

2.1.3 Reference na geografické objekty

Geografická data je potřeba pro jejich vizualizaci na mapě svázat s určitým místem na zemském povrchu. To můžeme udělat přímo pomocí definování zeměpisných souřadnic k daným datům nebo referencí na geografické objekty. Právě tyto reference nejčastěji používají jeden ze standardizovaných identifikátorů.

Open Location Code

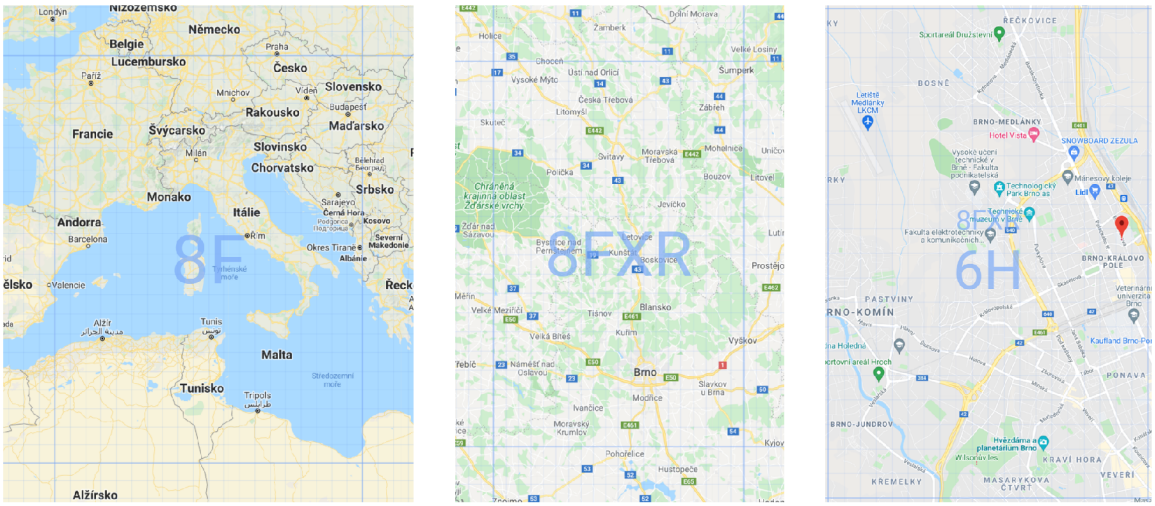
Open Location Code[1] je technologie vyvinutá firmou Google, která umožňuje snadnější vyjádření polohy pomocí jejího zakódování do formy, která se používá snadněji než zeměpisná šířka a délka. Tento algoritmus je veřejně dostupný a lze jej používat bez jakéhokoliv omezení. Kódy, vygenerované pomocí této technologie se nazývají „plus kódy“, protože je pro ně charakteristické, že obsahují znak +.

Tato technologie je navržena tak, aby vytvářela kódy, které můžeme použít jako náhrady za adresy ulic, a to zejména v místech, kde budovy nejsou číslovány, nebo ulice nejsou pojmenovány. Plus kódy představují oblasti, nikoliv body. Čím je plus kód delší, tak tím menší oblast vyjadřuje, tudíž dlouhé kódy jsou přesnější než krátké. Taktéž kódy, které jsou si více podobné, jsou umístěny blíže k sobě, než kódy, které jsou odlišné [15]. Ukázka využití Open Location Code je zobrazena na obrázku 2.4.

ISO 3166-1

ISO 3166 je mezinárodní norma pro kódy zemí a jejich podoblastí. Účelem normy ISO 3166 je definovat mezinárodně uznávané kódy písmen či číslic, které můžeme použít, když

[1] https://github.com/google/open-location-code



Obrázek 2.4: Ukázka využití Open Location Code pro lokalizaci vybrané oblasti[2]

odkazujeme na dané země a jejich podoblasti. Nedefinuje však názvy zemí – ty pocházejí ze zdrojů Organizace spojených národů. Používání kódů místo názvů zemí šetří čas a zabraňuje chybám, protože na rozdíl od názvů zemí, které se mění v závislosti na používaném jazyku, můžeme použít kombinaci písmen/číslic, které jsou srozumitelné po celém světě.

Pro příklad všechny národní poštovní organizace v celém světě si vyměňují mezinárodní poštu v kontejnerech, které jsou označeny příslušným kódem země. Systém internetových domén používá kódy k definování názvů domén nejvyšší úrovně a ve strojově čitelných pasech se kódy používají k určení národnosti uživatele.

Kódy zemí definuje standard ISO 3166-1, který je součástí standardu ISO 3166 a v tomto standardu mohou být reprezentovány buď jako dvoupísmenný kód (alpha-2), který se doporučuje jako univerzální kód či třípísmenný kód (alpha-3), který je bližší názvu země. Pokud se chceme vyhnout používání latinky, můžeme využít tří-číselný kód (numeric-3) [43].

2.2 Zpracování geografických dat

Geografická data mohou být zpracována v různých formátech. V této sekci se budeme zabývat nejčastějšími formáty pro práci s geografickými daty.

2.2.1 GeoJSON

GeoJSON je formát pro kódování geografických dat založený na formátu JSON. Definuje několik typů vlastních objektů. Dále definuje způsob, jakým jsou tyto vlastní objekty kombinovány pro reprezentaci geografických rysů, jejich vlastností a prostorové rozlohy. GeoJSON používá geografický souřadnicový referenční systém World Geodetic System 1984 a jednotku desetinných stupňů [5].

GeoJSON formát obsahuje následující typy geometrických objektů:

- Point – vyjádření bodu, obsahuje souřadnice umístění bodu.
- LineString – vyjádření úsečky, obsahuje souřadnice počátečního a koncového bodu.

[2]Zdroj: Plus Codes `https://plus.codes/map`

- Polygon – mnohoúhelník, obsahuje souřadnice jednotlivých vrcholů mnohoúhelníku.
- Multipoint – kolekce bodů, obsahuje pole souřadnic jednotlivých bodů.
- MultiLineString – kolekce úseček, obsahuje pole souřadnic počátečních a koncových bodů jednotlivých úseček.
- MultiPolygons – kolekce mnohoúhelníků, obsahuje pole souřadnic vrcholů jednotlivých mnohoúhelníků.
- GeometryCollections – kolekce geometrických objektů, obsahuje kolekci výše zmíněných typů geometrických objektů.

Na příkladu 2.1 je možné vidět jednoduchou ukázku objektu `GeometryCollections` obsahující dva geometrické objekty – bod a úsečku:

```
{
    "type": "GeometryCollection",
    "geometries": [{
        "type": "Point",
        "coordinates": [120.0, 25.0]
    }, {
        "type": "LineString",
        "coordinates": [
            [110.0, 20.0],
            [130.0, 30.0]
        ]
    }]
}
```

Výpis 2.1: Ukázka objektu GeometryCollections ve formátu GeoJSON

2.2.2 KML

KML (*Keyhole Markup Language*) je XML formát používaný k zobrazování geografických informací. Tak jako webové prohlížeče čtou a zobrazují HTML soubory, mapové prohlížeče jako Google Earth čtou a zobrazují KML soubory. Vzhledem k srozumitelnosti KML souborů je možno KML soubory lehce editovat a následně zobrazovat v mapových prohlížečích.

KML bylo vytvořeno v roce 2001 ve společnosti Keyhole jako datový formát pro jejich mapový prohlížeč zvaný *Earth Viewer*. Od té doby se KML vyvinulo a získalo status mezinárodního standardu pro vizualizaci geografických informací. Oficiální název KML je *OpenGIS KML 2.2 Encoding Standard* a je vyvíjen mezinárodní standardizační organizací *Open Geospatial Consortium*. V dnešní době je KML formát podporován velkým počtem aplikací jako například Google Earth, Google mapy, ArcGIS Explorer a mnoho dalších [45]. Na ukázce 2.2 můžeme vidět ukázku formátu KML zobrazující bod na mapě společně s popiskem, který následně může geografický informační systém u bodu zobrazit.

```
<?xml version='1.0' encoding='UTF-8'?>
<kml xmlns='http://www.opengis.net/kml/2.2'>
    <Placemark>
        <name>Karlův most</name>
        <description>
            Slavný most, který spojuje Staré město s Malou Stranou.
        </description>
        <Point>
            <coordinates>14.4115,50.0864,0</coordinates>
        </Point>
    </Placemark>
</kml>
```

Výpis 2.2: Ukázka formátu KML

2.2.3 Shapefile

Formát Shapefile slouží pro uložení vektorových prostorových dat pro geografické informační systémy. Obsahuje netopologickou geometrii a data atributů pro prostorové entity v jedné datové sadě. Shapefile byl vyvinut firmou Esri, která se zaměřuje na vývoj software určeného pro práci s geografickými informačními systémy.

Tento formát má výhodu oproti jiným datovým formátům v rychlejším vykreslování a možnosti editace. Je tomu tak díky tomu, že neobsahuje topologická data jako ostatní formáty. Shapefile podporuje 3 základní typy geometrických objektů – bod, úsečku a polygon [9].

2.2.4 GeoTIFF

GeoTIFF je rozšířením populárního formátu TIFF určeného pro ukládání rastrové grafiky. Tato datová struktura je navržena tak, aby splňovala funkci otevřeného, platformě nezávislého datového standardu pro přenášení a ukládání georeferenčních informací do TIFF souboru.

Cílem GeoTIFF je umožnit provázání rastrových obrazů se známými modely prostoru nebo projekcí map a pro popis těchto projekcí. Formát GeoTIF je plně kompatibilní s TIFF 6.0, takže i software, který není schopen přečíst a interpretovat specializovaná metadata je schopen tento soubor přečíst jako klasický TIFF [37].

Kapitola 3

Vizualizace geografických dat

Tato kapitola se zaměřuje na způsoby vizualizace geografických dat a na nástroje, které můžeme k vizualizaci využít. Kapitola nejprve definuje pojem tematických map a obsahuje popis nejčastějších typů tematických map. Následně jsou nástroje pro vizualizaci rozděleny do kategorií a k jednotlivým kategoriím uvedeny příklady těchto nástrojů.

3.1 Tematické mapy

Dle [7] se mapy klasifikují buď jako obecné neboli referenční mapy nebo tematické mapy. Obecné mapy běžně zobrazují objekty z geografického prostředí. Důraz je kladen na jejich umístění a účelem je ukázat různé rysy světa nebo jeho částí.

Tematické mapy, jak již napovídají z názvu, zobrazují data týkající se daného tématu. V oblasti geografických informačních systémů je to grafické zobrazení atributů jako je hustota osídlení, průměrný příjem obyvatelstva či denní změny teplot. Tyto atributy také nazýváme jako proměnné. Pro jednu tematickou mapu se vždy vybere pouze jedno téma, tímto se tematické mapy odlišují od referenčních map [38].

Tematické mapy vytváříme tak, že vezmeme geografické objekty, určíme, jakým způsobem budou tyto objekty vizualizovány a namapujeme na ně data. Vytvoříme tedy specifickou projekci dat nad danými geografickými objekty. Mohli bychom například použít geografické objekty obsahující polygony, ty zobrazíme na mapu a určíme, že tyto polygony budou nabývat daného barevného odstínu na základě dat, která tam následně dodáme.

Tematické mapy lze rozdělit na kvalitativní a kvantitativní. Účelem kvalitativních tematických map je znázornit prostorové rozložení či umístění jednoho tématu nominálních dat. Data kvalitativních tematických map nám značí pouze stav geografických objektů, ale nejsou zřejmé relace mezi těmito daty.

Kvantitativní tematické mapy zobrazují prostorové aspekty numerických dat. Ve většině případů se vybere jedna veličina, kterou chceme zobrazit (například populace či průměrný příjem obyvatelstva), která se následně zobrazuje na jednotlivých místech mapy. Kvantitativní tematické mapy zobrazují, jak moc či do jaké míry je něco přítomno v dané oblasti. Hlavní důvod existence kvantitativních map je transformace tabulkových dat do prostorového formátu mapy. Mapa se tak pro čtenáře stává přehlednější a srozumitelnější než tabulková forma dat [7]. Pokud bychom chtěli pomocí tematických map vyjádřit počasí, tak v případě kvalitativních tematických map bychom zobrazovali například oblačnost (jasno, polojasno, zataženo), zatímco s pomocí kvantitativních bychom zobrazovali číselné hodnoty teplot.

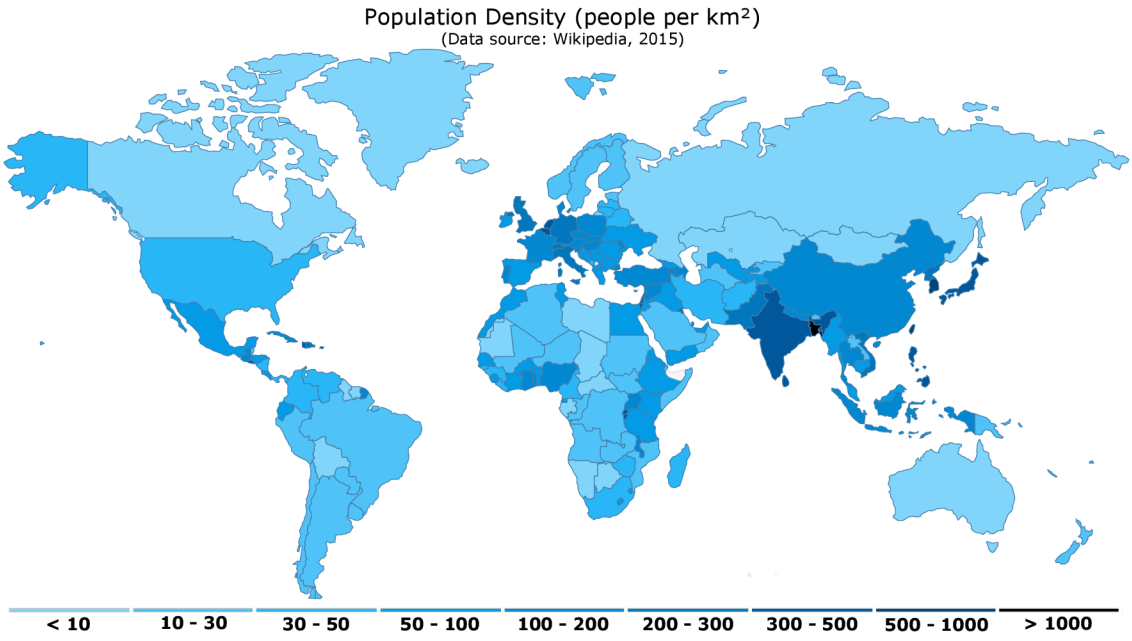
3.1.1 Kartogram

Dle [7] je kartogram neboli choropletová mapa (obrázek 3.1) jednou z nejčastěji používaných map v geoprostorových datech. Kartogramy se začaly používat na začátku devatenáctého století. Tento druh map se stal velmi oblíbeným díky snadnému pochopení a přehlednosti pro čtenáře.

Kartogramy využívají rozdílů barev, stínování či vzorů v jednotlivých oblastech mapy k reprezentaci hodnot, které nás zajímají. V dnešní době je nejpopulárnější vyjádření rozdílů hodnot s použitím barev, kdy můžeme měnit sytost barev v rámci jednoho barevného odstínu, či používat více odstínů barev. Ve většině případů platí, že tmavší/sytější barvy označují větší množství [7].

Kartogramy naopak nejsou vhodné pro zobrazení přesných hodnot v rámci jednotek. Proto by měli být pro tento typ tematických map použity normalizovaná data napříč geografickými oblastmi. To znamená, že pokud bychom chtěli vyjádřit formou kartogramu například populaci v jednotlivých zemích, tak místo počtu populace použijeme spíše četnost populace na čtvereční kilometr [36].

Při testování kartogramů se můžeme zaměřit na správné zobrazení barevných odstínů v jednotlivých oblastech.



Obrázek 3.1: Příklad kartogramu zobrazující hustotu populace v jednotlivých zemích v roce 2015[1]

3.1.2 Metoda teček

Mapování geografických jevů můžeme také zobrazit pomocí hustoty bodů, neboli metodou teček. V tomto druhu tematické mapy se zpravidla nemění tvar, velikost ani barva symbolu, pouze se používá frekvence teček k vyjádření počtu objektů zastoupených v dané oblasti.

[1] Zdroj: Anychart https://www.anychart.com/products/anymap/gallery/Maps_General_Features/World_Choropleth_Map.php

Obvykle jedna tečka vyjadřuje větší množství výskytů daného objektu [7]. Například na obrázku 3.2 můžeme vidět, že jedna tečka vyjadřuje zastoupení 100 000 obyvatel.

Dle [47] se tečkové mapy používají především k vyjádření distribuce diskrétních kvantitativních charakteristik především bodových jevů. Můžeme pomocí nich efektivně vyjádřit například množství hospodářských zvířat, obdělávané půdy či hustotu obyvatel. Nejsložitější činností při tvorbě tečkových map je správné zvolení rozmístění teček.

Testování metody teček můžeme provést třeba kontrolou hustoty teček v dané oblasti.

Obrázek 3.2: Příklad mapy populace USA využívající metodu teček[2]

3.1.3 Metoda proporcionálních bodových znaků

Metoda proporcionálních bodových znaků (obrázek 3.3) využívá bodové symboly různých velikostí k reprezentaci kvantitativních hodnot spojených s různými oblastmi či umístěními na mapě [32]. Například můžeme vyjádřit počet obyvatel města tak, že kruh umístíme na místo, kde se nachází dané město a plocha (velikost) kruhové výseče je úměrná počtu obyvatel daného města.

Kartograf[3] vybere formu symbolu, který bude používat (kruh, čtverec, trojúhelník, apod.) a mění jeho velikost v poměru k množství, které představuje. Zvolit vhodný tvar a velikost symbolů je proto velmi důležité pro tento typ tematické mapy [7].

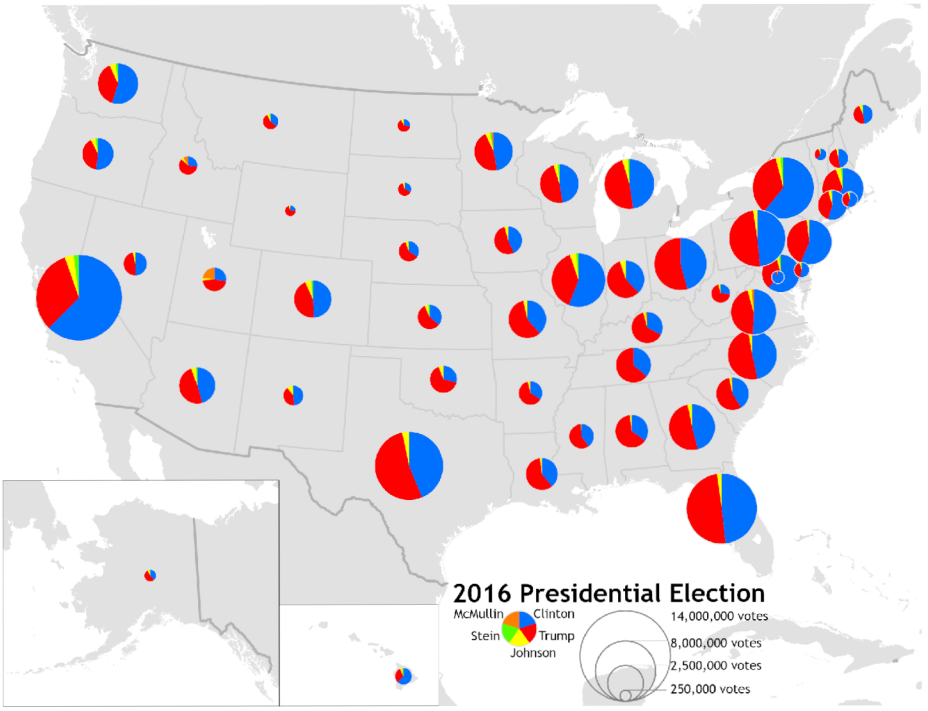
3.1.4 Mapa toků

Dle [42] mapy toků (obrázek 3.4) znázorňují lineární pohyb mezi místy. Znázornění pomocí toků kartografové používají, když chtějí ukázat jaký druh (v případě kvalitativního) či kolik pohybů (kvantitativní) existuje mezi dvěma či více místy.

V případě kvantitativního zobrazení šířka jednotlivých čar (toků) spojující jednotlivá místa znázorňuje poměr k celkovému množství zastoupeného pohybu. Změna velikosti šířky toků je nejběžnější používanou metodou pro tento typ mapy. V některých zobrazeních se ovšem můžeme setkat i se změnou odstínů či saturace toků [7].

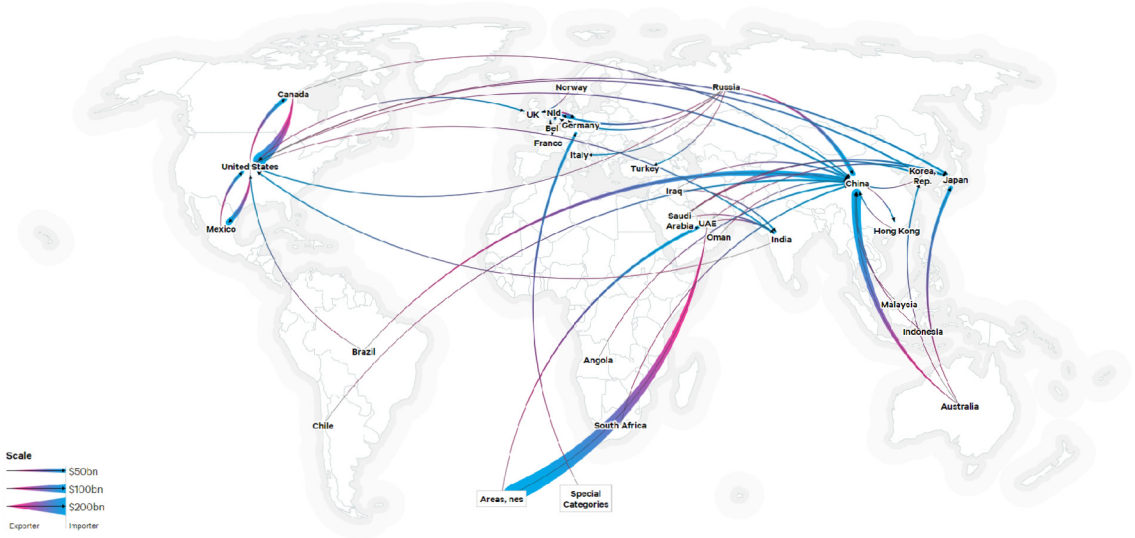
[2] Zdroj: The University of Arizona http://www.u.arizona.edu/~kbailey/GEOG416A/module8.htm

[3] Osoba, která se zabývá vytvářením a zpracováváním map



Obrázek 3.3: Příklad metody proporcionálních bodových znaků na mapě zobrazující výsledky voleb USA v roce 2016[4]

Při testování map toků můžeme kontrolovat například šířky jednotlivých čar v porovnání se vstupními daty.



Obrázek 3.4: Příklad mapy toků objemu obchodů[5]

[4]Zdroj: Wikimedia Commons https://commons.wikimedia.org/wiki/File:2016_US_Presidential_Election_Pie_Charts.png

[5]Zdroj: Chatham House https://resourcetrade.earth/

3.2 Nástroje pro vizualizaci

Stejně jako je trend v ostatních oblastech, tak i v oblasti geografických informačních systémů (dále jen GIS) se snažíme přenést nástroje pro práci s geografickými daty na web. Tomu přispívá rychlý vývoj technologií, který umožňuje provádět některé náročnější operace související s GIS na webu. S nynějšími možnostmi prohlížečů a výpočetním výkonem na straně klienta se dokonce ve webovém světě GIS objevil nový trend: WebGIS.

Tento nový trend zkoumá možnosti nasazení výkonných geografických informačních systémů na webu, umožnění obecných pracovních postupů prostorového analytika v prohlížeči a to platformově nezávislým způsobem. Díky OSGeo (*Open Source Geospatial Foundation* – nadace podporující vývoj otevřených geoinformačních technologií a dat), OGC (*Open Geospatial Consortium* – mezinárodní standardizační organizace), dalším společnostem a jednotlivcům prosazující open source filozofii měla rychlý a velký dopad na tuto oblast. V důsledku toho již existuje široká paleta open source aplikací a knihoven, které můžeme používat a budovat na nich [11].

Práci s geografickými daty a vytváření interaktivních vizualizací můžeme rozdělit do tří přístupů – ruční vytváření, geografické knihovny či autorských systémů.

3.2.1 Ruční vytváření interaktivních map

Základní způsob pro vytváření interaktivních map je využití HTML elementů pro zobrazení grafických prvků. Mezi tyto elementy se řadí především *canvas* (plátno) a SVG. Při využití plátna používáme jazyk JavaScript pro volání funkcí, které následně vykreslují základní geometrické objekty či text [25]. Druhou variantou je použití SVG (*Scalable Vector Graphics*) formátu. SVG je jazyk pro popis 2D grafiky v XML. Každý prvek v SVG je reprezentován jako samostatný prvek, což nám umožňuje na jednotlivé prvky připojit události vytvořené v JavaScriptu.

To je také jeden z největších rozdílů mezi plátnem a SVG – díky tomu, že v SVG je každý nakreslený tvar reprezentován jako samostatný objekt, při změně atributů daného objektu může prohlížeč tvar znovu automaticky vykreslit. U plátna vykreslení probíhá pixel po pixelu a jakmile je tvar nakreslen, prohlížeč jej zapomene. Pokud je potřeba následně změnit například polohu již vykresleného objektu, je potřeba překreslit celou scénu se všemi objekty [17].

Pro efektivnější práci lze využít některou z knihoven, které poskytují API pro vykreslování SVG elementů či kreslení na plátno. Jednou z nejpopulárnějších knihoven je knihovna D3.js[6]. Jedná se o JavaScriptovou knihovnu poskytující funkce pro práci s DOM[7] (*Data Object Model*) a transformace dat. Data jsou navázána na DOM a následně mohou být vizualizována pomocí SVG elementů.

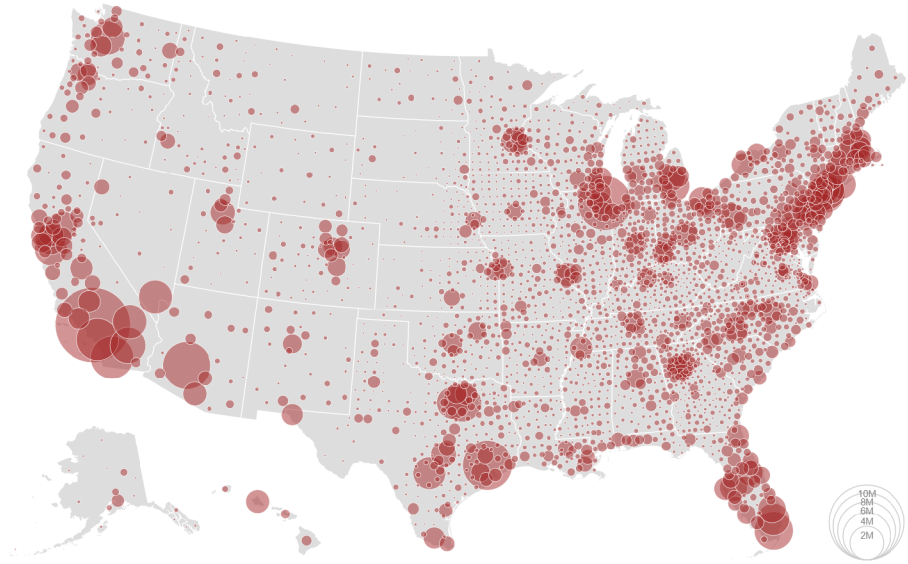
Samotná knihovna D3.js ovšem přímo neposkytuje žádné funkce pro vytváření tematických map. Tematické mapy jsou vytvářeny pomocí generování SVG elementů pro všechny prvky tematické mapy. Velkým problémem při zobrazování map je nutnost aplikovat mapové projekce, které zajišťují převod trojrozměrného zakřiveného povrchu koule do roviny [14]. Vytváření těchto projekcí je komplexní problém, jehož implementace není jednoduchá [2].

Pro řešení tohoto problému můžeme využít knihovnu d3-geo[8], která je jedním z modulů knihovny D3.js. Ta obsahuje sadu funkcí pro práci s geoprostorovými daty a vytvá-

[6] https://d3js.org/
[7] Objektová reprezentace HTML dokumentu
[8] https://github.com/d3/d3-geo

ření různých typů mapových projekcí. Pro příklad funkce `geoPath` nám při poskytnutí GeoJSONu vygeneruje datový řetězec cesty – atribut `d` u SVG elementu *path* určující hraniční body polygonu. Ukázku vytvořené tematické mapy pomocí knihovny D3.js můžeme vidět na obrázku 3.5.

Tento přístup vytváření interaktivních map nám zajišťuje největší volnost při jejich tvorbě, ale zároveň ze zmíněných přístupů vyžaduje nejvíce programovacích dovedností a matematických znalostí. V případě zájmu do mapy implementovat například funkci zoomu, je nutné vytvořit v JavaScriptu událost detekující pohyb kolečka myši, a ta by následně provedla přepočet atributů jednotlivých SVG elementů mapy tak, aby odpovídaly nové hodnotě přiblížení [19].



Obrázek 3.5: Mapa populace v USA v roce 2016 vytvořena pomocí D3.js (s rozšířením d3-geo)[9]

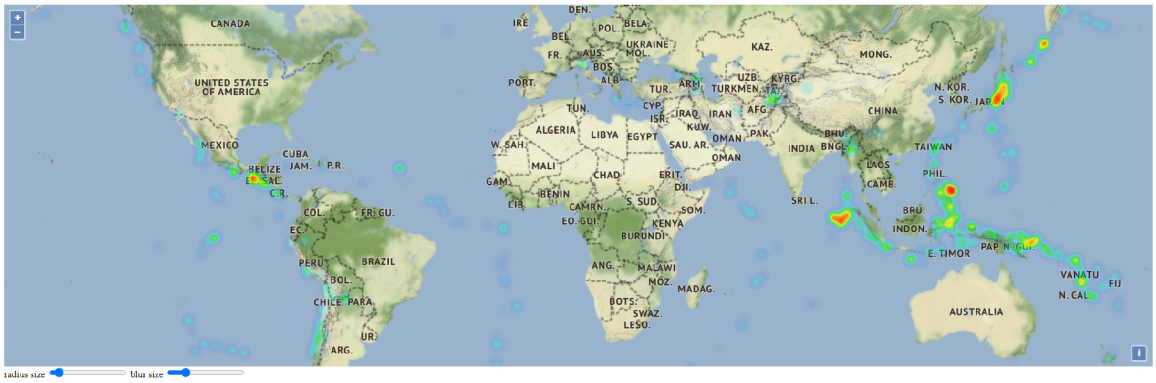
3.2.2 Geografické knihovny

Geografické knihovny poskytují rozhraní, které odstíní uživatele od nutnosti práce s projekcemi a s jednotlivými grafickými objekty (SVG). U těchto knihoven již uživatel není nucen aplikovat mapové projekce a kreslit objekty přímo na plátno či vytvářet pro ně SVG elementy. Místo toho zde může přímo definovat polohu objektů pomocí standardního souřadnicového systému a knihovna již vytvoření mapové projekce a vykreslení prvků zajistí za něj. Tyto knihovny obvykle zajišťují také základní funkce pro práci s vizualizacemi jako je přiblížení či posun mapy. Mezi nejznámější knihovny můžeme zařadit Leaflet[10], OpenLayers[11], Mapbox GL[12] či Google Maps Platform[13].

Při vytváření tematických map v geografických knihovnách vytváříme novou vrstvu s tematickou mapou (např. kartogramem) a tu následně vizualizujeme nad základní mapou.

[9] Zdroj: Observable https://observablehq.com/@d3/bubble-map
[10] https://leafletjs.com/
[11] https://openlayers.org/
[12] https://docs.mapbox.com/mapbox-gl-js/guides/
[13] https://mapsplatform.google.com/

Jako vrstvu základní mapy (*tile layer*) můžeme použít některou ze svobodných mapových podkladů, jako jsou třeba OpenStreetMap[14], ale existují také proprietální mapové vrstvy, které využívají například Google mapy či Mapbox[15]. Příklad mapy využívající mapový podklad OpenStreetMap, která je zobrazena pomocí knihovny OpenLayers, můžeme vidět na obrázku 3.6.



Obrázek 3.6: Mapa výskytu zemětřesení zobrazena pomocí knihovny OpenLayers[16]

3.2.3 Autorské systémy

Tato kategorie se vyznačuje odstíněním uživatele od nutnosti implementovat postup vykreslení mapy. Zde pouze popisujeme, jak má výsledná vizualizace vypadat, ale již nikoliv, jak se má vytvořit. U těchto aplikací obvykle načteme data, která chceme vizualizovat a následně deklarativně provádíme provázání dat s jednotlivými grafickými prvky. V praxi to vypadá tak, že pokud bychom například chtěli vytvořit tematickou mapu typu kartogram, tak zvolíme v poskytnutém rozhraní, jakou datovou doménu chceme namapovat na dimenzi dané tematické mapy, což může být například stát a z které datové domény chceme zobrazovat hodnoty. Do kategorie autorských systémů můžeme zařadit knihovnu Geovisto.

3.2.4 Geovisto

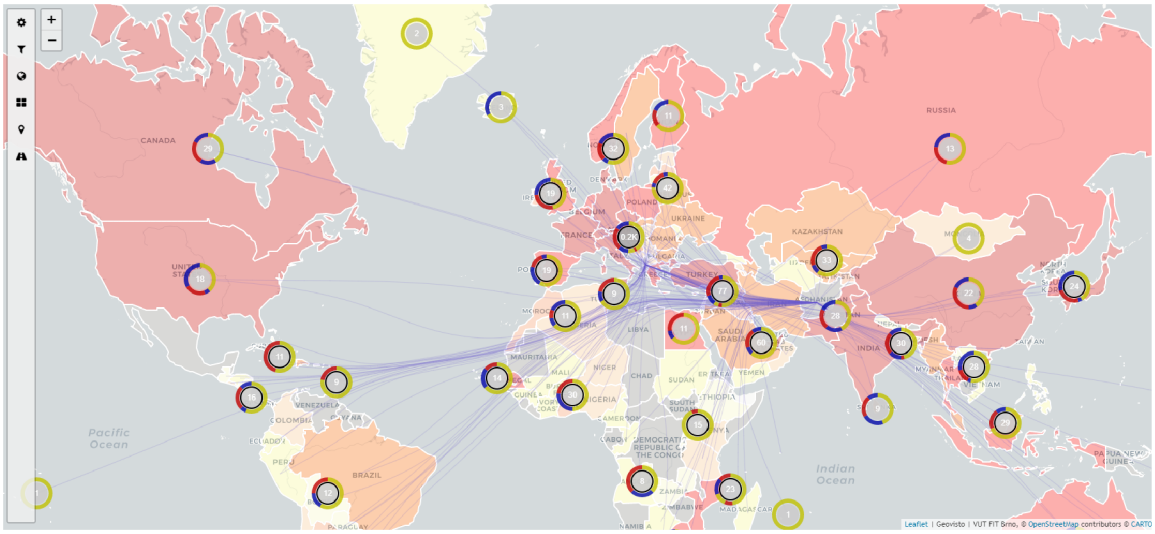
Knihovna Geovisto je programová knihovna, implementovaná pomocí knihoven Leaflet a D3.js. Tato knihovna se snaží abstrahovat tvorbu tematických map tím, že poskytuje sadu předpřipravených vrstev reprezentujících šablony pro jednotlivé tematické mapy. Je možné deklarativně definovat, jak tyto vrstvy mají vypadat a s jakými daty mají pracovat. Mimo to knihovna poskytuje rozšíření, která tato nastavení umožňují provádět prostřednictvím uživatelského rozhraní. Tato vlastnost knihovny patří mezi jednu z největších výhod, protože umožňuje uživatelům vytvářet geoprostorové vizualizace bez nutnosti znalosti programování.

Architekturu knihovny můžeme rozdělit na dvě části – jádro (tzv. *Geovisto core*) a nástroje (tzv. *Geovisto tools*). Jádro zajišťuje manipulaci s mapou (s použitím Leaflet API), zajišťuje inicializaci a správu jednotlivých Geovisto nástrojů a zpracovává datové sady s geografickými objekty a konfigurace. Nástroje pak přidávají další funkcionality, jako jsou ovládací prvky, mapové vrstvy či možnosti pro další zpracování dat (filtry). Jednotlivé nástroje

[14] https://www.openstreetmap.org/
[15] https://github.com/leaflet-extras/leaflet-providers
[16] Zdroj: OpenLayers https://openlayers.org/en/latest/examples/heatmap-earthquakes.html

mohou být dle potřeby zapnuty či vypnuty. Díky využití modulů v architektuře knihovny je umožněno jednoduché přidávání dalších funkcionalit v budoucnu.

Knihovna umožňuje exportovat stav mapy a jednotlivých nástrojů – konfigurace, které následně můžeme exportovat (JSON formát) a používat je na různých datových sadách. Tyto konfigurace mohou obsahovat konfigurace jednotlivých nástrojů či konfiguraci samotné mapy, jako je nastavení středu mapy či přiblížení. Na obrázku 3.7 můžeme vidět tematickou mapu využívající vrstvu kartogramu, vrstvu se značkami a spojovací vrstvu, znázorňující obchodování se surovinami mezi jednotlivými zeměmi vizualizovanou pomocí knihovny Geovisto. Níže jsou popsané nástroje, které lze využít [19].



Obrázek 3.7: Zobrazení obchodování surovin mezi jednotlivými zeměmi pomocí knihovny Geovisto

Sidebar tool

Tento nástroj poskytuje ovládací prvek v podobě postranního panelu. Tento panel lze skrýt a poskytuje API pro přidání nových položek do panelu.

Filter tool

Nástroj filtru umožňuje filtrování záznamů vizualizovaných dat. Nástroj taktéž poskytuje API rozhraní k definici vlastních pokročilých filtrů. Uživatelé specifikují filtrační pravidla jako podmíněné výrazy pro zvolené datové domény. K dispozici je filtr rovnosti, nerovnosti či možnost zadání regulárního výrazu.

Selection tool

Tento nástroj pro výběr poskytuje mechanismus pro propojení mapových vrstev s vybranými geografickými objekty mapové vrstvy. Komunikace mezi vrstvami je implementována pomocí návrhového vzoru *observer*. Každá událost je zaslána vrstvě obsahující informace o elementu, který uživatel zvolil. Ten se skládá z identifikátoru geografického prvku a dané vrstvy. Identifikátory geografických prvků lze využít ve více vrstvách. Filtrování je založeno na vyhledávání těchto identifikátorů v elementech mapové vrstvy.

Například při aktivní spojovací vrstvě, kartogramu a vrstvy se značkami se při výběru země vyvolá událost, která je předána dalším vrstvám mapy. Spojovací vrstva zpracuje událost, najde a zvýrazní všechna spojení, které spojují danou zemi s ostatními zeměmi. Toto vyvolá další událost, která obsahuje všechny země spojené se zvolenou zemí. To ovlivní vrstvu kartogramu a vrstvu se značkami, kde se zvýrazní příslušné země.

Themes tool

Nástroj motivů poskytuje sadu předdefinovaných stylů, které jsou dodány dalším nástrojům prostřednictvím událostí a obsahuje API, které umožňuje definovat vlastní motivy. Motivy ovlivňují nejen mapovou vrstvu, ale také nástroje jako je postranní panel.

Tile layer tool

Vrstva dlaždic představuje základní mapovou vrstvu, která využívá *Leaflet Tile layer API* pro zobrazení podkladových map. To může být vyžadováno, když je třeba propojit údaje se skutečnými zeměpisnými místy.

Choropleth layer tool

Nástroj vrstvy kartogramu umožňuje použít specifikace polygonů z GeoJSONu reprezentující geografické oblasti a propojit je s daty. Primárně se pracuje s oblastmi světových zemí. Lze však použít vlastní GeoJSON soubory, například pro rozdělení krajů, okresů či vlastních oblastí. Nastavují se zde datové dimenze země, hodnoty a druh agregace (počet nebo suma).

Marker layer tool

Nástroj vrstvy značek pracuje se specifikací bodů GeoJSON a pomocí značek (*marker*) vizualizuje data pro přesné geografické polohy. Podobně jako u kartogramů, kde se používají polygony místo bodů, má každá značka jedinečný identifikátor a zeměpisnou pozici (například střed země). Vzhledem k tomu, že vizualizace značek může být problematická, pokud je jich příliš mnoho u sebe, knihovna pro řešení tohoto problému využívá seskupení blízkých značek do skupin a agregaci jejich hodnot. Pro vrstvu se nastavují datové dimenze země, hodnoty, kategorie a druh agregace (počet nebo suma).

Connection layer tool

Nástroj spojovací vrstvy vizualizuje vztahy mezi geografickými místy formou hran. Vrstva vyžaduje nastavení datových dimenzí – od a do, které představují uzly vykreslených hran. Uživatel si může také aktivovat zobrazení animace směru hrany.

Kapitola 4

Testování

V této kapitole se zaměříme na definici testování a na pohled na testování z hlediska životního cyklu vývoje software. Dále jsou zde vysvětleny druhy testování. Následně se zaměříme na nástroje, které slouží pro testování a profilování projektů v jazyce JavaScript/TypeScript. Poslední sekce kapitoly se zabývá možnostmi profilování webových aplikací.

4.1 Definice

Dle [41] je testování software proces či série procesů navržených tak, abychom se ujistili, že implementovaný zdrojový kód dělá přesně to, k čemu byl navržen a nedělá nic, co nebylo zamýšleno. Software by měl být předvídatelný a konzistentní.

Hlavní motivací pro testování software je snížit riziko neplánovaných výdajů na vývoj či v horším případě riziko neúspěchu projektu [10]. Dle [27] bychom měli mít na paměti důležité zásady testování:

- Testování je proces spouštění programu se záměrem najít v tomto programu chyby.
- Testování bývá obvykle úspěšnější, pokud jej neprovádí přímo vývojář.
- Kvalitní testovací případ je takový, který má vysokou pravděpodobnost na neobjevenou chybu.
- Úspěšný testovací případ je takový, který odhalí dosud neobjevenou chybu.
- Úspěšné testování zahrnuje pečlivé definování očekávaného výstupu.
- Úspěšné testování zahrnuje pečlivé prostudování výsledků testů.

Testování můžeme dělit na testování černé (*black-box testing*) a bílé skříňky (*white-box testing*). Při testování *černé skříňky* (někdy také pod pojmy testování řízené daty či testování řízené vstupy/výstupy) vnímáme program jako černou skříňku se skrytým chováním a strukturou programu. Soustředíme se zde na nalezení okolností, za kterých se program nechová dle jeho specifikace. V tomto případě odvozujeme testovací data pouze ze specifikací programu. Pokud bychom chtěli využít tento přístup k nalezení všech možných chyb, které se v programu nacházejí, museli bychom vyzkoušet všechny platné vstupy. Bohužel, vyzkoušet všechny možné platné vstupy programu není obvykle možné, nemůžeme tak tímto druhem testování zaručit, že je program zcela bez chyb.

Testování *bílé skříňky* (neboli testování řízené logikou) využívá možnosti prozkoumat vnitřní strukturu programu. Tato testovací strategie vytváří testovací data ze zkoumání vnitřní logiky programu. Ovšem tento způsob testování nemusí být taktéž nijak jednoduchý. Je to například proto, že počet logických cest, kterými se může program při různých vstupech vydat je astronomicky velký. U této metody se vstupní data vytváří převážně na základě průzkumu vnitřní struktury programu. Kvůli tomu není zajištěno vyzkoušení vstupních dat, které nejsou pokryty v logice programu, a tak pro ně mohou vzniknout neočekávané výstupy neodpovídající specifikaci programu. Často se proto testování pomocí bílé a černé skříňky kombinuje a tuto kombinaci nazýváme testování šedé skříňky (*grey-box testing*) [27].

Z jiného hlediska můžeme testování rozdělit na statické a dynamické. Statické testování umožňuje testování software bez nutnosti jeho běhu. Díky tomu je možno tento typ testování použít již před vytvořením prvního prototypu. Výsledkem tohoto typu testování je vytvoření pohledu „shoda s požadavky“, který lze následně použít například pro zpřesnění odhadu náročnosti na čas vývoje projektu. Dynamické testování naopak vyžaduje spustitelnou verzi software. Toto testování využívá metodologie testování černé skříňky, jako jsou systémové testy a jednotkové (*unit*) testy. Testovací metody následně vyhodnocují produkt na základě vytvořených testů a označují je jako „vyhovující“ či „nevyhovující“. Tato technika testování vytváří pohled „vhodnosti pro použití“ produktu [23].

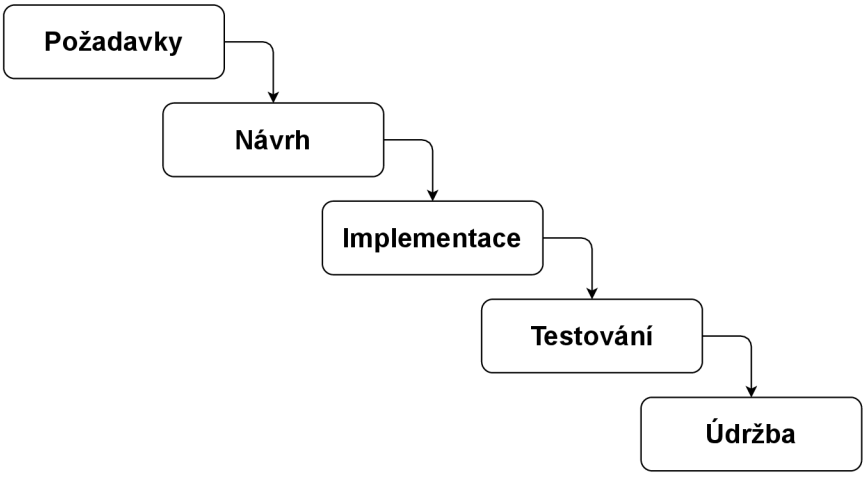
V oblasti webových aplikací nejčastěji rozdělujeme testování na testování backend části a frontend části. Zatímco z hlediska testování backend části se především kontroluje validace vstupních dat a zpracování dat pomocí implementované logiky, stranu frontendu testujeme pomocí simulace či přímým vykonáváním akcí uživatele.

Testování se může využívat také jako přístup k vývoji software. Programování řízené testy (*test-driven development*) je vývojová strategie vyžadující psaní automatických testů před vývojem funkčního kódu. Díky této strategii máme zajištěno, že aplikace je velmi dobře testovatelná a pro každou funkci v software je napsán test [21].

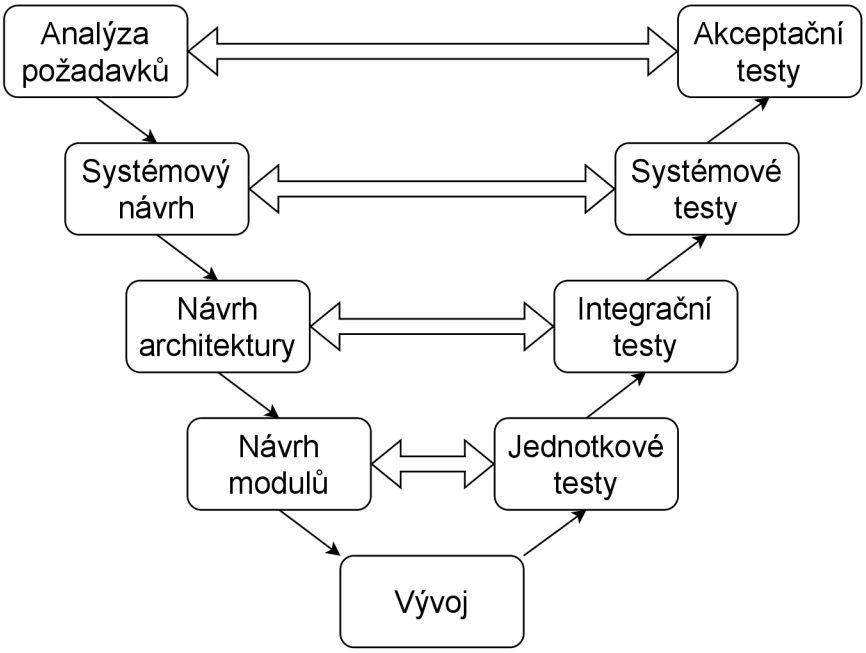
4.2 Životní cyklus vývoje software

Testování hraje v životním cyklu vývoje software velmi důležitou roli. Nejstarším modelem pro vývoj software je vodopádový model. Tento model se řadí mezi sekvenční modely a v modelu se jednotlivé fáze nepřekrývají, což znamená, že se vyžaduje plné dokončení každé fáze předtím, než začne další. Vodopádový model je velmi jednoduchý, ovšem má velké nevýhody. V tomto modelu není možné se v jednotlivých etapách vracet, proto je vhodný pouze pro projekty, které mají jasně předem definované požadavky. Nejčastěji je vodopádový model definován pomocí pěti základních fází, kde testování se nachází v předposlední fázi, po implementaci software a před údržbou (viz. obrázek 4.1) [1].

V-model je modifikovanou verzí vodopádového modelu. V-model udává vztah mezi každou fází vývoje a fází testování a dle tohoto modelu vývojáři a testeři pracují současně. Díky tomu je u tohoto modelu kladen důraz na důkladnou kontrolu a testování produktu. Jednotlivé etapy tohoto modelu a s nimi spojené dané testy jsou zobrazeny na obrázku 4.2 [3].



Obrázek 4.1: Ukázka fází vodopádového modelu (zdroj: vlastní zpracování dle [1])



Obrázek 4.2: Ukázka fází V-modelu (zdroj: vlastní zpracování dle [34])

Agilní model vývoje se uplatňuje v projektech, u kterých je jasný cíl, ale nelze přesně definovat všechny požadavky bez vytvořených prototypů. Cílem agilních metod vývoje bylo zkrátit dobu vývoje rozdělením produktu na překrývající se části. U tohoto modelu vývoj probíhá v iteracích (tzv. *sprinty*), které mají délku nejčastěji v řádu týdnů a po kterých je výsledkem vždy část funkčního software. Oproti vodopádovému modelu se zde klade důraz na komunikaci se zákazníkem a je možné rychle reagovat na změny v požadavcích zákazníka. Taktéž se vodopádový model používá častěji u rozsáhlých projektů, kde je jasně definován proces vývoje, zatímco agilní model se využívá spíše u menších projektů [6].

U agilních modelů se změnil přístup k testování software. Vzhledem k tomu, že oproti dřívějším modelům se čas od zadání nové funkce po její vydání výrazně zkrátil a na produkci se nasazuje velmi často po malých částech, nelze čekat na výsledky manuálního testování.

Proto je u tohoto modelu snaha co největší část testů automatizovat a nejlépe zařadit je do procesu nasazení software [20].

4.3 Druhy testů

Testování, které probíhá na nejnižší úrovni se nazývá jednotkové testování (někdy také testování modulů). Jako jednotky jsou testovány a nalezeny nízkoúrovňové chyby, které jsou následně opraveny, integrovány a integrační testování se provádí následně se skupinami modulů. Tento proces inkrementálního testování následně dává dohromady více a více částí software, dokud se nesloží celý softwarový produkt, či jeho velká část. Ten je pak najednou testován pomocí systémového testování. Pokud softwarový produkt projde úspěšně systémovým testováním, posledním stupněm před definitivním předáním software je provedení akceptačních testů na straně zákazníka.

Díky tomu můžeme snadněji identifikovat jednotlivé chyby nalezené při testování. Pokud je nalezena chyba v jednotkovém testu, musí být problém v jednotce, pro kterou je daný test vytvořen. Pokud je nalezena chyba v integračním testu, musí to souviset s tím, jak dané moduly z daného integračního testu spolu interagují. Díky tomu je testování a ladění software mnohem efektivnější než testování všeho najednou [30].

4.3.1 Jednotkové testy

Jednotkový test izolovaně testuje „jednotku“ kódu a následně porovnává výsledné hodnoty s očekávanými. Za jednotku kódu považujeme část programu, kterou můžeme samostatně izolovaně otestovat bez závislostí na ostatních komponentách. U procedurálního programování může být jednotkou například funkce, zatímco u objektově orientovaného programování nejčastěji považujeme za jednotku třídu, či metodu třídy. Pokud testovaná část využívá ostatní části programu, můžeme je uměle vytvořit a simulovat jejich chování. Tomuto procesu se říká „mockování“ objektu.

Zvýšení popularity objektově orientovaného programování ovlivnilo způsob, jakým programátoři začali přistupovat k testování software. Jednotkové testy se staly u objektově orientovaného programování velmi oblíbené, protože se mohou zaměřovat na testování jednotlivých tříd. Například v jazyku Java je jednotkou obvykle třída a jednotkové testy následně volají jednu či více metod z dané třídy, které produkují výsledky a ty jsou následně automaticky ověřovány. Technika jednotkových testů je velmi důležitá při využití filosofie extrémního programování či programování řízeného testy [28]. Ukázka jednotkového testu v jazyce Java je zobrazena ve výpisu 4.1.

```
class CalculatorTest{
    Calculator calculator;

    @BeforeEach
    void setUp(){
        calculator = new Calculator();
    }

    @Test
    void testAdd(){
        assertEquals(10, calculator.add(4, 6));
    }
```

```
    @Test
    void testMultiply(){
        assertEquals(15, calculator.add(3, 5));
    }
}
```

Výpis 4.1: Ukázka jednotkového testu v jazyce Java

4.3.2 Integrační testy

Jakmile proběhnou úspěšně jednotkové testy a všechny jednotky jsou otestovány, jsou integrovány dohromady. Integrační testování kontroluje správnost rozhraní jednotlivých komponent a spolupráce mezi nimi. Nemusíme ovšem testovat pouze integraci mezi komponentami programu, ale může to být i testování pro ověření komunikace mezi komponentou a operačním systémem. Testování většinou začíná ověřením integrace dvou komponent a postupně se přidávají další.

U malých projektů se občas tento druh testů vynechává, protože software neobsahuje velké množství komponent, a tak se chyby často zjistí při systémovém testování a není tak složité je dohledat. Ovšem jakmile se projekt rozrůstá, integrační testy zvyšují svůj význam, protože umožňují zjistit rychleji a přesněji chyby při integraci daných komponent. Další rozdíl integračního testování a systémového testování je ten, že zatímco systémové testování by mělo probíhat pouze na cílové platformě, integrační testování může probíhat i ve vývojářských prostředích [22].

4.3.3 Systémové testy

Testy, které se provádějí na kompletním a plně integrovaném softwarovém produktu, nazýváme systémové testy. Cílem systémových testů je lokalizovat systémové chyby před uvedením systému do produkčního prostředí [10]. Tento typ testování se provádí na základě systémových požadavků, dokumentů architektury, případě provozního dokumentu. Systémové testování by mělo zajistit, že softwarový produkt funguje v různých operačních systémech, internetových prohlížečích, apod. ve kterých bude následně provozován [12].

4.3.4 End-to-end testy

Pomocí end-to-end testování ověřujeme kompletní aplikaci, se všemi jejími závislostmi. Pro tento typ testování se vytváří prostředí shodné s tím, které budou používat skuteční uživatelé. Poté simulujeme všechny akce, které mohou uživatelé s aplikací provádět. Pomocí end-to-end testování testujeme celé toky procesů – jako je přihlášení na web či nákup produktu z online obchodu [13].

End-to-end testování lze provádět manuálně i automatizovaně. Přestože manuální testování vyžaduje na začátku méně finančních prostředků oproti automatizovanému, považuje se automatizované testování z dlouhodobého hlediska za lepší variantu. U ručního testování může tester provést lidské chyby, které přispívají ke špatnému testování. Automatizované testování oproti tomu proběhne vždy stejně a může vrátit podrobnější informace ve zpětné vazbě výsledků testování.

Tento typ testování se často zaměňuje se systémovými testy. Rozdíl mezi těmito typy testů je ten, že zatímco systémové testování zajišťuje testování všech komponent sys-

tému (bez definovaného pořadí), aby se zajistilo, že odpovídají funkčním požadavkům, tak end-to-end testy provádí testování skutečného toku jednotlivých kroků procesu (v definovaném pořadí). Jako příklad můžeme uvést testování procesu nákupu produktu v internetovém obchodu. U systémového testování zajišťujeme otestování funkčnosti tlačítka pro přidání produktu do košíku, vyvolání API pro akci přidání do košíku a platební službu. U end-to-end testování provádíme kompletní postup z hlediska uživatele. To znamená, že uživatel klikne na tlačítko přidání do košíku → celkový počet produktů v košíku se zvýší → uživatel klikne na košík → pokusí se daný produkt koupit a zaplatit. Můžeme tedy říci, že testování systému najde problémy v jednotlivých komponentách a end-to-end testování najde problémy v toku procesu [31].

4.3.5 Akceptační testy

Poslední úrovní testování je provedení akceptačních testů. Tyto testy jsou prováděny na straně zákazníka a reprezentují zájmy zákazníka. Akceptační testy dávají zákazníkovi jistotu, že vytvořený software má požadované vlastnosti a chová se správně. Tyto testy můžeme považovat za „smlouvu“ mezi zákazníkem a dodavatelem software, kdy definitivní předání software může proběhnout až po splnění všech akceptačních testů a tím se ověří, že se software chová tak, jak zákazník požadoval [24].

4.4 Nástroje pro testování

Vzhledem k popularitě jazyků JavaScript/TypeScript existuje rozsáhlé množství nástrojů k jejich testování. V této sekci si představíme nejznámější z těchto nástrojů. Testovací frameworky Jest, Mocha a Jasmine se zaměřují na testování zdrojového kódu a Cypress společně se Selenium slouží k end-to-end testování. End-to-end testování slouží k testování frontendových částí aplikací tak, že simulují akce vyvolané uživatelem.

4.4.1 Jest

Jest[1] je testovací framework pro jazyk JavaScript. Je vyvíjen společností Meta (dříve Facebook) a původně byl určen pro testování projektů využívajících React. Tento framework se soustředí na co největší jednoduchost používání a i díky tomu je jeden z nejpopulárnější frameworků používaných pro testování projektů v jazyce JavaScript. Jest umí spolupracovat s projekty využívajícími Babel, TypeScript, Node a mnoho dalších. Samozřejmostí je i spolupráce s nejznámějšími frontendovými frameworky, jako jsou React, Angular či Vue. Jest je široce používán ve velkých společnostech jako Twitter, Spotify či Airbnb. Ukázku jednoduchého jednotkového testu ve frameworku Jest můžeme vidět ve výpisu 4.2.

Díky tomu, že se Jest zaměřuje na jednoduchost používání, je většinou připraven na práci ve většině JavaScriptových projektech ihned po jeho instalaci, a to bez nutnosti jakékoliv konfigurace. Díky zajištění unikátního globálního stavu pro testy umožňuje Jest spouštění testů paralelně. Aby bylo testování ještě rychlejší, Jest spouští nejdříve neúspěšné testy a dále organizuje pořadí testů na základě minulých zkušeností, jak dlouho jednotlivé testy trvaly.

Tento framework umožňuje také jednoduše zjistit pokrytí zdrojového kódu testy. Není pro to potřeba žádná konfigurace a umožňuje zjistit pokrytí kódu testy z celého projektu,

[1] https://jestjs.io/

nejen z testovaných částí. Jest poskytuje širokou škálu funkcí pro testování, které je možno díky dobře udržované dokumentaci jednoduše používat [4].

```
const { UserRepository, User } = require('./user');

describe('User unit tests', () => {
    let userRepository;
    let user;

    beforeEach(() => {
        jest.restoreAllMocks();
        userRepository = new UserRepository();
        user = new User('John');
    });

    it('Should return a single user', async () => {
        jest.spyOn(userRepository, 'findOne').mockResolvedValue(user);
        const result = await userRepository.findOne('John');

        expect(result).toBeInstanceOf(User);
        expect(result).toEqual(user);
    });
});
```

Výpis 4.2: Ukázka jednotkového testu pro otestování objektu uživatele a jeho repozitáře (komponenta, která zapouzdřuje logiku pro přístup ke zdrojům dat) ve frameworku Jest. Nejprve se ve funkci `beforeEach` vytvoří repozitář pro přístup ke zdroji s uživateli a vytvoří se objekt uživatele se jménem John. Následně ve funkci `it` nastavíme chování metody `findOne` repozitáře tak, aby se při zavolání této metody vracel uživatel v proměnné `user`. Tato funkcionalita se nazývá *mockování* a využívá se k simulaci chování metod. Následně zavoláme *namockovanou* metodu `findOne` a v posledních dvou řádcích kontrolujeme pomocí funkce `expect`, zdali je navrácený objekt pomocí metody `findOne` v proměnné `result` opravdu typu uživatel a zdali se shoduje s objektem v proměnné `user`.

4.4.2 Mocha

Mocha[2] je testovací framework fungující na Node.js. Na rozdíl od frameworku Jest zde probíhají testy sériově, nikoliv paralelně. Mocha se zaměřuje na širokou škálu testů od jednotkových testů, přes integrační až po end-to-end testování, díky čemuž se běžně používá jak pro frontend, tak pro backend testování. Mocha neumožňuje používání ihned po instalaci podobně jako je tomu u frameworku Jest, většinou vyžaduje ještě dodatečnou instalaci příslušných knihoven a vytvoření konfiguračních souborů.

Například pro vytvoření jednoduchého jednotkového testu s podporou mockování, Mocha vyžaduje použití vyhodnocovací knihovny, což může být například knihovna *Chai* a mockovací knihovny – například knihovny *Sinon*. Na druhou stranu, použití dodatečných knihoven nepřináší pouze nevýhody, protože například vyhodnocovací knihovna Chai přináší mnohem větší flexibilitu oproti vyhodnocovacím funkcím frameworku Jest, a to například možnost použití logických operátorů jako je „a“ či „nebo“. Obecně se Mocha dopo-

[2] https://mochajs.org/

ručuje pro použití ve velkých projektech, pro které se hodí díky své flexibilitě a různorodosti knihoven, které jsou dostupné pro zkušenější vývojáře. Ukázku jednoduchého jednotkového testu ve frameworku Mocha můžeme vidět ve výpisu 4.3.

```
const sinon = require('sinon');
const { UserRepository, User } = require('./user');

describe('User unit tests', () => {
    let userRepository;
    let user;

    beforeEach(() => {
        sinon.restore();
        userRepository = new UserRepository();
        user = new User('John');
    });

    it('Should return a single user', async () => {
        sinon.stub(userRepository, 'findOne').resolves(user);

        expect(await userRepository.findOne('John')).to.be.instanceOf(User)
          .and.to.equal(user);
    });
});
```

Výpis 4.3: Ukázka jednotkového testu ve frameworku Mocha. Logika testu je shodná s ukázkou testu 4.2 ve frameworku Jest. Rozdílem je zde mírně odlišný způsob *mockování* pomocí komponenty `sinon` a využití logických spojek ve funkci `expect`, které framework Mocha umožňuje.

4.4.3 Jasmine

Jasmine[3] je testovací framework určený primárně pro testování Angular aplikací. Jasmine využívá knihovnu *Karma*, která umožňuje spouštění testů na několika prohlížečích a dokonce i zařízeních, aby bylo zajištěno fungování kódu na všech podporovaných platformách. Tento framework by se dal považovat za takový střed mezi frameworky Jest a Mocha, jelikož stejně jako Jest v sobě již obsahuje knihovny pro vyhodnocování či mockování, ale na druhou stranu není tak rychlý jako Jest. Proti Mocha není Jasmine tak flexibilní framework, ale jak již bylo zmíněno, obsahuje již knihovny pro vyhodnocování či mockování, a tak vývojář nestráví tolik času nad zajištěním potřebných závislostí a konfigurací [35].

4.4.4 Cypress

Cypress[4] je testovací framework sloužící pro end-to-end testování. Většina end-to-end testovacích nástrojů funguje tak, že běží mimo prohlížeč a spouští vzdálené příkazy přes síť. Cypress využívá odlišnou architekturu, při které se spouští ve stejné smyčce běhu jako aplikace. Na pozadí testovacího frameworku je proces Node.js, se kterým neustále komunikuje,

[3] https://jasmine.github.io/
[4] https://www.cypress.io/

synchronizuje se a provádí úkoly jeden za druhým. Tento framework také funguje na síťové vrstvě tak, že čte a upravuje síťový provoz webu za běhu. To nám například umožňuje upravovat vše, co přichází a odchází z prohlížeče. Taktéž má Cypress nativní přístup ke každému jednotlivému objektu, a to ať už je to okno, dokument, prvek DOM či jakákoliv funkce.

Díky těmto třem vlastnostem se nabízí nové možnosti testování, které předtím nebyly možné. Cypress nám díky tomu umožňuje změnit jakékoliv aspekty naší aplikace. Namísto pomalých a drahých testů, pomocí vytvoření stavu požadovaného pro danou situaci, jej můžeme *mockovat* (vytvořit tento stav uměle) stejně, jako to můžeme udělat v jednotkovém testu. Můžeme díky tomu například programově měnit stav naší aplikace (např. Redux) přímo z testovacího kódu či otestovat, jak aplikace zareaguje na chyby z našeho serveru, změnou stavových kódů odpovědí na požadavky na kód 500.

Mezi další velkou výhodu knihovny Cypress patří to, že ví o všem, co se v naší aplikaci děje synchronně. Je upozorněn v okamžiku načtení, takže není možné, aby minul prvky, když spouští události. Pokud se začne přecházet na jinou stránku, pozastaví provádění příkazu, dokud není následující stránka plně načtena. Můžeme dokonce přímo říci, aby počkal na dokončení konkrétních síťových požadavků. Cypress provádí drtivou většinu příkazů uvnitř prohlížeče, takže nedochází k žádnému zpoždění sítě. Příkazy se spouštějí a řídí aplikaci tak rychle, jak je schopna se vykreslovat. Jsme zcela izolováni od nutnosti a zmatků s ručním čekáním či opakováním.

Tento framework byl postaven především pro co nejlepší použitelnost. Obsahuje stovky chybových zpráv, které přesně popisují důvod, proč na daném testu selhal. K dispozici je uživatelské rozhraní (viz. obrázek 4.3), které nám vizuálně zobrazuje provádění jednotlivých příkazů, vyhodnocování výsledků, síťové požadavky, změny URL a mnoho dalšího. Během provádění testů jsou pořizovány snímky aplikace, což umožňuje cestovat časem zpět do stavu, ve kterém byl, když byly spuštěny dané příkazy. Zatímco se provádí testy, můžeme mít zapnuté vývojářské nástroje prohlížeče, a tak vidět každou zprávu konzole nebo každý síťový požadavek [26].

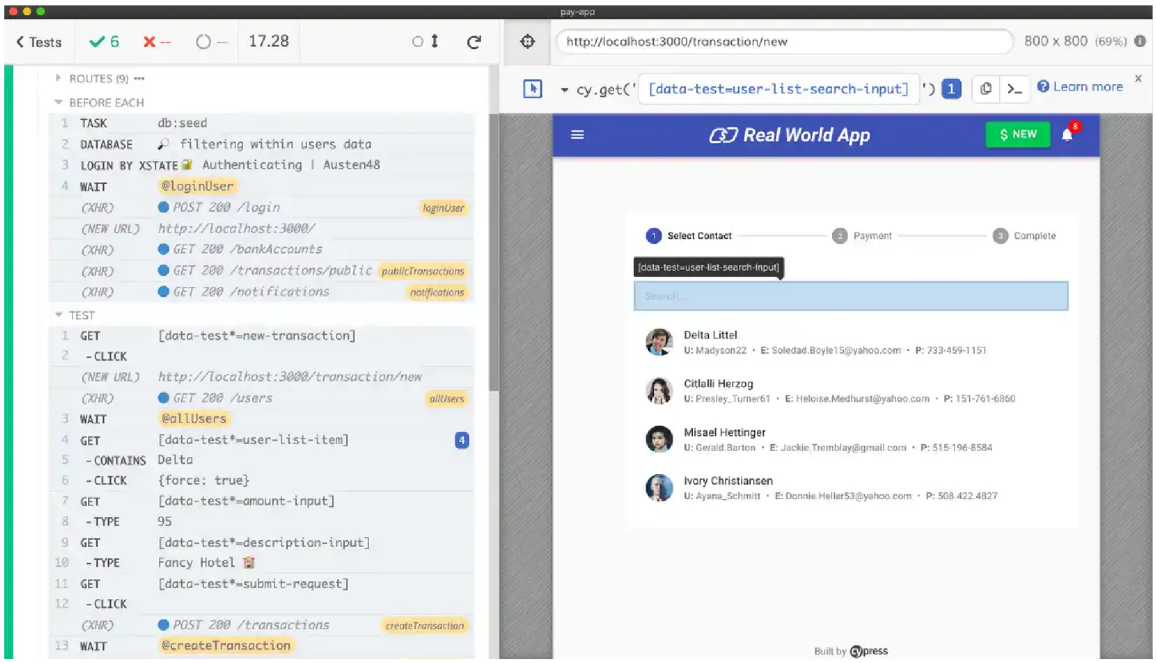
Testy se píší v JavaScriptu, díky čemuž je testovací kód nakonec spuštěn přímo uvnitř samotného prohlížeče. Cypress v sobě obsahuje nástroje, které jsou pro end-to-end testy vyžadovány a obvykle musí být instalovány samostatně, jako jsou knihovny Mocha či Chai pro vyhodnocení testů. Mezi podporované prohlížeče pro testování patří Chrome, Firefox, Edge, Electron a Brave.

4.4.5 Selenium

Selenium[5] je sada komponent zaměřených především na automatizované testování webových aplikací. Skládá se ze tří součástí – Selenium IDE, Selenium WebDriver a Selenium Grid. Selenium umožňuje spouštět testy v různých prohlížečích, jako je například Firefox, Chrome, Internet Explorer a mnoho dalších a je možno jej použít z řady programovacích jazyků, jako jsou Java, Perl, PHP, C#, Python a mnoho dalších.

Selenium používá k popisu jednotlivých kroků testování jazyk zvaný Selenese. Selenium IDE je rozšíření prohlížeče, které umožňuje zaznamenávání akcí uživatele a jejich převedení do příkazů v jazyce Selenese. Toto rozšíření je k dispozici pro prohlížeče Google Chrome a Mozilla Firefox. Rozšíření zaznamenává jednotlivé příkazy tak, že uživatel spustí záznam příkazů a následně provede sekvenci akcí, které chce zaznamenat jako testovací případ. Po ukončení záznamu se následně zaznamenané akce převedou do jazyka Selenese a vy-

[5] https://www.selenium.dev/



Obrázek 4.3: Uživatelské rozhraní nástroje Cypress[6]

tvořený testovací případ se uloží. Testovací případ je možné následně znovu spustit či jej vyexportovat do programovacích jazyků jako je C#, Java, Ruby či Python.

Selenium Web Driver je uváděn jako následník nástroje Selenium RC, který následně kompletně nahradil. Selenium Web Driver eliminoval nevýhody, které měl Selenium RC. Hlavní nevýhoda, která byla odstraněna vůči Selenium RC byla ta, že pro provádění testovacích případů není nutno mít spuštěn Selenium Server, který fungoval jako HTTP proxy, spouštěl a vypínal prohlížeče a interpretoval v nich Selenium příkazy. Selenium Web Driver místo toho komunikuje přímo s prohlížečem a používá jeho nativní podporu pro automatizované provádění testovacích případů [33].

Na obrázku 4.4 se nachází architektura Selenium Web Driveru. Můžeme na něm vidět, že v prvním kroku se kód Selenium napsaný v některém z podporovaných programovacích jazyků převede na JSON formát. Tento JSON je zaslán ovladači prohlížeče pomocí HTTP protokolu (každý prohlížeč má svůj ovladač – například pro Google Chrome je to `chromedriver`, pro Firefox `geckodriver`). Ovladač prohlížeče následně spouští jednotlivé příkazy z obdrženého JSONu na příslušném prohlížeči. Prohlížeč vykoná danou akci a zašle odpověď zpět ovladači. Ovladač zabalí odpověď do JSON formátu a zašle ji uživatelskému rozhraní (například Eclipse IDE), které zobrazí výsledky testování.

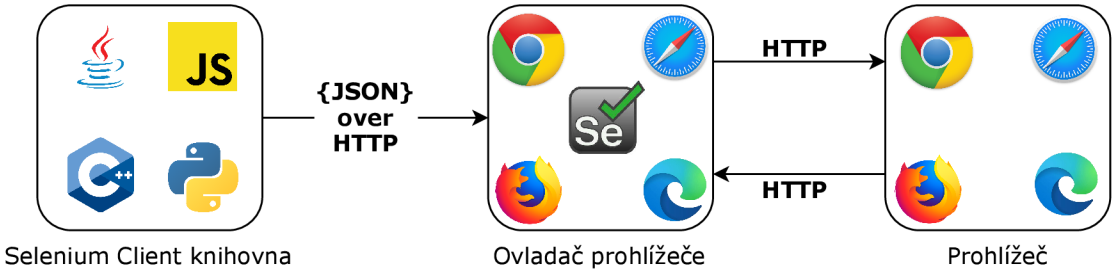
Nevýhodou při použití Web Driver je nutnost upravovat testy pro každý prohlížeč separátně, protože testy vyžadují specifikovat v jakém prohlížeči budou probíhat a jaká verze Web Driveru by se měla vytvořit. Při vytváření testů pomocí Selenium musíme taktéž implementovat čekání před prováděním příkazů, například čekání na načtení stránky, které se dá nastavit implicitně (počet sekund) či explicitně (dokud neexistuje nějaký element).

Pokud chceme škálovat spouštění testů a spravovat je z centrálního místa, můžeme použít nástroj Selenium Grid. Cílem tohoto nástroje je poskytnout snadný způsob paralelního

[6]Zdroj: Cypress.io https://www.cypress.io/features

spouštění testů na více strojích. To nám usnadňuje spouštění testů na obrovské kombinaci prohlížečů či operačních systémů [18].

Mezi hlavní výhody Selenium patří především podpora velkého množství prohlížečů a jejich verzí a možnost použít jej s různými programovacími jazyky. Také má velmi rozsáhlou komunitu díky delší existenci. Mezi největší nevýhody v porovnání s výše zmíněným nástrojem Cypress můžeme považovat složitější nastavení a menší výkonnost při provádění testů.



Obrázek 4.4: Architektura Selenium WebDriver (zdroj: vlastní zpracování dle [44])

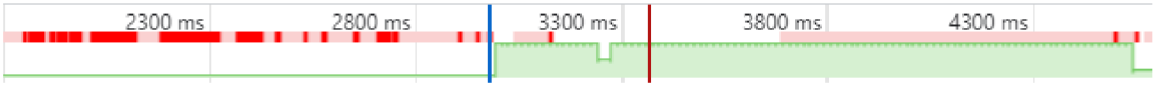
4.5 Profilování

Profilování je forma dynamické analýzy programu, která měří prostorovou (paměťovou) nebo časovou složitost programu, použití konkrétních instrukcí či frekvenci a trvání volání jednotlivých funkcí. Tyto informace poté může programátor využít k optimalizaci programu [46].

V prostředí webových aplikací se pro profilování nejčastěji používají nástroje, které jsou vestavěné v prohlížečích. U prohlížeče Google Chrome je tento nástroj pojmenován jako Chrome DevTools, alternativou může být Firefox Developer Tools v prohlížeči Firefox a podobně.

V nástroji Chrome DevTools nám pro profilování webové aplikace nejlépe poslouží karta „Výkon“. Na této kartě můžeme buď ručně spustit profilování webu a následně ho zastavit, pokud se chceme zaměřit na danou činnost v aplikaci nebo můžeme spustit profilování při znovu načtení stránky. To způsobí, že se spustí profilování, znovu se začne načítat stránka a profilování se následně automaticky zastaví, jakmile se zastaví vytížení CPU a síťová aktivita stránky.

Výsledek profilování obsahuje velké množství údajů, které lze rozdělit na několik sekcí, z nichž ty nejdůležitější jsou zde dále rozebrány. V horní části se nachází graf snímků za sekundu (viz. obrázek 4.5). Zde můžeme detekovat propady snímků na webu způsobené prováděním různých animací apod. Pro uživatelsky nejpříjemnější používání by měly všechny animace běžet v 60 snímcích za sekundu. Pokud v tomto grafu vidíme červenou čáru, tak v tu chvíli se značně propadly snímky, což může snižovat uživatelský dojem. Na zeleném grafu vidíme počty snímků za sekundu, čím je sloupec vyšší, tím byly snímky za sekundu vyšší. S tímto grafem souvisí také sekce „Snímky“, ve které lze zobrazit například délky trvání jednotlivých propadů snímků za sekundu.



Obrázek 4.5: Graf snímků za sekundu (FPS)

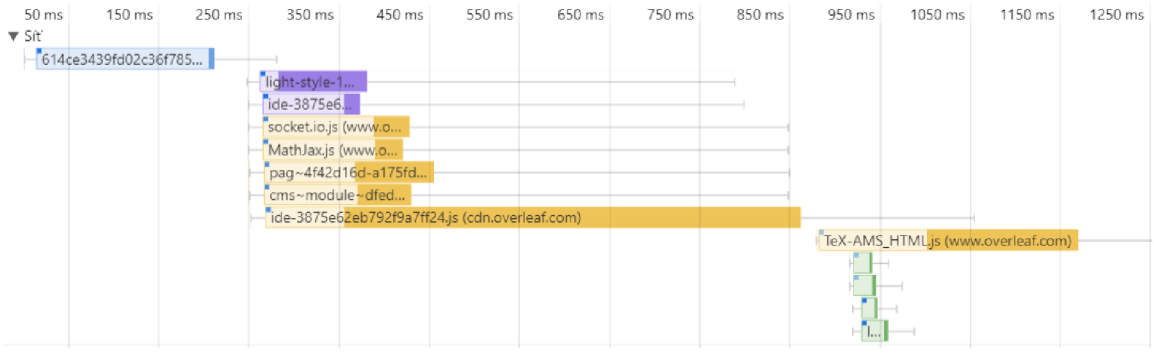
Pod grafem snímků za sekundu se nachází graf využití CPU (viz. obrázek 4.6). V grafu se nachází více barev, které značí, co za typ úlohy v tu chvíli využívalo CPU. Pro příklad žlutá barva značí práci s JavaScriptem, fialová práci s rozložením (vykreslování) a šedá systémové úlohy. Po dokončení načtení stránky by mělo být vytížení CPU minimální, pokud tomu tak není, uživatelé (obzvláště na pomalejších zařízeních) mohou pocítit pomalejší reakce stránky při interakcích.

Obrázek 4.6: Graf vytížení CPU

Další sekcí je sekce „filmový pás“ (viz. obrázek 4.7). V této sekci můžeme vidět proces vykreslování stránky záznamu profilování. To je zajištěno díky tomu, že při profilování se periodicky zaznamenávají snímky obrazovky a ty se následně v této sekci zobrazí a přiřadí k časové ose. Následně při najetí myší na příslušný snímek můžeme vidět detail, jak byla stránka vykreslena v daný čas.

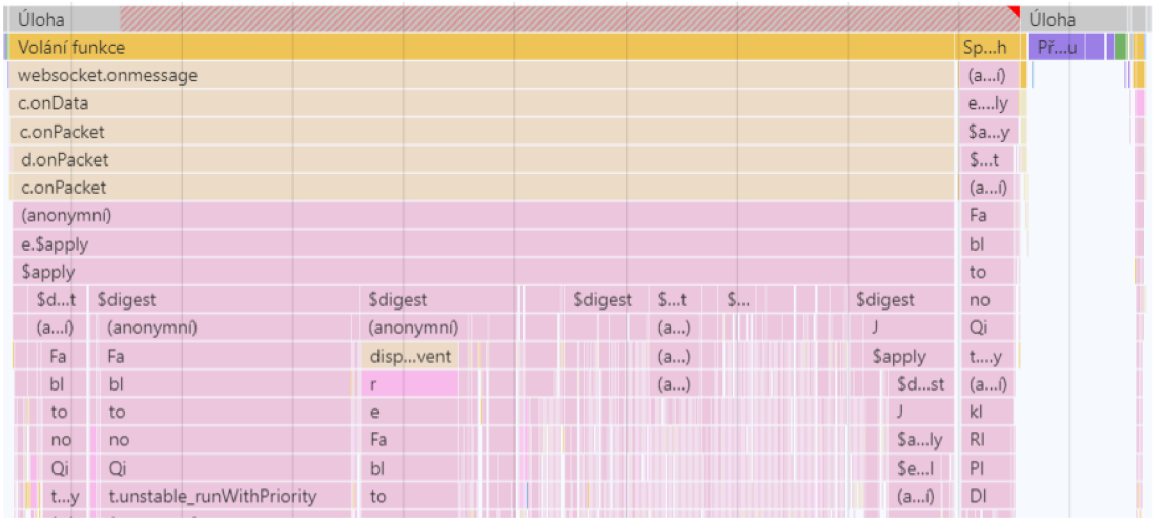
Obrázek 4.7: Snímky obrazovky při vykreslování

Sekce s časovou osou síťových požadavků (viz. obrázek 4.8) zobrazuje jednotlivé síťové požadavky, které při profilování probíhaly. Pro jednotlivé požadavky je možné zobrazit si jejich detail, kde můžeme vidět podrobnosti jako přesnou délku načítání, metodu požadavku, typ MIME či velikost načítaného zdroje.



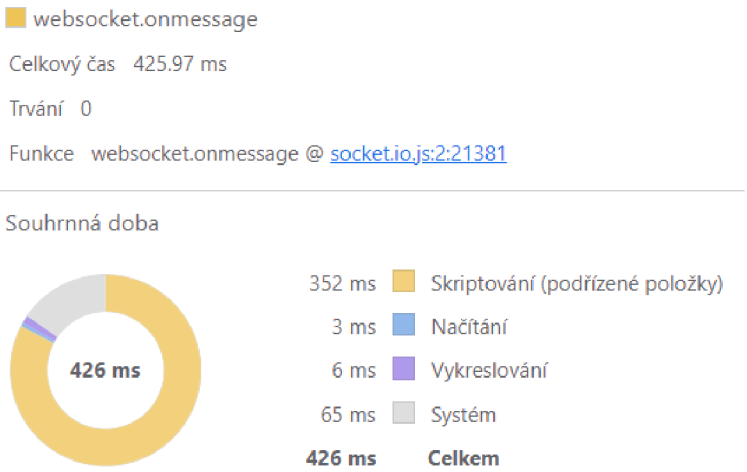
Obrázek 4.8: Časová osa síťových požadavků

Hlavní část na obrazovce s výsledky profilování se nazývá „Hlavní“ (viz. obrázek 4.9). V této sekci se zobrazují jednotlivé úlohy a volání JavaScriptových funkcí (včetně celého stromu volání jednotlivých funkcí). Úlohy je možné rozkliknout a zobrazit si o nich další podrobnosti.



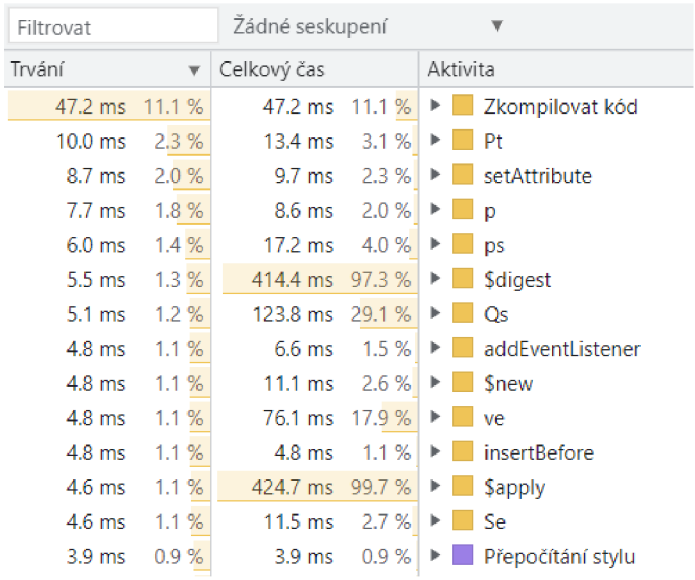
Obrázek 4.9: Hlavní sekce zobrazující jednotlivé úlohy probíhající při profilování

Při zobrazení podrobností o úloze se v dolní části okna s výsledky profilování zobrazí sekce „Souhrn“ (viz. obrázek 4.10). Zde můžeme vidět celkový čas trvání úlohy, v případě JavaScriptové funkce je možné prokliknout se do souboru, kde se funkce nachází. Dále je zde uvedena souhrnná doba, kde je celkový čas trvání úlohy rozdělen do několika částí – například čas provádění skriptu, načtení skriptu, vykreslování apod.



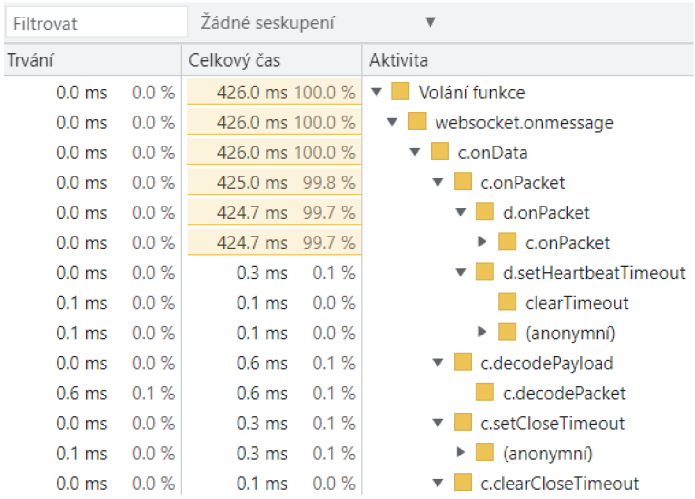
Obrázek 4.10: Souhrn zobrazující podrobnější informace o dané úloze

Další kartou při zobrazení podrobností o dané úloze je karta s názvem „Zdola nahoru“ (viz. obrázek 4.11). Zde se zobrazují volání funkcí z pohledu zdola-nahoru, proto se zde často objevuje volání nativních funkcí prohlížeče. Tyto volání nízkoúrovňových funkcí můžeme rozbalit a zjistit, jaký kód je volá a jaká je doba jejich trvání.



Obrázek 4.11: Volání funkcí z pohledu zdola-nahoru

Podobnou kartou kartě „Zdola nahoru“ je karta „Strom volání“ (viz. obrázek 4.12). Zde vidíme strom volání JavaScriptových funkcích shoda-dolů a jejich časy vykonávání. Můžeme zde najít místa ve zdrojovém kódu vhodná pro optimalizaci – obvykle místa, kde prohlížeč stráví velké množství času.



Obrázek 4.12: Karta stromu volání funkcí

Kapitola 5

Analýza problému

V této kapitole bude definována cílová skupina uživatelů a průzkum potřeb, které mají. Pomocí nich bude definován problém a proveden průzkum aktuálních možných řešení problému.

5.1 Cílová skupina uživatelů a jejich potřeby

Cílovou skupinou uživatelů, kterých se týká daná problematika testování knihovny Geovisto, jsou vývojáři této knihovny. Jejich hlavní potřebou je co nejefektivněji otestovat funkčnost knihovny při vývoji nových funkcionalit či úpravách těch stávajících. Pokud například vývojář knihovny pracuje na tvorbě nového nástroje, může být nucen provést změny či rozšířit jádro knihovny. Ovšem na tomto jádře závisí i ostatní nástroje knihovny, u kterých by mohlo dojít k porušení zpětné kompatibility a tyto nástroje by následně nemusely fungovat správně. Ovšem testování není třeba pouze při vývoji nových funkcionalit, ale mělo by se provádět po provedení jakýkoliv změn ve zdrojovém kódu tak, aby bylo zajištěno správné fungování knihovny.

Pro řešení této potřeby by potřebovali rozhraní, které by jim poskytlo přehled testů, které jsou k dispozici a tyto testy si mohli opakovaně spustit za účelem zjištění, zdali správně fungují jednotlivé části knihovny. Po dokončení testů by rozhraní mělo poskytnout přehled o průběhu testů a v případě, že v testu nastala chyba, dát uživateli podrobnější informace o této chybě. Pro testy, které se zaměřují na testování uživatelského rozhraní pomocí simulace uživatelských akcí, by mělo rozhraní poskytnout možnost podrobnějšího detailu průběhu testu v podobě snímku obrazovky či videa, jak daný test probíhal.

Je žádoucí, aby rozhraní již obsahovalo základní testy pro otestování funkcí, které jsou uživateli knihovny nejčastěji využívány. Mezi tyto funkce patří nejen základní operace s jádrem knihovny, jako jsou načítání datových sad či konfigurací, ale tyto testy by měly pokrývat ověření správného fungování jednotlivých nástrojů, které mohou být do jádra knihovny integrovány. Vhodným typem testů pro tyto nástroje jsou testy užívání jednotlivých nástrojů pomocí uživatelského rozhraní. Rozhraní by rovněž mělo umožňovat jednoduchou integraci nových testů, které budou v budoucnu vytvářeny jak pro existující nástroje, tak pro nástroje, které budou dále vytvářeny.

U jednotlivých testů by měly být připraveny datové sady a konfigurace, které jsou vhodné pro otestování funkcionality, na kterou se daný test zaměřuje. Tyto datové sady společně s konfiguracemi by rozhraní mělo umožnit změnit. Pro lepší přehled o případných problémech, které nastávají u některých funkcí, by rozhraní mělo umožňovat zobrazení his-

torických výsledků testů. V této části pro zobrazení minulých běhů testů by mělo rozhraní zobrazit datum, kdy jednotlivé testy byly spuštěny, kterým uživatelem byly spuštěny a s jakou konfigurací a datovou sadou probíhaly. V případě, že v průběhu testu nastala chyba, měly by tyto chyby být ukládány, aby mohly být zpětně zobrazeny. Pro vývojáře je taktéž důležité mít způsob, pomocí kterého by ověřili, že případné změny implementace nezhoršily výkonnost jádra knihovny či některého z nástrojů.

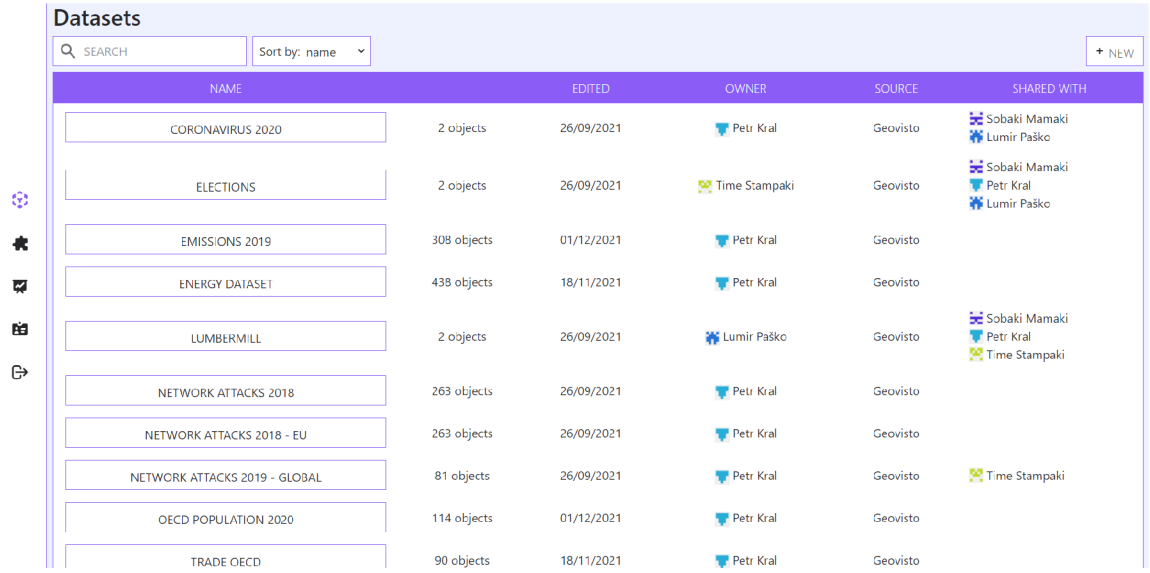
5.2 Aktuální řešení

Aktuálním způsobem testování knihovny Geovisto a jejich nástrojů je ruční testování. To znamená, že při jakýkoliv změnách v implementaci knihovny je nutné ručním způsobem projít a zkontrolovat všechny funkce, které by mohly být změnami ovlivněny. Pokud jsou změny implementace provedeny v jednotlivých nástrojích, mělo by vývojáři stačit zkontrolovat správné fungování daného nástroje společně s jádrem. Pokud jsou ovšem změny provedeny přímo v jádru knihovny Geovisto, je nutné otestovat nejen správnou funkčnost jádra, ale i všech nástrojů, které mohou být do knihovny integrovány. Těchto nástrojů nyní v době psaní práce existuje osm a několik dalších je ve vývoji.

5.3 Definice problému

Hlavním problémem je tak neexistující způsob efektivního testování knihovny Geovisto. Vzhledem k tomu, že knihovna je již nyní velmi rozsáhlá a stále se její rozsáhlost zvětšuje tvorbou nových nástrojů, je metoda ručního testování pro knihovnu z dlouhodobého hlediska neudržitelná. Ruční testování sebou taktéž nese rizika v podobě zapomenutí otestování některých funkcionalit či přehlédnutí chyb, které se vyskytují. Proto je nutnost při vývoji této knihovny přejít z ručního testování na automatizované, které zvládne vykonat větší množství testů za kratší čas, poskytuje přehledné výsledky jednotlivých testů a eliminuje možná rizika v podobě lidského pochybení přehlédnutí chyby při testování či zapomenutí otestování některé z funkcionalit.

Vzhledem k tomu, že Geovisto je poměrně specifická knihovna, neexistuje žádné již existující řešení, které by se dalo pro řešení tohoto problému využít. Jediným aktuálním možným ulehčením při řešení tohoto problému, je možnost využít již existující informační systém pro správu vizualizací geografických dat (viz. obrázek 5.1), který umožňuje spravovat datové sady a nad nimi vytvořené konfigurace pro opakované použití [16]. Vývojář tak může mít připraveny datové sady s konfiguracemi, které jsou vhodné pro testování jednotlivých funkcionalit (nástrojů) knihovny. Vzhledem k tomu, že nutnost mít připravené vhodné datové sady a jejich konfigurace je potřeba i při automatizovaném testování, bylo by ideální zakomponovat část automatizovaného testování do tohoto informačního systému.



Obrázek 5.1: Ukázka informačního systému pro správu datových sad a konfigurací

5.4 Definice požadavků

Vzhledem k výše zmíněným potřebám uživatelů a vymezení problému byly definovány následující požadavky:

- Vytvoření rozhraní pro přehled existujících testů s možností jednotlivé testy spustit a zobrazit si jejich výsledek.
- Možnost zobrazení si výsledků minulých běhů testů s jejich podrobnostmi.
- Vytvoření základní sady testů pro otestování klíčových funkcí jádra knihovny a jednotlivých nástrojů, testování nástrojů pomocí uživatelského rozhraní knihovny.
- Připravení vhodných datových sad a konfigurací, které jsou vhodné pro vytvořenou sadu testů.
- Možnost změnit si pro jednotlivé testy konfigurace, se kterými budou probíhat.
- Vytvoření způsobu pro ověření výkonnosti jádra knihovny a jejich nástrojů.

Kapitola 6

Návrh řešení

V této kapitole je popsána architektura implementace řešící definovaný problém. Dále je zde popis testů, které budou implementovány, s jakými datovými sadami a konfiguracemi budou provozovány a které nástroje knihovny nad nimi budou testovány.

6.1 Architektura

Vzhledem k existenci informačního systému, který umožňuje správu datových sad a jejich konfigurací, které jsou potřeba i k automatizovanému testování, jsem se rozhodl přidat část pro automatizované testování knihovny Geovisto do tohoto informačního systému (viz. obrázek 6.1). Pro testování knihovny se budou provádět jednotkové a end-to-end testy. Pomocí jednotkových testů se budou ověřovat především správné výpočty a korektní načítání konfigurací a datových sad. End-to-end testy budou ověřovat správnou funkčnost uživatelského rozhraní knihovny a zobrazení správných údajů ve vizualizacích při daných konfiguracích.

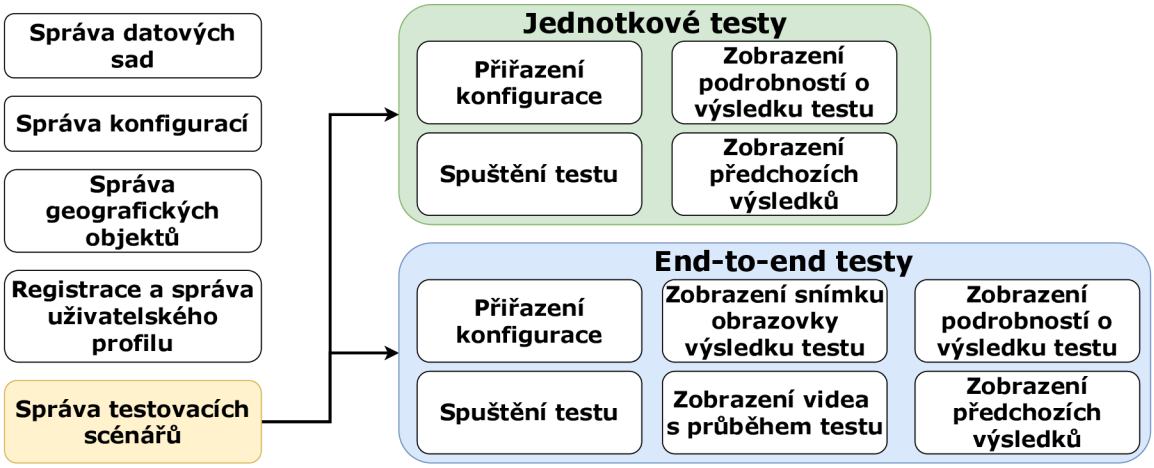
Zobrazení výsledků testování se bude zobrazovat v části „testování“ v informačním systému pro správu datových sad a jejich konfigurací. Bude zde možnost spustit kompletní testování knihovny, které zaručí spuštění všech implementovaných testů či možnost spuštění jednotlivých testů. Po dokončení testování se zobrazí výsledky jednotlivých testů způsobem ikony pro úspěch či neúspěch a v případě neúspěšného průběhu testu bude možnost zobrazit si podrobnosti s důvody, proč daný test selhal. Taktéž budou v databázi uloženy historické výsledky testování, které si uživatel může zpětně zobrazit.

6.2 Typy testů

Pro testování knihovny budou využity jednotkové a end-to-end testy. Níže jsou stručně popsány jednotlivé testy, jaké nástroje budou testy pokryty a nad kterými datovými sadami a konfiguracemi budou spouštěny. Pro zjednodušení popisu budou datové sady vyjádřeny následujícími zkratkami:

- Emise – zdroje znečištění umělých emisních skleníkových plynů a emisí plynů dělené pro jednotlivé země v roce 2019. Data obsahují zkratku země ve formátu ISO 3166-1 alpha-3, druh znečištění (látka) a množství dané látky v ovzduší.[1]

[1] https://stats.oecd.org/



Obrázek 6.1: Schéma informačního systému obsahující rozšíření pro automatizované testování

- Populace – populace v jednotlivých zemích v roce 2020. Data obsahují zkratku země ve formátu ISO 3166-1 alpha-3, pohlaví a množství populace daného pohlaví v dané zemi.[2]
- Obchod služeb – množství obchodu služeb mezi jednotlivými zeměmi za rok 2019. Data obsahují zkratku země exportéra a importéra ve formátu ISO 3166-1 alpha-3 a množství obchodovaných služeb.[3]
- Obchod – rozsáhlý zdroj dat obchodů mezi jednotlivými zeměmi. Pro více druhů zboží a více let. Data obsahují zkratku země exportéra a importéra ve formátu ISO 3166-1 alpha-3, druh zboží, množství zboží a rok pro který platí daná data.[4]

6.2.1 Jednotkové testy

V této sekci se nachází výčet jednotkových testů, které budou implementovány a testovány. U jednotlivých testů je popis, které funkce daný test kontroluje a jakým způsobem to provádí.

Načtení datové sady

Tento jednotkový test kontroluje, zda knihovna správně načítá datové sady. V testu se nejprve načte do knihovny Geovisto datová sada ve formě JSON souboru. Následně se získá z instance knihovny pole objektů představující záznamy datové sady, které byly vytvořeny při načtení datové sady. U těchto objektů se kontroluje, zdali souhlasí jejich počet s počtem objektů v JSON souboru datové sady a jestli se všechny objekty z JSON souboru v poli nacházejí.

[2] https://stats.oecd.org/
[3] https://stats.oecd.org/
[4] https://resourcetrade.earth/

Testovaný nástroj	Jádro
Datová sada	Obchod služeb
Očekávaný výstup	Úspěšné načtení datové daty, kontrola korektnosti dat
Postup	1. Spuštění nové instance. 2. Načtení datové sady „Jádro“ ve formě JSON souboru. 3. Kontrola, zdali načtené objekty odpovídají objektům v JSON souboru s datovou sadou.

Načtení konfigurace s vrstvami

Jednotkový test pro ověření načtení konfigurace s vrstvami kontroluje správné načtení JSON souboru s konfiguracemi jednotlivých nástrojů knihovny. Test nejprve načte do knihovny Geovisto konfiguraci společně s datovou sadou ve formě JSON souboru. Následně se z instance knihovny získají konfigurace jednotlivých nástrojů. U takto získaných konfigurací se kontroluje, zdali jejich hodnoty odpovídají hodnotám ve zdrojovém JSON souboru s konfiguracemi, který byl do knihovny předtím načten.

Testovaný nástroj	Všechny
Datová sada	Populace
Očekávaný výstup	Úspěšné načtení konfigurace, aktivní všechny dostupné vrstvy
Postup	1. Spuštění nové instance. 2. Načtení datové sady „Populace“ ve formě JSON souboru. 3. Načtení připravené konfigurace ve formě JSON souboru, která má aktivní všechny mapové vrstvy – kartogram, vrstvu se značkami a spojovací vrstvu. 4. Kontrola, zdali jednotlivé nástroje s mapovými vrstvami mají korektní nastavení hodnot (datových domén), které se nacházejí v JSON souboru s konfigurací.

6.2.2 End-to-end testy

V této sekci se nachází výčet end-to-end testů, které budou implementovány a testovány. Stejně jako u jednotkových testů je u každého testu stručný popis, které funkce daný test kontroluje a jakým způsobem to provádí.

Aktivace jednotlivých vrstev

End-to-end test pro kontrolu aktivace jednotlivých vrstev ověřuje, zdali lze provést aktivování jednotlivých vrstev pomocí uživatelského rozhraní – konkrétně kartogramu, vrstvy se značkami a spojovací vrstvy. Pro tento test je nutné použít konfiguraci, která má nastavené datové domény pro tyto tři nástroje. Test nejprve otevře nástroj postranního panelu a v něm postupně rozbaluje záložky pro jednotlivé vrstvy. Při každém rozbalení záložky pro vrstvu aktivuje vrstvu pomocí zaškrtávacího políčka v záhlaví záložky. Po rozbalení všech záložek s jednotlivými vrstvami a jejich aktivaci se postranní panel skryje a je pořízen snímek obrazovky pro možnou vizuální kontrolu výsledku testu uživatelem.

Testovaný nástroj	Všechny
Datová sada	Emise
Očekávaný výstup	Aktivní všechny vrstvy
Postup	1. Načtení datové sady „Emise“ ve formě JSON souboru. 2. Načtení připravené konfigurace ve formě JSON souboru, která má nastavené konfigurace pro všechny mapové vrstvy – kartogram, vrstvu se značkami a spojovací vrstvu. 3. Otevření postranního panelu. 4. Otevření karty s nastavením vrstvy kartogramu a aktivace vrstvy prostřednictvím zaškrtávacího tlačítka. 5. Otevření karty s nastavením vrstvy se značkami a aktivace vrstvy prostřednictvím zaškrtávacího tlačítka. 6. Otevření karty s nastavením spojovací vrstvy a aktivace vrstvy prostřednictvím zaškrtávacího tlačítka.

Kontrola barevného odstínu kartogramu při agregaci počtu

Tento test se zaměřuje na kontrolu nástroje vrstvy kartogramu. Proto je nutné pro tento test použít konfiguraci, která má nastavené datové domény pro vrstvu kartogramu. Taktéž je důležité mít pro tento test nastavenou agregaci počtu v konfiguraci vrstvy kartogramu. Jelikož klíčová vlastnost kartogramu je využití rozdílů barev či stínování v jednotlivých oblastech k reprezentaci hodnot, test se zaměřuje právě na kontrolu barev. Postupně se tak prochází jednotlivé oblasti nacházející se v mapě a kontroluje se, zdali oblasti mají správnou barvu či odstín dané barvy.

Testovaný nástroj	Choropleth layer
Datová sada	Populace
Očekávaný výstup	Správné zobrazení barevného odstínu
Postup	1. Načtení datové sady „Populace“ ve formě JSON souboru. 2. Načtení připravené konfigurace ve formě JSON souboru, která má nastavenou konfiguraci pro testování vrstvy kartogramu. 3. Kontrola správného odstínu v jednotlivých oblastech.

Kontrola barevného odstínu kartogramu při agregaci sumy

Test kontroly barevného odstínu kartogramu se stejně jako předchozí test zaměřuje na kontrolu nástroje vrstvy kartogramu. Hlavní rozdíl oproti předchozímu testu je ten, že tento test kontroluje hodnoty při použití agregaci sumy místo agregace počtu.

Testovaný nástroj	Choropleth layer
Datová sada	Populace
Očekávaný výstup	Správné zobrazení barevného odstínu
Postup	1. Načtení datové sady „Populace“ ve formě JSON souboru. 2. Načtení připravené konfigurace ve formě JSON souboru, která má nastavenou konfiguraci pro testování vrstvy kartogramu. 3. Kontrola správného odstínu v jednotlivých oblastech.

Kontrola správnosti údajů zobrazených v pop-upu značky a kontrola správného zobrazení barevných částí značky při agregaci počtu

Tento test se zaměřuje na vrstvu se značkami. Pro test je nutné použít konfiguraci, která má nastavené datové domény pro vrstvu se značkami a má nastavenou agregaci hodnot na „počet“. Test se zaměřuje na kontrolu hodnot ve vyskakovacích bublinách, které se zobrazí po kliknutí na příslušnou značku a kontrolu samotných značek v jednotlivých oblastech mapy. Ve vyskakovacích bublinách se kontrolují hodnoty pro jednotlivé kategorie a hodnota součtu. U jednotlivých značek se pak kontroluje hodnota v nich zobrazená a jestli jednotlivé barevné části značky odpovídají správným poměrům. Agregované hodnoty pro porovnávání se získávají výpočty v testu z datové sady, která je nastavena u dané konfigurace testu.

Testovaný nástroj	Marker layer
Datová sada	Emise
Očekávaný výstup	Správná hodnota v pop-upu značky
Postup	1. Načtení datové sady „Emise“ ve formě JSON souboru. 2. Načtení připravené konfigurace ve formě JSON souboru, která má nastavenou konfiguraci pro testování vrstvy se značkami. 3. Zobrazení detailu – pop-upu v dané zemi pomocí kliknutí. 4. Kontrola správných hodnot uvedených v pop-upu. 5. Kontrola správné hodnoty zobrazené uvnitř značky. 6. Kontrola správného poměru barevných částí jednotlivých značek.

Kontrola správnosti údajů zobrazených v pop-upu značky a kontrola správného zobrazení barevných částí značky při agregaci sumy

Tento test se stejně jako předchozí zaměřuje na nástroj vrstvy se značkami. Postup testu je shodný s předchozím, jediným rozdílem je, že tento test kontroluje hodnoty při použití agregaci sumy místo agregace počtu.

Testovaný nástroj	Marker layer
Datová sada	Emise
Očekávaný výstup	Správná hodnota v pop-upu značky
Postup	1. Načtení datové sady „Emise“ ve formě JSON souboru. 2. Načtení připravené konfigurace ve formě JSON souboru, která má nastavenou konfiguraci pro testování vrstvy se značkami. 3. Zobrazení detailu – pop-upu v dané zemi pomocí kliknutí. 4. Kontrola správné hodnoty uvedené v pop-upu. 5. Kontrola správné hodnoty zobrazené uvnitř značky. 6. Kontrola správného poměru barevných částí jednotlivých značek.

Ověření existence spojení ve spojovací vrstvě

Tento test se soustředí na kontrolu nástroje spojovací vrstvy. Pro test je nutné použít konfiguraci, která má nastavené datové domény pro spojovací vrstvu. Samotný test následně kontroluje, zdali pro všechny relace z datové sady dle konfigurace existují prvky (křivky) vyjadřující spojení mezi jednotlivými oblastmi v mapě.

Testovaný nástroj	Connection layer
Datová sada	Obchod služeb
Očekávaný výstup	Existence hrany mezi zeměmi
Postup	1. Načtení datové sady „Obchod služeb“ ve formě JSON souboru. 2. Načtení připravené konfigurace ve formě JSON souboru, která má nastavenou konfiguraci pro testování spojovací vrstvy. 3. Kontrola existence hran mezi jednotlivými oblastmi.

6.3 Profilování

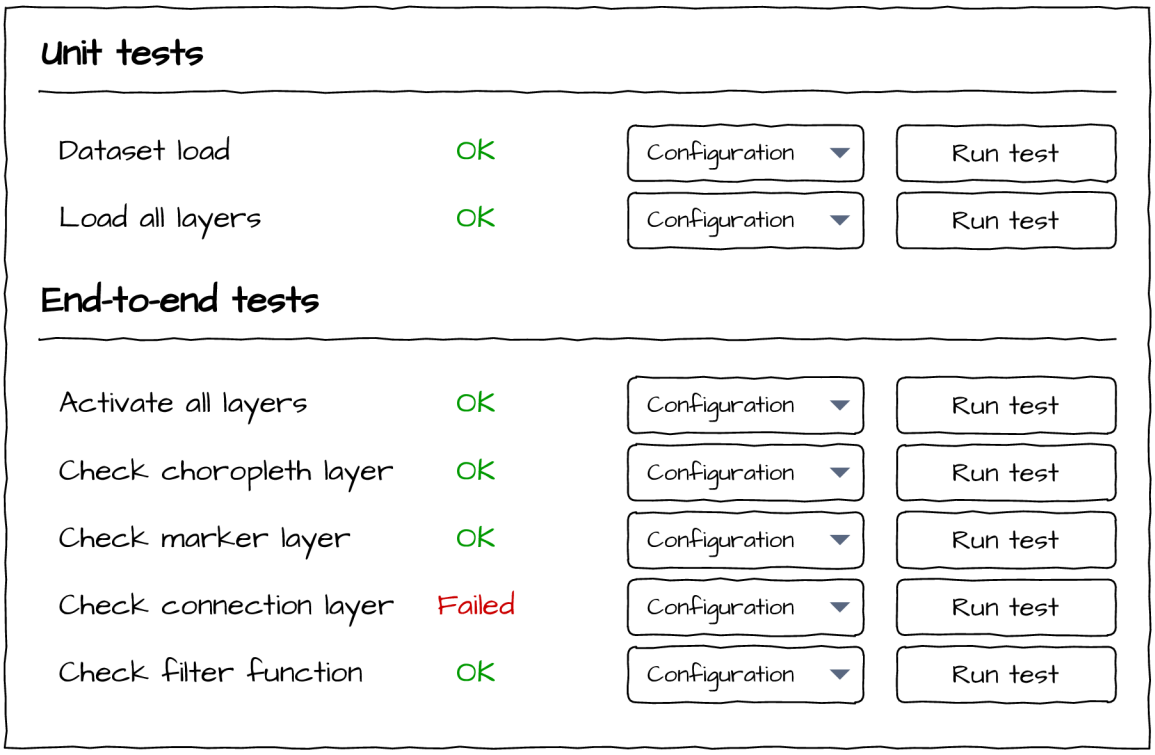
Scénáře pro profilování budou spouštěny s datovou sadou „Obchod“, která obsahuje přes 335 tisíc záznamů (každý záznam se skládá z 11 hodnot) tak, aby se ověřila výkonnost knihovny při práci s rozsáhlejšími daty. Tato datová sada bude rozdělena do několika menších částí, aby se mohl ověřovat vliv velikosti datové sady na čas vykonání jednotlivých funkcí. Pomocí profilování se budu snažit odhalit výkonnostní problémy knihovny v jejím jádru i v jejich nástrojích. Pro nalezené výkonnostní problémy budou vytvořeny návrhy pro optimalizaci výkonu či přímo provedeny optimalizace zdrojového kódu knihovny pro odstranění těchto výkonnostních problémů.

První scénář pro profilování bude spuštěn před načtením datové sady a zastaven po kompletním načtení. Další scénáře pro profilování budou spuštěny vždy před aktivací vrstvy příslušného nástroje, následně budou provedeny akce nad danou vrstvou (posun mapy, zobrazení pop-upu apod.) a pak bude profilování zastaveno. Scénář pro profilování bude vhodně uložen společně s referenčními hodnotami tak, aby se mohl kdykoliv při budoucích změnách opakovaně spouštět.

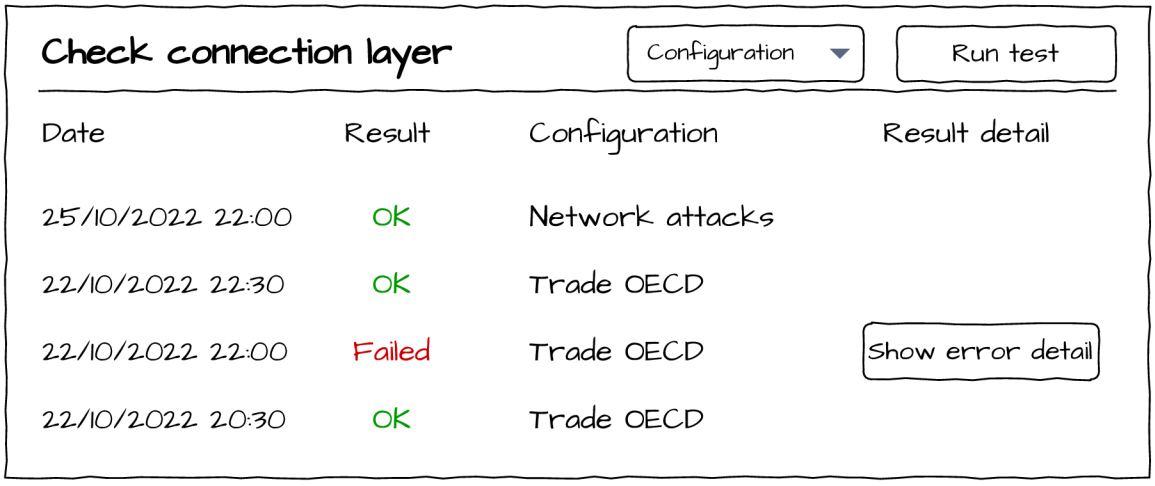
6.4 Uživatelské rozhraní

Na obrázku 6.2 je znázorněn návrh obrazovky s přehledem dostupných testů, které lze spustit. Obrazovka bude rozdělena na dvě části, a to část pro jednotkové testy a část pro end-to-end testy. Pro jednotlivé testy v těchto částech bude zobrazen název testu, výsledek posledního vykonání tohoto testu, možnost nastavení konfigurace pro daný test a tlačítko pro spuštění testu. Výčet konfigurací, které lze zvolit pro jednotlivé testy, bude shodný se seznamem konfigurací uložených v informačním systému pro správu datových sad a nad nimi vytvořených konfigurací.

Na obrázku 6.3 je zobrazen návrh obrazovky s detailem testu. Do této obrazovky uživatel přejde z obrazovky s přehledem kliknutím na příslušný test. V detailu daného testu je možné upravit nastavení jeho konfigurace či jej spustit. Dále je zde zobrazena tabulka s historií spouštění daného testu a jeho výsledky. U každého záznamu je zobrazen datum a čas spuštění testu a výsledek testu, který může nabývat tří hodnot – úspěšný, neúspěšný a právě probíhající test. Dále je zde zobrazena konfigurace, se kterou byl test spouštěn a v případě neúspěšného testu se zobrazí tlačítko pro zobrazení detailu chyby, proč daný test selhal. Obrazovka s detailem testu je shodná jak pro jednotkové testy, tak pro end-to-end testy.



Obrázek 6.2: Návrh obrazovky s přehledem dostupných testů

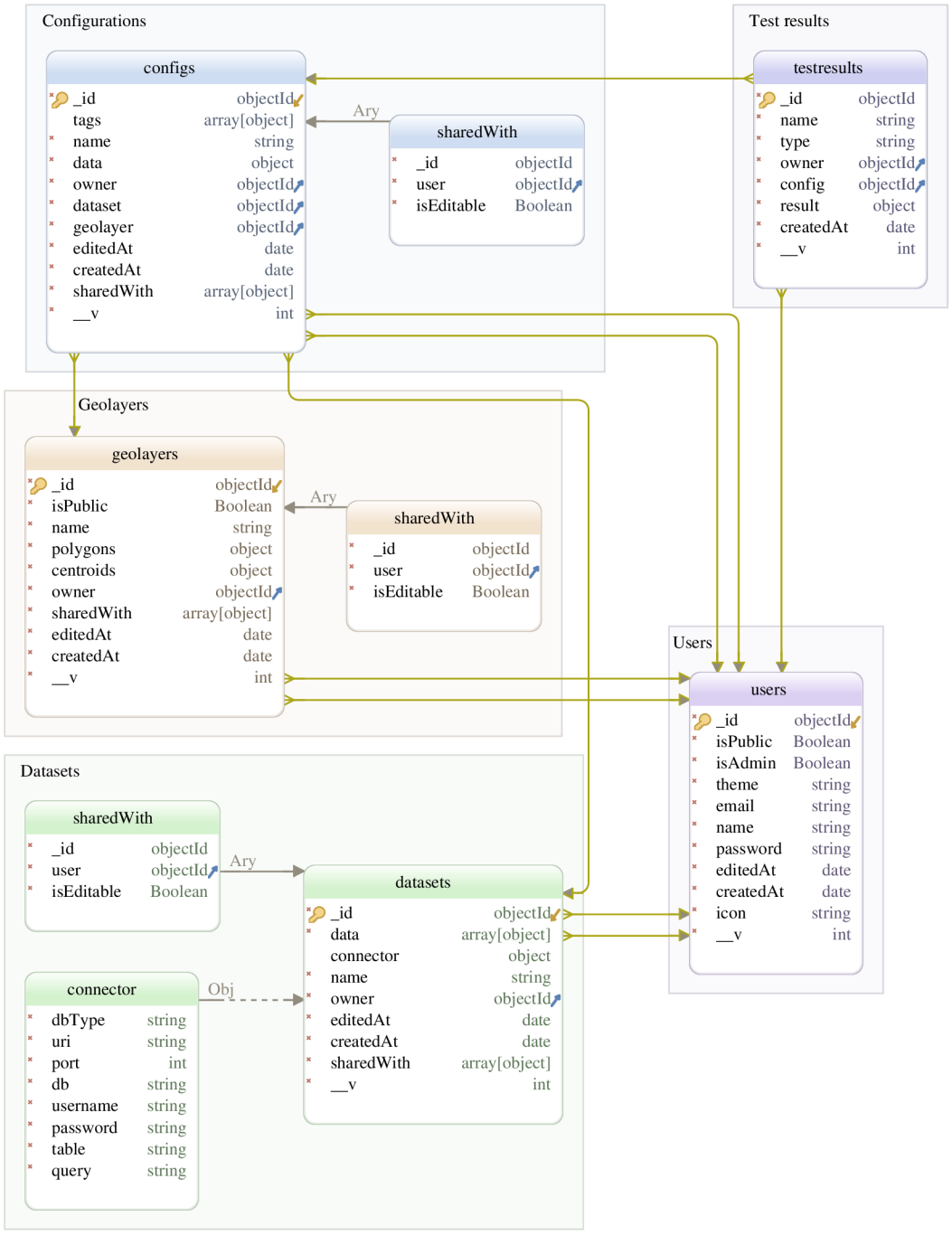


Obrázek 6.3: Návrh obrazovky s detailem vybraného testu

6.5 Datový model

Rozhraní pro automatizované testování knihovny bude využívat stejnou databázi, jako využívá samotný informační systém, do kterého bude toto rozhraní implementováno. V databázi informačního systému se vyskytují 4 typy entit – konfigurace, geografické objekty, datové sady a uživatelé. Tato databáze bude rozšířena o entitu `testresults` za účelem ukládání výsledků běhů testů spuštěných pomocí rozhraní implementovaného v informačním sys-

tému. Entita `testresults` bude obsahovat název testu a jeho typ, ID uživatele, který daný test spustil, dále ID konfigurace, se kterou byl test spuštěn a datum a čas, kdy byl test spuštěn. Následně bude entita obsahovat objekt `result`, který bude obsahovat samotné výsledky daného testu. Struktura tohoto objektu záleží na typu testu a také na tom, zdali daný běh testu proběhl úspěšně, či neúspěšně (v takovém případě obsahuje více polí s informacemi o chybách, které nastaly). Schéma entit, které bude informační systém využívat, je znázorněno na obrázku 6.4.



Obrázek 6.4: Navržené schéma databáze znázorňující entity v informačním systému

Kapitola 7

Implementace

V této kapitole se zaměřím na samotnou implementaci rozhraní pro automatizované testování knihovny Geovisto. Zdůvodním volbu použitých technologií a popíši implementační detaily rozhraní pro automatizované testování. V této kapitole je taktéž zmíněna volba technologií použitých pro profilování knihovny a způsob jejich použití.

7.1 Architektura

Informační systém pro správu datových sad a jejich konfigurací byl vyvinut v rámci diplomové práce pomocí technologií JavaScript a MongoDB. Přesněji, pro backend část je použit framework Express[1], což je webový framework pro Node.js[2]. Tento framework patří mezi nejpopulárnější frameworky používané na backend části. Umožňuje rychlejší a přehlednější vývoj, než pokud bychom pracovali přímo s Node.js, kde bychom museli používat přímo `http` modul a měli tak mnoho práce například s parsováním požadavků či směrováním pomocí regulárních výrazů [39]. Express je nedogmatický, což znamená, že se nesnaží nabízet nějaké doporučené komponenty či „nejlepší postup“, ale nechává volné ruce vývojáři, aby si našel a použil komponenty a postupy, které mu vyhovují [8].

Informační systém využívá jako databázi pro ukládání dat MongoDB[3]. MongoDB je dokumentově orientovaná NoSQL databáze, která nevyžaduje pevně dané schéma, ale využívá dynamické databázové schéma. To znamená, že místo tabulek se data ukládají v podobě dokumentů, které mají podobnou strukturu jako JSON objekty. Jednotlivé hodnoty polí mohou obsahovat nejen jednoduché hodnoty, ale mohou obsahovat také dokumenty, pole hodnot či pole dokumentů. Dokumenty se ukládají do takzvaných kolekcí, což je obdoba tabulek v relačních databázích. Entity jsou reprezentovány pomocí objektů validovaných knihovnou Mongoose s definovaným datovým schématem. Mongoose zde rovněž zajišťuje mapování objektů na dokumenty a zpět, jejich vytváření a naplnění dat z databáze. Nové datové sady do informačního systému lze přidat dvěma způsoby – s využitím JSON či propojením s databází. Pro toto využití informační systém podporuje tři databázové systémy – MongoDB, MySQL a PostgreSQL. Propojení s databázovými systémy je zajištěno pomocí konektorů, takže není problém v případě potřeby přidat podporu některých dalších databázových systémů.

[1] https://expressjs.com
[2] https://nodejs.org/en
[3] https://www.mongodb.com/

Pro frontend část byl využit framework React[4]. Tento framework se dnes společně s frameworky Angular[5] a Vue.js[6] těší velké popularitě mezi vývojáři. React je založen na takzvaných komponentách, což jsou znovupoužitelné HTML elementy se zapouzdřenou funkcionalitou. Postupným skládáním těchto komponent následně vzniká samotné uživatelské rozhraní aplikace. Samotné komponenty mají své vlastnosti a vnitřní stav. Častým využitím frameworku React je tvorba tzv. single-page aplikací (SPA).

Informační systém umožňuje nejen správu datových sad a správu konfigurací, ale také umožňuje kooperaci uživatelů při tvorbě dat. Kooperace je zajištěna díky možnosti sdílení datových sad a jejich konfigurací mezi zaregistrovanými uživateli v informačním systému. V informačním systému se nachází dvě role, kterých může uživatel nabývat. Uživatel může mít buď roli normálního uživatele nebo roli administrátora. Administrátorská role umožňuje použití stejných funkcí, které má dostupné normální uživatel, ale navíc má po přihlášení přístup do sekce „Systém“, kde má přístup k logům pro případné zjištění zdrojů problémů, které v systému nastaly.

7.2 Práce s testy

Jak již bylo zmíněno v návrhu, rozhodl jsem se, že pro testování knihovny Geovisto budou implementovány jednotkové a end-to-end typy testů. V této sekci se nachází popis implementace jednotkových a end-to-end testů společně s popisem implementace rozhraní pro automatizované testování, které je přidáno jako nová část informačního systému pro správu vizualizací geografických dat[7].

7.2.1 Jednotkové testy

Pro spouštění jednotkových testů jsem zvolil framework Jest. Tento framework byl zvolen především díky jednoduchosti jeho používání a jeho dobré rychlosti při vykonávání jednotlivých testů. Vzhledem k implementaci rozhraní, které následně spouští testy na backend části, bylo taktéž při výběru důležité vybrat framework s dobrou podporou možnosti spouštění testů ze zdrojového kódu, což bylo díky průzkumu zjištěno, že je u některých testovacích frameworků problém. Tuto vlastnost framework Jest splňuje a proto byl zvolen pro práci s jednotkovými testy.

Jednotlivé testy byly implementovány pomocí jazyka TypeScript. Pro zjednodušení práce s knihovnou Geovisto při vytváření jednotlivých jednotkových testů byla vytvořena třída `GeovistoInstanceBuilder`, která zajišťuje vytváření instancí knihovny Geovisto. Tato třída je založena na návrhovém vzoru stavitel (*Builder*). Pomocí jednotlivých metod umožňuje nastavit geografická data, datovou sadu či konfiguraci, se kterou bude vytvářená instance pracovat. Dále je zde možnost si dle potřeby přidat nástroje, které bude vytvářená instance obsahovat. Tato možnost byla implementována kvůli tomu, že pokud bude vytvářený jednotkový test pracovat pouze s nějakým nástrojem, například spojovací vrstvou, tak si můžeme zvolit, aby vytvářená instance obsahovala pouze tento nástroj a nemusela zbytečně obsahovat všechny nástroje, se kterými nebude test nijak pracovat. Po nastavení konfigurace a zvolení příslušných nástrojů třída pomocí metody `createInstance` navrátí nakonfigurovanou instanci knihovny Geovisto.

[4] https://reactjs.org/
[5] https://angular.io/
[6] https://vuejs.org/
[7] https://www.vut.cz/www_base/zav_prace_soubor_verejne.php?file_id=231051

Aby se zabránilo spuštění všech testů, byl využit parametr `testNamePattern`, pomocí kterého Jest spouští pouze testy, které jsou uvedeny v regulárním výrazu. Název testu je dán parametrem `title` ve funkci `it`, která obsahuje zdrojový kód testu. Při provádění jednotkových testů se nejprve vytvoří instance knihovny Geovisto, které se nastaví požadovaná konfigurace – například pomocí zmíněné třídy `GeovistoInstanceBuilder`. Následně se na instanci použijí dané metody dle toho, jaká funkcionalita má být v daném testu otestována. Nakonec se získá stav instance knihovny Geovisto, ze kterého se získají data nutná k porovnání hodnot tak, aby se zjistilo, zda daná funkcionalita funguje správně. Pro příklad při testování správné funkčnosti načítání dat z JSON souboru s datovou sadou se získají z instance objekty představující tato data již zpracovaná. Hodnoty získané z těchto objektů se následně porovnávají s hodnotami z datové sady, která byla do knihovny načtena.

7.2.2 End-to-end testy

Pro spouštění end-to-end testů byl vybrán framework Cypress. Tento framework byl zvolen z důvodu jeho odlišné architektury proti ostatním end-to-end testovacím knihovnám, která umožňuje jednodušší způsob jeho používání, rychlejší a efektivnější psaní testů a rozsáhlé možnosti ladění. Tento framework rovněž umožňuje nahrávat průběh testů či pořizovat snímky obrazovky. Cypress umožňuje definovat globálně akce, které se mají provést před či po každém průběhu testů. Tato vlastnost byla při implementaci využita pro to, aby se před spuštěním každého testu připravila datová sada a konfigurace do definované proměnné a aby se po dokončení každého testu pořídil snímek obrazovky.

Nahrané videa s průběhy testů a snímky obrazovky je možné si v rozhraní po dokončení testů stáhnout a prohlédnout. Stejně jako u jednotkových testů je u end-to-end testů dán jejich název parametrem `title` ve funkci `it`, která obsahuje zdrojový kód testu. Při implementaci end-to-end testů byly vytvořeny pomocné funkce v takzvaných `util` souborech, které umožňují kontrolu dat, které se mají zobrazit v uživatelském rozhraní aplikace. Mezi tyto funkce patří například funkce pro získání hodnot, které se mají zobrazit v pop-upech značek či hodnoty barevných intenzit, kterých mají nabývat jednotlivé oblasti v kartogramu.

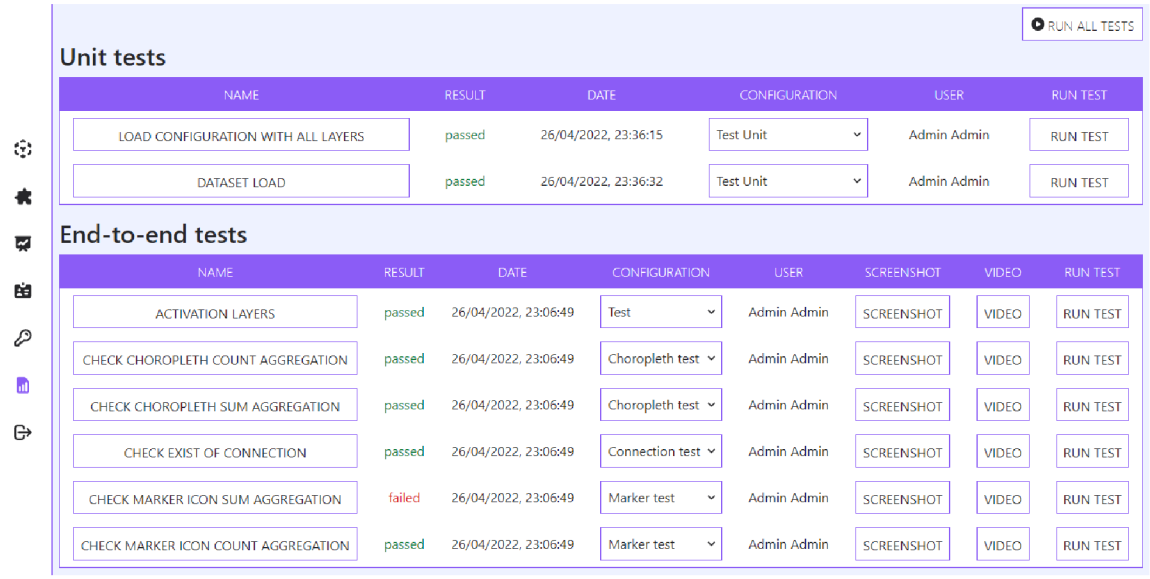
Pro end-to-end testy byla vytvořena v informačním systému obrazovka, která umožňuje změnu datové sady, konfigurace či geografických objektů pomocí nahrání souborů. V případě spuštění end-to-end testu se tak přejde na tuto obrazovku, kde se postupně nahraje datová sada a geografické objekty, které jsou uvedené ve vybrané konfiguraci, v podobě JSON souborů. Následně se nahraje konfigurace a poté se spustí samotné tělo testu.

V end-to-end testech se získávají jednotlivé elementy, obsahující hodnoty či vlastnosti, které chceme testovat, pomocí takzvaných selektorů. Tyto selektory fungují shodně jako je tomu u technologie CSS (*Cascading Style Sheets*). Můžeme tak elementy získat pomocí názvu jejich HTML identifikátoru či třídy, ale i pomocí například data atributů. Pro některé typy testů bylo nutné upravit jádro knihovny tak, aby některé elementy obsahovaly navíc vlastnosti, které pomohly při porovnání správnosti hodnot. Toto bylo například nutno provést u nástroje vrstvy se značkami, kde se jednotlivým značkám přidávaly vlastnosti, které obsahují identifikátory oblastí, ve kterých se značky nacházejí.

7.2.3 Implementace rozhraní

Sekce pro testování byla přidána do části informačního systému, která je dostupná pouze uživatelům s právy administrátora. Všichni administrátoři zde vidí průběhy testů, které byly spuštěny i ostatními administrátory. Výsledek implementace rozhraní pro testování do informačního systému můžete vidět na obrázku 7.1. Rozhraní umožňuje vidět poslední výsledky

jednotlivých testů – jestli byl test úspěšný, kdy byl test spuštěn a kým byl test spuštěn. Dále je zde možnost změnit konfiguraci jednotlivých testů či jednotlivé testy spouštět. Do výběrového pole s možnými konfiguracemi u jednotlivých testů se načítají konfigurace, které má přihlášený uživatel vytvořené v sekci informačního systému pro konfigurace. U end-to-end testů je navíc možnost stáhnout si snímek obrazovky, který byl pořízen po dokončení testu nebo si stáhnout video s průběhem testu.

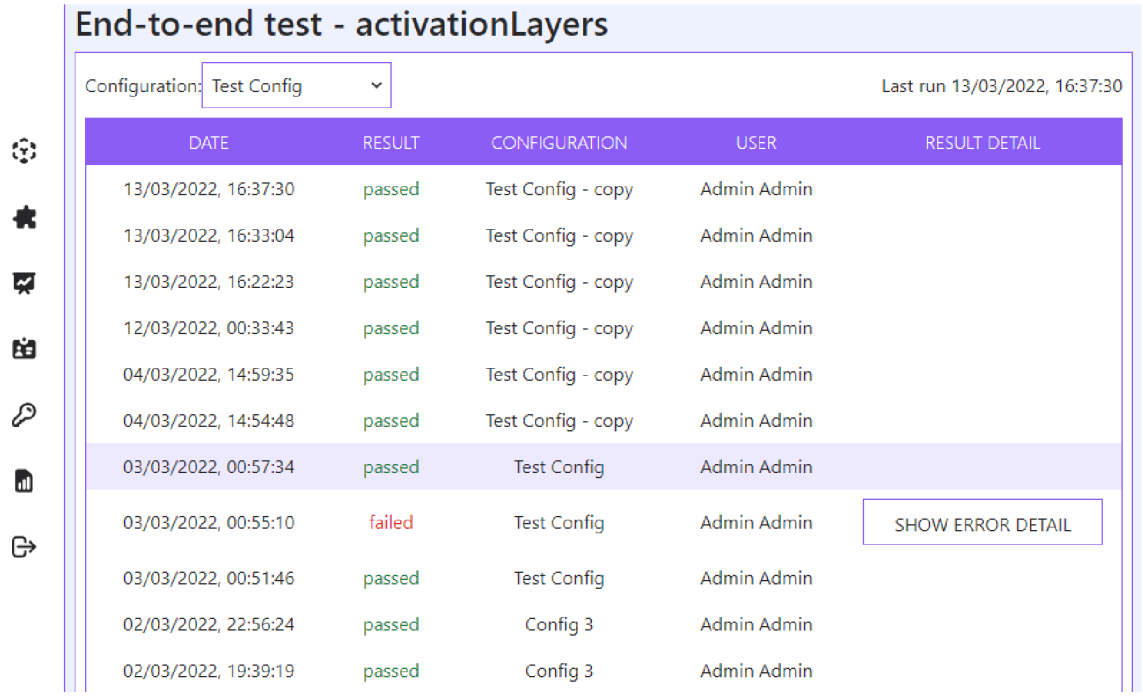


Obrázek 7.1: Rozhraní pro testování – obrazovka s přehledem testů

Při zobrazení detailu testu – kliknutím na jeho název, se zobrazí historie spuštění daného testu (viz. obrázek 7.2). Zde se zobrazí přehled spuštění daného testu společně s datem, výsledkem testu, jeho konfigurací při spuštění a uživatelem, který jej spouštěl. Pokud test skončil neúspěšně, zobrazí se tlačítko, které umožňuje zobrazit detail, proč test selhal. Na této obrazovce stejně jako na obrazovce s přehledem všech testů, lze nastavit konfiguraci pro příslušný test či daný test spustit.

Při tvorbě nových testů stačí vytvořit soubor s testy ve vyhrazených složkách pro jednotkové a end-to-end testy. Informační systém si následně automaticky příslušné složky prohledá a zobrazí v rozhraní všechny dostupné testy v těchto složkách. Není tak třeba, aby uživatel testy nijak ručně přidával do rozhraní pro testování knihovny, informační systém to zařídí sám. Výchozí konfigurace společně s datovými sadami vhodnými pro jednotlivé testy, které jsou v rámci práce implementovány, jsou vytvořeny při inicializaci databáze informačního systému.

Při spuštění testu se nejprve vytvoří soubory s datovou sadou, geografickými objekty a konfigurací, které byly zvoleny pomocí rozhraní. Pokud datová sada či konfigurace není vložena pomocí JSON notace, ale nachází se v externí databázi, stáhne se z databáze a vytvoří se z nich příslušné soubory. Soubory pro jednotlivé testy jsou uloženy ve složkách `fixture` zvlášť pro jednotkové testy a pro end-to-end testy a jsou pojmenovány dle názvu testu, ke kterému přísluší. Následně se vytvoří záznamy s výsledky testů v kolekci `testresults` v MongoDB databázi. Tyto záznamy obsahují všechny informace kromě výsledku, který je doplněn po doběhnutí daného testu. Tyto záznamy jsou vytvářeny, aby se mohlo v rozhraní zobrazit, že daný test je právě spuštěn – probíhá. Nakonec jsou samotné



Obrázek 7.2: Rozhraní pro testování – obrazovka s historií testu

testy spuštěny pomocí příslušného testovacího frameworku dle typu testu. Po doběhnutí testu je v případě potřeby upravena struktura dat s výsledky a výsledek je doplněn k příslušnému záznamu v MongoDB databázi. Celý proces je zobrazen na obrázku 7.3.

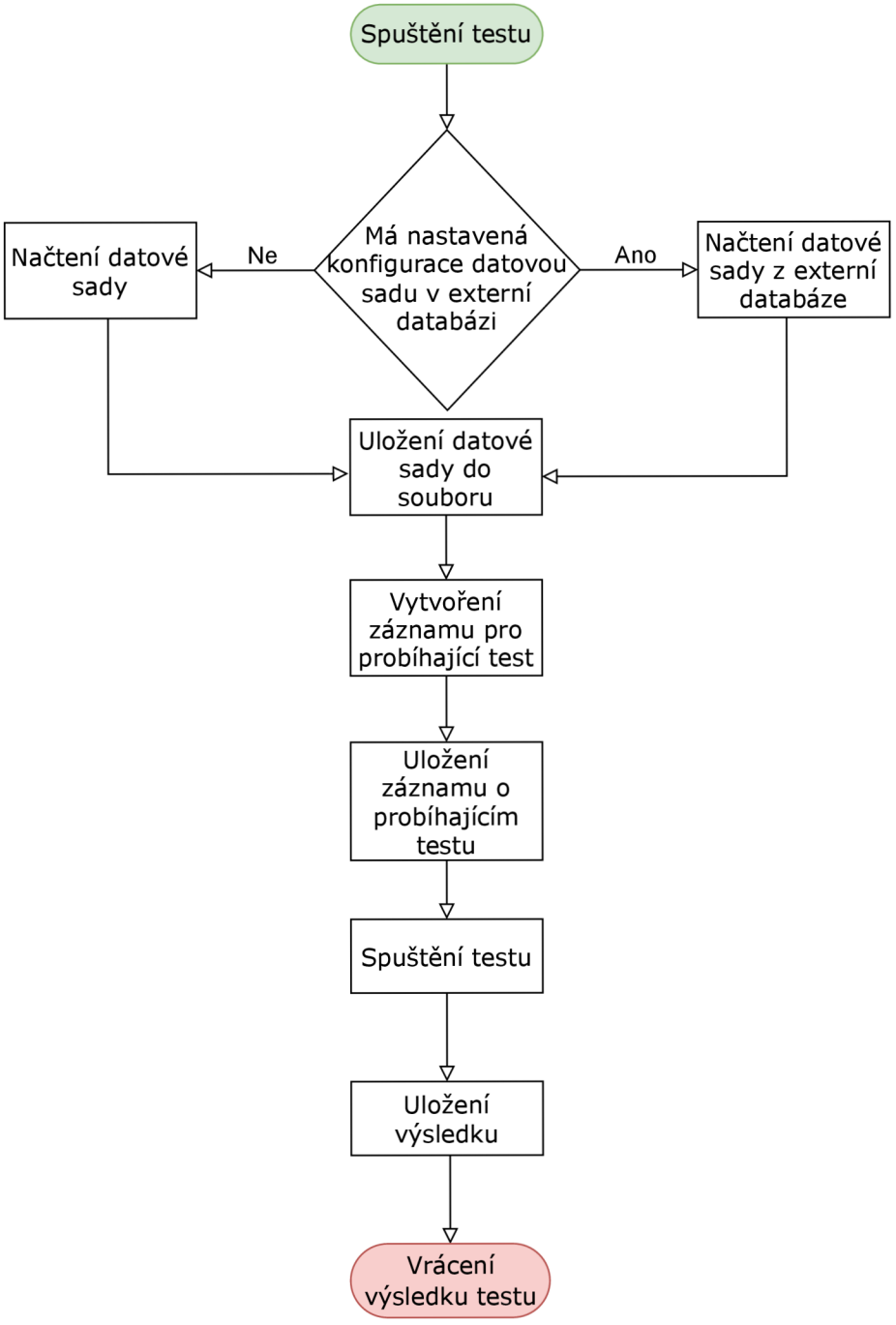
7.3 Profilování

Pro část profilování byl vybrán nástroj Chrome DevTools. Tento nástroj je součástí prohlížeče Google Chrome. Chrome DevTools je velmi komplexní nástroj, který umožňuje mnoho funkcí jako například zobrazení jednotlivých prvků a jejich vlastností na webové stránce, zobrazení zdrojů stránky a využití sítě či sledování využití zdrojů počítače webovou aplikací a mnoho dalšího. Navíc lze další funkcionality přidat pomocí Google Chrome doplňků. Právě již zmíněná funkcionalita sledování využití zdrojů počítače je využita v práci k profilování.

Tato funkce se nachází v nástroji Chrome DevTools pod záložkou „Výkon“. Tato záložka se dále rozděluje na několik sekcí, které jsou vysvětleny v sekci 4.5. V rámci práce byly při profilování využity nejvíce části „Graf vytížení CPU“ a „Hlavní“, která se dále dělí na několik sekcí. Z těchto sekcí byly nejvíce využity sekce „Zdola nahoru“ a „Strom volání“, které umožňují sledování času potřebného k vykonání jednotlivých funkcí. Pro samotnou optimalizaci jednotlivých funkcí po zjištění výkonnostních problémů díky profilování, byla využita aplikace JSBench.Me[8], která umožňuje porovnání výkonu dvou vložených bloků kódu v JavaScriptu. Tato aplikace funguje tak, že dané bloky kódu opakovaně několikrát spustí (aby se zabránilo případným odchylkám měření způsobeným různým zatížením počítače) a následně zobrazí, kolikrát se zvládne daný blok kódu vykonat za sekundu. Dané bloky kódu mezi sebou navíc porovná a zobrazí procentuální rozdíl výkonnosti mezi nimi.

[8] https://jsbench.me/

Profilování bylo provedeno komplexně i zvlášť pro sledování výkonnosti jednotlivých nástrojů knihovny Geovisto tak, aby bylo nalezeno pokud možno co nejvíce výkonnostních problémů aplikace. Pro profilování byla použita již zmíněná datová sada se záznamy obchodování se surovinami mezi jednotlivými zeměmi, která byla velmi rozsáhlá. Tato datová sada byla dále rozdělena na několik menších sad pro možnost sledování výkonnosti knihovny nad různě velkými datovými sadami. Toto rozdělení umožnilo lepší sledování výsledků případných optimalizací knihovny.



Obrázek 7.3: Proces spuštění testu

Kapitola 8

Testování

V této kapitole je popsán průběh a výsledky testování rozhraní pro automatizované testování knihovny Geovisto. Dále je v této kapitole popsán proces profilování a uvedeny optimalizace, které byly na základě výsledků profilování provedeny.

8.1 Testování funkcionality

Pro testování funkcionality jsem vytvořil konfigurace, které vycházely z již zmíněných datových sad a přiřadil je k jednotlivým testům, které byly navrženy v sekci 6.2. Polovina datových sad pro testování byla vložena pomocí JSON notace a druhá polovina se nacházela v databázi PostgreSQL tak, aby se ověřila funkčnost načítání datových sad z externích databází.

Následně byly spouštěny jednotlivé testy, které byly navrženy a poté implementovány a ověřovala se jejich funkčnost. Účelně byla změněna implementace v některých nástrojích tak, aby vykazovaly nekorektní data a ověřilo se, zda tyto chyby testy detekují. Tyto scénáře proběhly úspěšně a rozhraní v těchto případech poskytlo smysluplnou vazbu, pomocí které šlo lehce zjistit zdroj problému. U end-to-end testů je navíc k dispozici video, na kterém lze pozorovat celý průběh testu a poskytne tak ještě lepší zpětnou vazbu pro odhalení problému, který v daném testu nastal.

Při spouštění end-to-end testů bylo zpozorováno značně pomalejší chování při spuštění jednotlivých testů. Při podrobnějším průzkumu tohoto chování bylo zjištěno, že testovací framework Cypress při zadání názvu jednoho testu, který má vykonat, prochází všechny soubory s testy a kontrola shod názvu jednotlivých testů mu trvá značně dlouhou dobu. Tento problém byl následně odstraněn vytvořením funkce, která vyhledala soubory, ve kterých se nacházejí požadované testy, které se mají vykonat a názvy těchto souborů následně byly předány frameworku Cypress pomocí parametru při spouštění.

8.2 Uživatelské testování

Po dokončení části testování funkcionality, jsem požádal vývojáře knihovny Geovisto o otestování rozhraní pro automatizované testování knihovny. Během tohoto testování byly nalezeny dva hlavní problémy. První z těchto problémů se netýkal samotného rozhraní, ale celého informačního systému, ve kterém je rozhraní zabudováno. Problém spočíval v problematickém prvním spuštění tohoto informačního systému a jeho konfiguraci, která nebyla

dostatečně zdokumentována. Problém byl odstraněn pomocí doplnění dokumentace pro konfiguraci a odstranění chyb, které nastávaly při prvním spuštění informačního systému.

Druhý problém, který byl nalezen, byla chybějící možnost práce s vlastními geografickými objekty v testech. Tato možnost byla poté implementována do rozhraní pro testování. Mezi další problémy, které byly v rámci uživatelského testování odhaleny, patřilo rozdílné chování vývojových prostředí při implementaci testů. Zatímco v software JetBrains PhpStorm, ve kterém byla práce implementována nenastával žádný problém, ve vývojovém prostředí Visual Studio Code, které je využíváno vývojáři knihovny, nastávaly problémy v nenalezení funkcí použitých v jednotkových testech. Tato chyba byla poté odstraněna pomocí úprav konfigurace TypeScriptu.

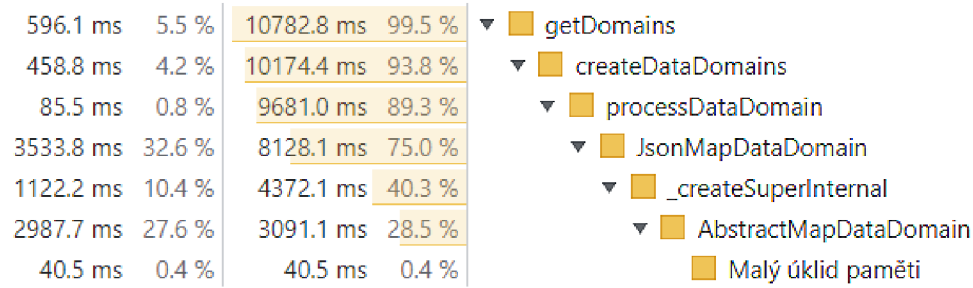
Dalším z větších problémů byl problém, kvůli kterému nebylo možno používat vlastní geografické objekty v informačním systému. To bylo způsobeno tím, že jádro knihovny Geovisto začalo používat odlišnou strukturu zadávání geografických objektů. Po odhalení tohoto problému byl informační systém upraven tak, aby byl s novou strukturou geografických objektů kompatibilní. Další odhalené problémy již byly drobnější povahy a jejich odstranění nezabralo mnoho času. Mezi tyto problémy patřil například nedostatečný popis chyb, které nastaly v průběhu jednotlivých testů či nutnost ručního vytvoření složek pro datové sady, geografické objekty a konfigurace.

8.3 Výsledky profilování

Jak již bylo zmíněno, profilování bylo provedeno pomocí nástroje Chrome DevTools. Scénářů pro profilování bylo vytvořeno několik tak, aby scénáře byly zaměřeny na jednotlivé nástroje, které knihovna Geovisto obsahuje. Po provedení profilování byly zjištěny dvě části, jejichž zpracování trvalo dlouho a existoval u nich prostor pro možné optimalizace. Pro měření výsledků zrychlení po optimalizaci bylo provedeno 20 měření a výsledné zrychlení bylo z jednotlivých časů zprůměrováno.

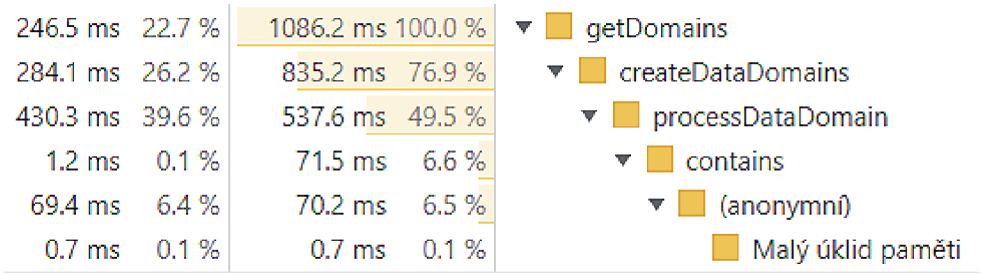
8.3.1 Načtení datové sady

První výkonnostní problém zjištěný při profilování bylo načtení datové sady. Pokud byla datová sada rozsáhlá – například datová sada se záznamy obchodování mezi jednotlivými zeměmi, která byla použita k profilování či letové záznamy mezi letišti, trvalo načtení datové sady i několik sekund. Pro příklad, načtení datové sady obsahující 100 000 záznamů, kde každý záznam obsahuje 11 datových domén, trvalo před optimalizací přibližně 10,8 sekundy (viz. obrázek 8.1). V případě 300 000 záznamů to bylo již 36 sekund, což je již velká prodleva a při práci uživatelů s knihovnou je tato prodleva nežádoucí.

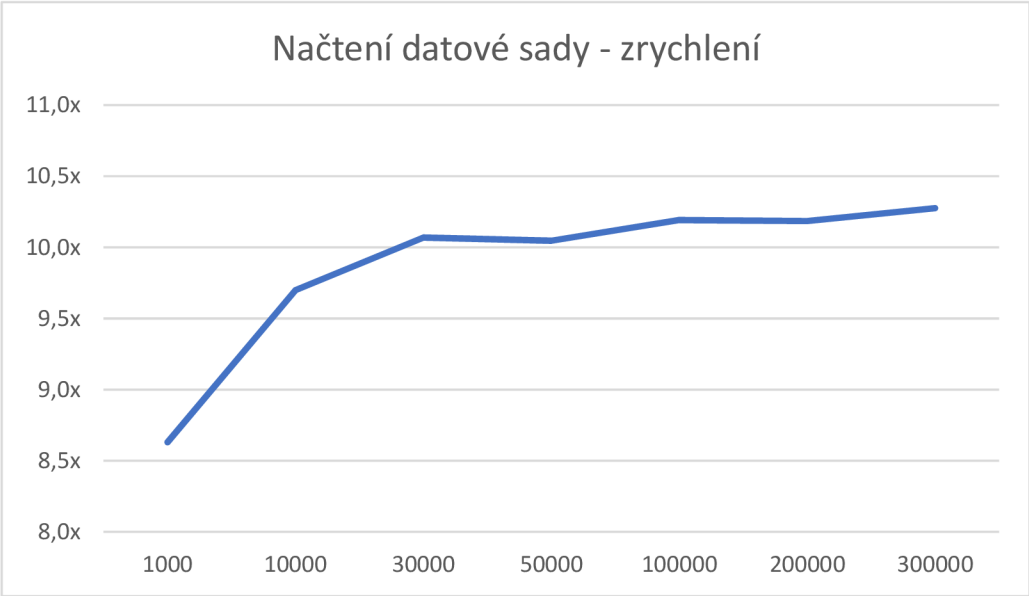


Obrázek 8.1: Výsledek profilování funkce pro načtení datové sady před optimalizací

Při optimalizaci tohoto výkonnostního problému byla upravena funkce, která zajišťovala zpracování datových domén z JSON notace a vytváření tříd označující příslušné datové domény. Optimalizace spočívala v upravení funkce pro hledání již existujících datových domén a upravení způsobu vytváření tříd označující datové domény. Tyto optimalizace zajistily přibližně 10-násobné zrychlení při načtení datových sad (viz. obrázek 8.2). Zrychlení můžeme vidět na obrázku 8.3, kde na vodorovné ose grafu je znázorněn počet objektů v datové sadě a na svislé ose je znázorněno průměrné zrychlení.



Obrázek 8.2: Výsledek profilování funkce pro načtení datové sady po optimalizaci

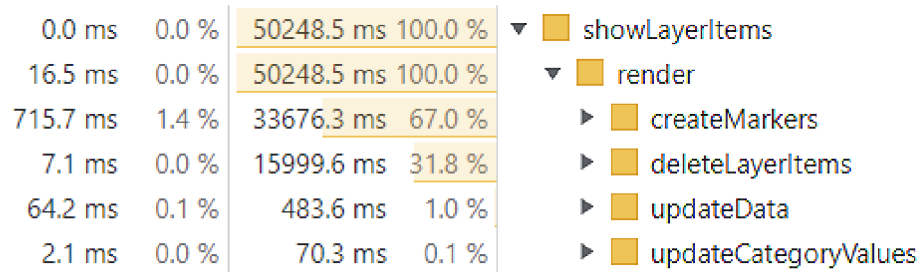


Obrázek 8.3: Graf zrychlení načtení datové sady po optimalizaci

8.3.2 Vrstva se značkami

Druhý výkonnostní problém při profilování byl zjištěn v nástroji pro tvorbu vrstvy se značkami. U tohoto problému nezávisí ani tak na velikosti datové sady, jako na počtu geografických objektů, se kterými knihovna pracuje. Při práci se značkami pro jednotlivé země světa či jednotlivé části země tento výkonnostní problém není pro uživatele tak znatelný, ale pokud by uživatel potřeboval například pracovat s geografickými objekty, které by označovaly senzory pro sledování hustoty dopravy, kterých může být velké množství, je prodleva při práci s tímto nástrojem značným problémem. Pro představu délka vykreslení vrstvy

se značkami pro každou zemi trvalo zhruba 300 milisekund před optimalizací. Při použití simulované sady s geografickými objekty představující senzory, kterých bylo 770, což například u zmíněného příkladu je reálný scénář, vykreslení vrstvy se značkami trvalo 50,2 sekund (viz obrázek 8.4).



Obrázek 8.4: Výsledek profilování funkce pro zobrazení vrstvy se značkami při použití geografické sady o 770 značkách před optimalizací

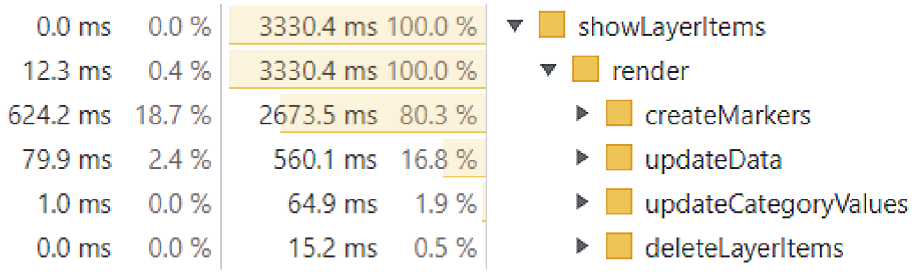
První problém u tohoto nástroje byl díky profilování zjištěn u funkce, která přidává jednotlivé značky do vrstev. Jednotlivé značky byly v cyklu vytvářeny a následně se provedlo zavolání funkce, která zajistila přidání značky do vrstvy. Problém nastával v tom, že při každém zavolání této funkce pro přidání značky do vrstvy se provedlo překreslení vrstev, což způsobovalo značné problémy ve výkonnosti. Řešením tohoto problému bylo vytvořit si nejprve pole, do kterého se jednotlivé značky vytvořené v cyklu přidaly. Jakmile toto pole obsahovalo všechny vytvořené značky, použila se funkce knihovny, která přidala všechny značky z pole najednou. Překreslení tak proběhlo pouze jednou a tím se běh funkce pro přidání značek značně zkrátil. Optimalizace tohoto problému přinesla různě velké zrychlení v závislosti na velikosti sady s geografickými objekty. Na obrázku 8.5 je zobrazen graf zrychlení po optimalizaci, kde na vodorovné ose grafu jsou znázorněny násobky počtu geografických objektů, které byly použity a na svislé ose je znázorněno průměrné zrychlení.

Obrázek 8.5: Graf zrychlení přidání značek do vrstev po optimalizaci

Druhý problém byl zjištěn ve funkci, která zajišťuje smazání všech značek z vrstev. V této funkci byly nejprve získány všechny značky a následně byly tyto značky jednotlivě odstraňovány. Problémem bylo, že odstranění těchto značek, které probíhalo v cyklu, bylo zdlouhavé a každé použití funkce pro odstranění jednotlivých značek způsobovalo volání dalších funkcí na pozadí, které celý proces zpomalovaly. Toto chování bylo nahrazeno použitím funkce `clearLayers`, která zajistila smazání všech značek najednou a celý proces velmi urychlila. Před optimalizací proces trval přibližně 200 milisekund a pro sadu s desetinásobným počtem geografických objektů to bylo přes 4 sekundy.

Podobně jako u předchozího problému, i zde optimalizace tohoto problému přinesla různě velká zrychlení v závislosti na velikosti sady s geografickými objekty. Na obrázku 8.6 je zobrazen graf zrychlení po optimalizaci, kde na vodorovné ose grafu jsou znázorněny násobky počtu geografických objektů, které byly použity a na svislé ose je znázorněno průměrné zrychlení. Po optimalizacích obou zmíněných funkcí bylo opět provedeno profilování na simulované sadě geografických objektů představujících senzory, které ukázalo, že změny implementace pro zobrazení vrstvy se značkami přinesly přibližně 15-násobné zrychlení (viz. obrázek 8.7).

Obrázek 8.6: Graf zrychlení odstranění značek z vrstev po optimalizaci



Obrázek 8.7: Výsledek profilování funkce pro zobrazení vrstvy se značkami při použití geografické sady o 770 značkách po optimalizaci

Kapitola 9

Závěr

Cílem této práce bylo navrhnout rozhraní pro automatizované testování knihovny Geovisto. Záměrem bylo usnadnit proces testování a zároveň jej učinit efektivnějším. V rámci teoretické části jsem definoval, co to jsou geografická data a geografické objekty, které jsou s nimi spojeny. Dále jsem provedl průzkum možností zpracování geografických dat a možnosti jejich vizualizace v rámci tematických map.

Následně jsem definoval pojem testování, představil různé druhy testů a provedl průzkum nástrojů pro testování a profilování, které je možné využít pro projekty v jazyce JavaScript/TypeScript. Mezi těmito nástroji jsem provedl jejich srovnání a shrnutí jejich výhod a nevýhod.

Dále jsem provedl analýzu problému testování knihovny Geovisto – určil cílovou skupinu uživatel, jejich potřeby a provedl průzkum aktuálních řešení problému. Následně byl definován samotný problém a vytvořen návrh řešení tohoto problému. Návrh řešení zahrnoval návrh architektury rozhraní pro automatizované testování a výčet jednotkových a end-to-end testů, které jsou v rámci práce implementovány. V rámci návrhu řešení byl taktéž zahrnut návrh uživatelského rozhraní a datového modelu.

Následně bylo samotné rozhraní pro automatizované testování knihovny Geovisto implementováno. Dále bylo provedeno otestování funkcionality rozhraní pro automatizované testování a s pomocí implementovaných testů, které byly navrhnuty, byly otestovány jednotlivé nástroje knihovny. V rámci implementace bylo také provedeno profilování knihovny Geovisto. Z výsledků profilování byly následně provedeny optimalizační změny implementace knihovny Geovisto, které měly za následek výrazné zvýšení výkonnosti některých funkcí knihovny.

Implementací rozhraní pro automatizované testování knihovny Geovisto se podařil zefektivnit proces testování této geografické knihovny. Rozhraní umožňuje vývojářům mnohem efektivnější vývoj této knihovny a poskytuje přehled nad jednotlivými nástroji knihovny, že fungují správně. Do budoucna lze pokládat jako hlavní vylepšení rozhraní implementaci dalších testů tak, aby funkce jednotlivých nástrojů knihovny byly co nejlépe pokryty testy.

Literatura

[1] Alshamrani, A. a Bahattab, A. A comparison between three SDLC models waterfall model, spiral model, and Incremental/Iterative model. *International Journal of Computer Science Issues (IJCSI)*. International Journal of Computer Science Issues (IJCSI). 2015, sv. 12, č. 1, s. 106.

[2] Bacinger, T. *A map to perfection: Using d3.js to make beautiful web maps*. Toptal, prosinec 2014. Dostupné z: https://www.toptal.com/javascript/a-map-to-perfection-using-d3-js-to-make-beautiful-web-maps.

[3] Balaji, S. a Murugaiyan, M. S. Waterfall vs. V-Model vs. Agile: A comparative study on SDLC. *International Journal of Information Technology and Business Management*. 2012, sv. 2, č. 1, s. 26–30.

[4] Bekkhus, S. *Getting started · jest*. říjen 2021. Dostupné z: https://jestjs.io/docs/getting-started.

[5] Butler, H., Daly, M., Doyle, A., Gillies, S., Schaub, T. et al. *The GeoJSON Format* [RFC 7946]. RFC Editor, 2016. DOI: 10.17487/RFC7946. Dostupné z: https://rfc-editor.org/rfc/rfc7946.txt.

[6] Cohen, D., Lindvall, M. a Costa, P. Agile software development. *DACS SOAR Report*. 2003, sv. 11, s. 2003.

[7] Dent, B. *Cartography : thematic map design*. New York: McGraw-Hill Higher Education, 2009. ISBN 9780072943825.

[8] Docs, M. W. *Express/node introduction - learn web development: MDN*. MDN Web Docs, Apr 2022. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction.

[9] Environmental Systems Research Institute, I. *ESRI Shapefile Technical Description*. 1998. Dostupné z: https://www.esri.com/content/dam/esrisites/sitecore-archive/Files/Pdfs/library/whitepapers/pdfs/shapefile.pdf.

[10] Everett, G. D. a McLeod Jr, R. Software Testing. *Testing Across the Entire*. 2007.

[11] Farkas, G. *Mastering OpenLayers 3*. Packt Publishing Ltd, 2016.

[12] Freeman, H. Software testing. *IEEE instrumentation & measurement magazine*. IEEE. 2002, sv. 5, č. 3, s. 48–50.

[13] Georgian, S. *What is end-to-end testing and when should you use it?* freeCodeCamp.org, březen 2021. Dostupné z: https://www.freecodecamp.org/news/end-to-end-testing-tutorial/.

[14] Ghaderpour, E. Some Equal-area, Conformal and Conventional Map Projections: A Tutorial Review. *Journal of Applied Geodesy*. 2016, sv. 10, s. 197 – 209.

[15] Google. *Open Location Code – opensource.google*. Dostupné z: https://opensource.google/projects/open-location-code.

[16] Grossmann, J. *Informační systém pro správu vizualizací geografických dat*. 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

[17] Group, S. W. *Scalable Vector Graphics (SVG) 2*. říjen 2018. Dostupné z: https://www.w3.org/TR/SVG2/.

[18] Gundecha, U. a Avasarala, S. *Selenium WebDriver 3 Practical Guide: End-to-end automation testing for web and mobile browsers with Selenium WebDriver, 2nd Edition*. Packt Publishing, 2018. ISBN 9781788996013. Dostupné z: https://books.google.cz/books?id=_AhnDwAAQBAJ.

[19] Hynek, J., Kachlík, J. a Rusnák, V. Geovisto: A Toolkit for Generic Geospatial Data Visualization. In: *VISIGRAPP (3: IVAPP)*. 2021, s. 101–111.

[20] Hák, T. *Testing V Agilním Prostředí*. Září 2020. Dostupné z: https://www.tesena.com/novinky/testing-v-agilnim-prostredi.

[21] Janzen, D. a Saiedian, H. Test-driven development concepts, taxonomy, and future direction. *Computer*. IEEE. 2005, sv. 38, č. 9, s. 43–50.

[22] Jorgensen, P. C. a Erickson, C. Object-oriented integration testing. *Communications of the ACM*. ACM New York, NY, USA. 1994, sv. 37, č. 9, s. 30–38.

[23] Limaye, M. G. *Software testing*. Tata McGraw-Hill Education, 2009.

[24] Miller, R. a Collins, C. T. Acceptance testing. *Proc. XPUniverse*. 2001, sv. 238.

[25] Mozilla. *Drawing graphics - learn web development: MDN*. Leden 2022. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Drawing_graphics.

[26] MWAURA, W. *AUTOMATION TESTING WITH CYPRESS explore modern techniques to automate frontend testing with... cypress and javascript*. S.l: PACKT PUBLISHING LIMITED, 2021. ISBN 978-1-83921-385-4.

[27] Myers, G. J., Sandler, C. a Badgett, T. *The art of software testing*. John Wiley & Sons, 2011.

[28] Olan, M. Unit testing: test early, test often. *Journal of Computing Sciences in Colleges*. Consortium for Computing Sciences in Colleges. 2003, sv. 19, č. 2, s. 319–328.

[29] ORMELING, F. *CARTOGRAPHY visualization of geospatial data, fourth edition*. CRC Press, 2020. ISBN 9781138613959.

[30] Patton, R. *Software testing.* Indianapolis, IN: Sams Pub, 2006. ISBN 0-672-32798-8.

[31] Perforce Software, I. *A comprehensive guide to end to end (E2E) testing: By perforce.* Srpen 2020. Dostupné z: https://www.perfecto.io/blog/comprehensive-guide-end-end-e2e-testing.

[32] Peňáz, T. *Metoda proporcionalních bodových znaků.* Katedra geoinformatiky, Hornicko-geologická fakulta, VŠB-Technická univerzita Ostrava, červenec 2014. Dostupné z: http://tk.vsb.cz/?page_id=54.

[33] Ramya, P., Sindhura, V. a Sagar, P. V. Testing using selenium web driver. In: *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT).* 2017, s. 1–7. DOI: 10.1109/ICECCT.2017.8117878.

[34] Rather, M. A. a Bhatnagar, M. V. A comparative study of software development life cycle models. *Int. J. Appl. or Innov. Eng. Manag.* 2015, sv. 4, č. 10, s. 24–26.

[35] Ravindranath, H. *Jest vs mocha vs Jasmine: Comparing the top 3 javascript testing frameworks.* Listopad 2021. Dostupné z: https://www.lambdatest.com/blog/jest-vs-mocha-vs-jasmine/.

[36] Ribecca, S. *Choropleth map.* Dostupné z: https://datavizcatalogue.com/methods/choropleth.html.

[37] Ritter, N. a Ruth, M. The GeoTiff data interchange standard for raster geographic images. *International Journal of Remote Sensing.* Taylor & Francis. 1997, sv. 18, č. 7, s. 1637–1647.

[38] Rolf, A., By, R. de et al. Principles of geographic information systems. *The International Institute for Aerospace Survey and Earth Sciences (ITC), Hengelosestraat.* 2001, sv. 99.

[39] Sedláček, P. *Lekce 3 - Rozběhnutí Projektu a první řádky V expressu.* 2019. Dostupné z: https://www.itnetwork.cz/javascript/nodejs/rozbehnuti-projektu-a-prvni-radky-v-expressu.

[40] Shellito, B. *Introduction to geospatial technologies.* New York, NY: W.H. Freeman and Co, 2012. ISBN 978-1-4292-5528-8.

[41] Singh, S. K. a Singh, A. *Software testing.* Vandana Publications, 2012.

[42] Slocum, T. A. *Thematic cartography and geovisualization.* Pearson Education Limited, 2014.

[43] Standardization, I. O. for. *ISO 3166 - country codes.* červen 2021. Dostupné z: https://www.iso.org/iso-3166-country-codes.html.

[44] Vaidya, N. *All you need to know about selenium WebDriver architecture.* červenec 2021. Dostupné z: https://www.edureka.co/blog/selenium-webdriver-architecture/.

[45] Wernecke, J. *The KML Handbook: Geographic Visualization for the Web.* Pearson Education, 2008. ISBN 9780321606617.

[46] Wun, B. *Survey of Software Monitoring and Profiling Tools*. Dostupné z: https://www.cse.wustl.edu/~jain/cse567-06/ftp/sw_monitors2/index.html.

[47] Čerba, O. *Metoda teček*. Katedra geomatiky, Fakulty aplikovaných věd, Západočeské univerzity v Plzni, září 2004. Dostupné z: http://geomatika.kma.zcu.cz/studium/tka/Slides/metoda_tecek.pdf.

Příloha A

Obsah přiloženého paměťového média

/
- `README` ..... Soubor popisující obsah média
- `thesis` ..... Složka s textem diplomové práce a LaTeX soubory
  - `thesis.pdf` ..... Text diplomové práce
  - `latex` ..... Zdrojové soubory LaTeX
- `src` ..... Zdrojové kódy diplomové práce
  - `geovisto-fe` ..... Zdrojové kódy frontendové části informačního systému
  - `geovisto-be` ..... Zdrojové kódy backendové části informačního systému
  - `geovisto-map` ..... Zdrojové kódy knihovny Geovisto