VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

PROGRAMOVÁ ABSTRAKCE KNIHOVNY GEOVISTO

PROGRAMMING ABSTRACTION OF GEOVISTO LIBRARY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. MARTIN CHLÁDEK**
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: **Chládek Martin, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Informační systémy a databáze
Název: **Programová abstrakce knihovny Geovisto**
Programming Abstraction of Geovisto Library
Kategorie: Uživatelská rozhraní
Zadání:

1. Prostudujte problematiku zpracování a vizualizace geografických dat na webu a prozkoumejte existující technologie určené pro tento účel (Leaflet, d3-geo, Geovisto, apod.).
2. Seznamte se s existujícími webovými frameworky pro tvorbu uživatelských rozhraní (UI) a UI komponent (např. React, Angular, Vue, Svelte, apod.). Porovnejte tyto frameworky. Prostudujte dostupné knihovny pro podporu vývoje UI komponent v jednotlivých UI frameworcích (např. Storybook).
3. Analyzujte knihovnu Geovisto a její současná rozšíření. Definujte požadavky programátorů na použití knihovny se současnými webovými frameworky pro tvorbu UI.
4. Navrhněte rozšíření knihovny Geovisto, které poskytne podporu pro vybrané webové frameworky pro tvorbu UI.
5. Navržené rozšíření implementujte formou několika knihoven pro jednotlivé frameworky.
6. Výsledné řešení otestujte.

Literatura:

- Hynek, J., Kachlík, J. a Rusňák, V.: Geovisto: *A Toolkit for Generic Geospatial Data Visualization*. In *VISIGRAPP (3: IVAPP)* (pp. 101-111).
- Dent, B., D., a spol.: *Cartography: Thematic Map Design*. McGraw-Hill Higher Education 2009, ISBN 978-128-3388-023.
- Leaflet: *Leaflet API reference* [online]. 2021 [cit. 2021-10-09]. Dostupné z: https://leafletjs.com/reference-1.7.1.html
- Facebook Inc.: *React: Getting Started* [online]. 2021 [cit. 2021-10-09]. Dostupné z: https://reactjs.org/docs/getting-started.html
- Google: *Introduction to the Angular Docs* [online]. 2021 [cit. 2021-10-09]. Dostupné z: https://angular.io/docs
- You E.: *Vue.js: Introduction* [online]. 2021 [cit. 2021-10-09]. Dostupné z: https://vuejs.org/v2/guide/
- Storybook community: *Introduction to Storybook for React* [online]. 2021 [cit. 2021-10-09]. Dostupné z: https://storybook.js.org/docs/

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

Tato semestrální práce vznikla jako rozšíření knihovny Geovisto, které si klade za cíl usnadnit možnosti vizualizace geografických dat v moderních webových aplikacích. Předmětem této práce je provedení analýzy použitelnosti nástroje ve spojitosti s webovými frameworky pro tvorbu uživatelských rozhraní a návrh rozšíření podporujícího snadnou integraci nástroje do projektu klienta. Výstupem řešení je vytvoření vrstvy programové abstrakce nad jádrem knihovny a jejími moduly, jež by umožnila vývojářům konfigurovat vrstvy tematické mapy užitím deklarativního přístupu. Realizované rozšíření je tvořeno sadou předdefinovaných komponent v paradigmatu aplikačního rámce React. Prostřednictvím komponent je možné zobrazení mapy inicializovat a dynamicky měnit vykreslený obsah. Řešení je exportováno jako samostatná knihovna a bylo vytvořeno užitím především technologií React a TypeScript.

Abstract

The purpose of this thesis is to create an extension of the Geovisto library, which aims to make the visualization and presentation of geographic data in modern web applications easier. This work analyzes possible integrations with popular web UI frameworks and design extensions that help integrate the tool within client projects. The goal is to develop an abstraction for the core and other modules of the Geovisto library, which enables declarative configuration of visualized map layers by users of the abstraction code. The implemented extension consists of a set of predefined configurable React components. Using the components makes it possible to initialize the map view and dynamically change the rendered content. The solution is exported as a separate library and was created using mainly the technologies React and TypeScript.

Klíčová slova

Geovisto, vizualizace geografických dat, programová abstrakce, React, TypeScript

Keywords

Geovisto, geographic data visualization, programming abstraction, React, TypeScript

Citace

CHLÁDEK, Martin. *Programová abstrakce knihovny Geovisto.* Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jiří Hynek, Ph.D.

Programová abstrakce 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.

....
Martin Chládek
16. května 2022

Poděkování

Tímto bych velmi rád poděkoval mému vedoucímu práce panu Ing. Jiřímu Hynkovi, Ph.D. za obrovskou ochotu a pomoc při řešení, věcné rady, čas strávený konzultacemi a za odborné vedení mé diplomové práce.

Obsah

1 Úvod 3

2 Geografická data a způsoby jejich vizualizace 5
2.1 Geografická data 5
2.2 Formáty geografických dat 6
2.2.1 GeoJSON 6
2.2.2 GML 7
2.2.3 Shapefile 8
2.3 Vizualizace geografických dat 8
2.3.1 Kartogram 9
2.3.2 Mapa symbolů 9
2.3.3 Mapa spojení 10
2.3.4 Teplotní mapa 11
2.4 Nástroje k programové vizualizaci 11
2.4.1 d3-geo 12
2.4.2 Leaflet 12
2.4.3 Geovisto 13

3 Webové frameworky pro tvorbu UI 14
3.1 Porovnání UI frameworků 16
3.2 Podrobná analýza 22
3.2.1 React 22
3.2.2 Angular 26
3.2.3 Vue.js 30
3.3 Podpůrné nástroje pro vývoj komponent 32
3.3.1 Storybook 32
3.3.2 Styleguidist 34
3.3.3 Bit 34

4 Analýza 35
4.1 Charakteristiky knihovny Geovisto 35
4.2 Architektura knihovny Geovisto 37
4.3 Použití Geovisto s webovými UI frameworky 39
4.4 Shrnutí 39

5 Návrh rozšíření **41**
5.1 Uvažované přístupy . . . 41
5.2 Návrh sady komponent . . . 43
5.3 Výstup . . . 45

6 Implementace **46**
6.1 Architektura řešení . . . 46
6.2 Inicializace objektu mapy . . . 50
6.3 Aktualizace konfigurace komponenty . . . 52
6.3.1 Aktualizace identifikátoru . . . 55
6.3.2 Aktualizace stavu zobrazení . . . 56
6.4 Uživatelské moduly . . . 59

7 Testování **61**

8 Závěr **65**

Literatura **66**

A Rozšíření knihovny o nový oficiálně podporovaný modul **69**

Kapitola 1

Úvod

Aniž bychom si to možná uvědomovali, geografická data se stala součástí našich životů a máme možnost se s nimi setkat každý den. Může se jednat například o uložené souřadnice v metadatech pořízené fotografie, naplánovanou trasu do oblíbené restaurace v navigaci vozu či grafické zobrazení průměrného platu v jednotlivých krajích. Tato data jsou specifická tím, že se vážou k určitému místu. Abychom geografická data mohli dlouhodobě uchovávat, je potřeba vyřešit otázku, v jakém formátu je uložit. Jako dlouhodobě nejefektivnější řešení se ukázalo uchovávat informace o konkrétním místě v podobě zeměpisných souřadnic. Tento formát je efektivní při velkém množství dat a snadno interpretovatelný pro stroje. Nese s sebou ale i nevýhody, ta hlavní je přitom vcelku zásadní: špatná čitelnost pro člověka. Pokud bychom známému poslali místo našeho setkání ve formě zeměpisné šířky a délky, pravděpodobně by na místo nedorazil. Člověk v tomto případě potřebuje znát konkrétní bod na mapě, aby byl schopen informaci správně uchopit. Geografická data je tedy potřeba vizualizovat.

Ačkoliv existuje k vytváření tematických map velké množství nástrojů, způsobem jejich konfigurace je lze klasifikovat do dvou tříd. Do první třídy spadají knihovny, kde uživatel pro vytvoření mapy musí mít programátorské, často i matematické znalosti. Druhou skupinu pak tvoří autorské nástroje, které jsou bližší běžným uživatelům, neboť nevyžadují žádné speciální dovednosti. Uživatel musí pouze poskytnout data, nicméně je odkázán na používání mapových vrstev z předem definovaných palet, což značně limituje možnosti perzonalizace.

Zde se snaží uplatnit knihovna Geovisto[1], která hledá kompromis mezi uvedenými dvěma přístupy. Cílem této knihovny je zkombinovat výhody dvou zmíněných přístupů a vyřešit problémy, se kterými se každý z nich potýká. Geovisto nabízí uživatelské rozhraní pro běžné uživatele, díky čemuž si mohou mapy přizpůsobovat dle svých představ i lidé bez nadstandardních počítačových znalostí. Současně však k tomu disponuje rozhraním pro vývojáře, kteří si tak mohou zahrnout mapy do svých aplikací a pomocí kódu je konfigurovat. Ovšem i toto řešení má své nedostatky.

S rostoucí popularitou webových aplikací vznikají náročnější požadavky na jejich tvorbu. Jejich implementace se stává čím dál komplexnější záležitostí, a tak programátoři sahají po nástrojích, které vývoj usnadní a činí kód snadněji udržitelným. Volba vývojářů pak obvykle spadá na některý z frameworků pro tvorbu uživatelských rozhraní. Pokud se však takový programátor rozhodne jako součást své aplikace využít nástroj Geovisto, může se dostat do nesnází. Zde totiž zmiňovaná knihovna nalézá prostor pro zlepšení, neboť navzdory

[1] https://github.com/geovisto

nabízenému aplikačnímu rozhraní není integrace do takového projektu snadná a vyžaduje i jistou míru znalosti vnitřní implementace knihovny.

Vyplnit zmíněný prostor pro zlepšení a usnadnit tak vývojářům práci je poté hlavní motivací této práce. K dosažení tohoto cíle je zapotřebí vytvořit vrstvu abstrakce, která by sloužila jako prostředník mezi UI frameworkem a samotnou knihovnou vykreslující tematické mapy a zajišťovala tak programátorovi snadnou práci s jejími moduly. Předmětem práce je implementovat sadu komponent abstrahující konkrétní moduly knihovny pro nejpopulárnější webový framework React. Výstupem bude publikovaný balíček obsahující zmíněné komponenty, které budou sloužit jako nástroj pro práci s knihovnou Geovisto.

Obsah dokumentu je strukturován do jednotlivých kapitol. Následující kapitola 2 pojednává o podstatě geografických dat a problematice formátů, ve kterých je možné data ukládat a šířit. Dále řeší možnosti jejich vizualizace a nástroje, které může vývojář za tímto účelem využít. V kapitole 3 je detailně popsán rozbor nejpopulárnějších frameworků pro tvorbu uživatelských rozhraní, jejich společné znaky, odlišnosti a důvody jejich oblíbenosti. Rovněž zde lze najít stručné představení podpůrných nástrojů, které jsou v praxi využívány pro efektivnější vývoj komponent. Kapitola 4 se věnuje analýze současného stavu rozšiřované knihovny a vymezuje požadavky na realizovanou nadstavbu. V navazující kapitole (kapitola 5) je popsán koncept destrukturalizace původního řešení a postupy, které byly užity při návrhu sad komponent. V kapitole 6 popíši konkrétní implementační detaily vytvářených programových vrstev. Zdrojový kód popsaný předchozí kapitole je vhodné podrobit testování, jehož popis je předmětem kapitoly 7.

Kapitola 2

Geografická data a způsoby jejich vizualizace

V moderním světě je velká část digitálních dat obohacena, či přímo tvořena geografickými daty. Obecnému představení geografických dat a způsobu jejich reprezentace se věnuje první část této kapitoly. Mít tisíce záznamů dat už není žádnou výjimkou a s rostoucími daty roste problematika dolování cenných informací. Aby bylo možné data čitelným způsobem prezentovat uživateli, je nutná jejich vizualizace. O potřebě geovizualizace pojednává třetí sekce kapitoly, přičemž součástí je i představení nejužívanějších zástupců tematických map. V závěru tohoto tématu popisuji nástroje, které mohou vývojáři moderních webových aplikací využít k vizualizaci tematických map ve svých projektech.

2.1 Geografická data

Geografická data jsou oproti běžným datům specifická, neboť jsou vztahována k určitému místu, či objektu na planetě a mají tak nějaký odkaz v prostoru. Příkladem mohou být geografické souřadnice (zeměpisná šířka a délka), názvy ulic, popisná čísla budov, poštovní směrovací čísla a další. Lze tak popisovat objekty reálného světa jako domy, silnice, pole, nebo abstraktnější území (například státy) a abstrahovat je do digitálního kartografického modelu popsaného sadou bodů, úseček a polygonů [20, 24].

Identifikátory států

Jak už bylo popsáno v předchozím odstavci, v mapách je možné referencovat i územní celky, přičemž jednou z nejčastěji odkazovaných oblastí jsou země. S cílem mezinárodně sjednotit kódy pro označení zemí a s nimi souvisejících oblastí definuje organizace ISO[1] standard ISO 3166 [19]. Kódy se mohou skládat pouze z písmen abecedy a arabských číslic. Definice názvů států ale není předmětem uvedeného standardu, jejich poskytovatelem je Organizace spojených národů.

Jednotné kódování je užitečné zejména díky jeho invarianci vůči světovým jazykům a je tedy stejné pro celý svět. S jeho porozuměním tak nemají problém lidé z různých koutů světa a pod konkrétním označením si dokáží vybavit název odkazovaného státu/území.

[1] International Organization for Standardization

Alpha-2

Norma značená jako ISO 3166-1 alpha-2 definuje kódové označení pro identifikaci jako kombinaci dvou písmen. Odkazovány tímto způsobem mohou být státy, závislá území a oblasti geografického zájmu. Vzhledem k limitovanému počtu kombinací dvou písmen abecedy se tato norma omezuje pouze na územní celky nejvyšších domén. Jedná se o organizací ISO obecně doporučený formát pro odkazování názvů zemí.

Česká republika nese v tomto formátu kódové označení *CZ*, kupříkladu Španělské království nese ve standardu ISO 3166-1 alpha-2 identifikátor *ES*.

Alpha-3

Součástí standardu ISO 3166-1 je i norma značená jako Alpha-3, která staví na předchozím přístupu Alpha-2, využívá však pro kódové označení tří písmen abecedy. Kombinace tří písmen poskytuje lepší vizuální asociaci mezi kódovým označením a skutečným názvem země.

V případě formátu Alpha-3 nese Česká republika označení *CZE*, dříve zmiňované Španělské království poté kód *ESP*.

2.2 Formáty geografických dat

Aby bylo možné s geografickými daty pracovat, je nutné je systematicky popisovat s využitím stanovené syntaxe, a reprezentovat je tak v určitém formátu. Každý formát má svá vlastní specifika, jakým způsobem je data schopen ukládat. Problémem může být také schopnost formátu reagovat na různé souřadnicové systémy, rozdílná měřítka, typy ukládaných dat či jejich přesnost. V případě rozsáhlých geografických datových sad může způsob formátování geografických dat značně ovlivnit velikost souboru a mít vliv na výslednou rychlost vykreslování.

2.2.1 GeoJSON

GeoJSON je formát pro výměnu geografických dat založený na JavaScriptové objektové notaci (JSON) a respektuje tedy jeho syntaxi. Každý objekt definovaný v tomto formátu může reprezentovat geometricky vyjádřený územní celek, prostorově vázanou entitu či případně jejich kolekci. K vyjádření pozice prostorových bodů na konkrétním místě na světě využívá souřadnicový systém definovaný standardem World Geodetic System 1984 [4].

GeoJSON pracuje s atomickými objekty, jakými jsou bod (*point*), lomená čára (*lineString*), mnohoúhelník (*polygon*) a se složenými kolekcemi *MultiPoint*, *MultiLineString*, *MultiPolygon* a *GeometryCollection*. Kombinací těchto elementárních geometrických útvarů vytváří reprezentaci geografických objektů, jejich vlastností a územních celků [4]. Názornou ukázku jednoduchého objektu ve formátu GeoJSON lze vidět na příkladu 2.1.

```
{
    "type": "Feature",
    "properties": {
        "name": "Coors Field",
        "amenity": "Baseball Stadium",
    },
    "geometry": {
        "type": "Point",
        "coordinates": [-104.99404, 39.75621]
    }
}
```

Výpis 2.1: Příklad kódování dat ve formátu GeoJSON

2.2.2 GML

Zkratka GML vychází z anglického *Geography Markup Language* a definuje formát pro ukládání a přenos geografických entit. Ty jsou ukládány ve formě textu a lze je tak stejně jako GeoJSON upravovat v jakémkoliv textovém editoru. GML staví na bázi XML a je možné ho tak chápat jako rozšíření XML za účelem zahrnutí podpory geografických souřadnic. Značky, o které je GML obohaceno, nesou v kódu prefix `gml`. Tento formát byl představen seskupením Open Geospatial Consortium a je definován standardem ISO 19136 [18].

Základními stavebními kameny, které GML používá pro vytváření komplexních geografických objektů jsou: bod (*point*), lomená čára (*lineString*), mnohoúhelník (*polygon*), z komplexnějších primitiv poté křivka (*curve*), plocha (*surface*) a další. Stejně tak jako GeoJSON využívá popisovaný formát referenční souřadnicový systém [18].

V následující ukázce 2.2 lze vidět georeferenci zakódovanou syntaxí GML na stejný bod se shodnými atributy, jako v případě příkladu předchozího formátu GeoJSON (2.1).

```
<?xml version="1.0" encoding="UTF-8"?>
<gml:FeatureCollection xmlns:gml="http://www.opengis.net/gml">
    <gml:featureMember>
      <name>Coors Field</name>
      <amenity>Baseball Stadiu,Field</amenity>
      <location>
        <gml:Point>
            <gml:coordinates>-104.99404, 39.75621</gml:coordinates>
        </gml:Point>
      </location>
    </gml:featureMember>
</gml:FeatureCollection>
```

Výpis 2.2: Příklad kódování dat ve formátu GML

Při porovnání příkladu 2.2 zapsaném ve formátu GML a ukázky 2.1 kódovaném ve formátu GeoJSON, lze pozorovat větší množství znaků nutných pro vyjádření stejného množství informace. Tento rozdíl vychází z obsáhlosti zápisu v jazyce XML. Při velkém

počtu geometrických objektů a uchovaných vlastností vede uvedený problém k většímu objemu výsledného `.gml` souboru v porovnání s formátem GeoJSON.

2.2.3 Shapefile

Shapefile je jeden z běžně užívaných formátů pro výměnu vektorových dat. Vyvinut a spravován je soukromou společností Esri[2]. Obdobně jako dva předchozí popisované formáty GeoJSON a GML skládá komplexní objekty ze základních vektorových entit. Těmi jsou v případě formátu Shapefile body, liniové prvky (úsečky, popř. lomené čáry) a plošné prvky (uzavřené smyčky či mnohoúhelníky). Formát samotný není schopen ukládat topologické informace o geografických objektech a tak je nutné ho doplnit o další soubory, z nichž dva jsou povinné. Pro celistvý popis geoprostorových dat je tak nutné distribuovat následující tři typy souborů [11]:

- `.shp` – hlavní soubor tvořený kolekcí geometrických primitiv, které jsou definovány pro určité světové souřadnice.
- `.dbf` – databázová tabulka obsahující atributy jednotlivých prvků.
- `.shx` – indexová reference spojující geometrický objekt se záznamem v atributové tabulce.

Pro provázání souborů je nutné, aby všechny sdílely shodný název kořene. Díky absenci topologické struktury dat disponují soubory formátu Shapefile vyšší rychlostí vykreslování objektů a snadnější úpravy. Obvykle také data definovaná v těchto souborových formátech zabírají méně diskového prostoru [11].

2.3 Vizualizace geografických dat

Pro vizualizaci geografického obsahu neboli geovizualizaci existuje velké množství různých definic. Například MacEachren a Kraak [24] definují geovizualizaci jako „integraci přístupů vizualizace v informatice, kartografie, analýzy obrazu, vizualizace informací, průzkumné analýzy dat a geografických informačních systémů za účelem poskytnutí teorie, metod a nástrojů k vizuálnímu průzkumu, analýze, syntéze a prezentace geoprostorových dat“. Jednodušeji řečeno se geovizualizace snaží konzumentovi prezentovat pomocí vizuálního rozhraní geografická data za použití různých výpočetních metod.

V současnosti obrovské množství digitálních dat zahrnuje georeferencování, problémem je ale jejich velikost a komplexita. Data sama o sobě totiž nemají pro člověka žádnou vypovídající hodnotu, ale je zapotřebí jejich transformace na informace a následně na znalosti tak, aby v určitém kontextu přinášela užitek. Zde nalezly své uplatnění metody a nástroje pro vizualizaci, neboť prezentují data uživateli způsobem, který dokáže rychle a snadno uchopit [24].

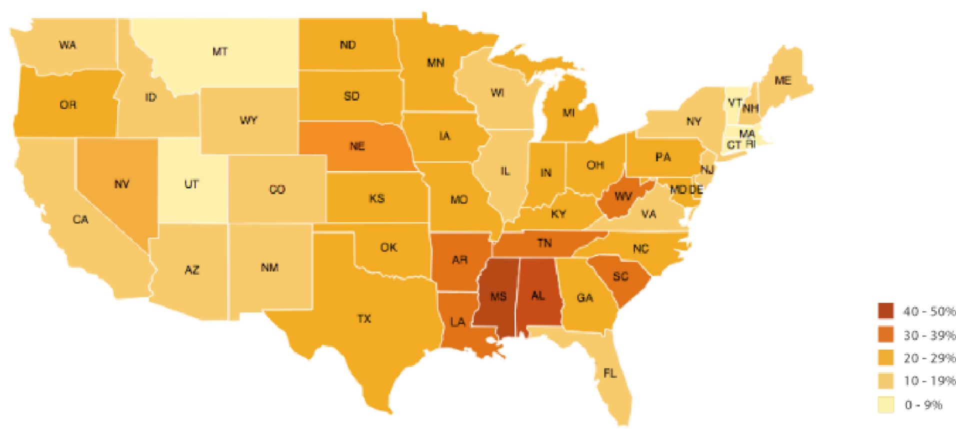
Jedním ze způsobů, jak informace předat, je reprezentace v podobě tematických map. Tematické mapy se soustředí na ilustraci typicky jedné charakteristiky dat a zasazují je do kontextu geografického objektu. Využívají k tomu grafické prvky (body, mnohoúhelníky, apod.) nad určitým mapovým podkladem [8].

V rámci této sekce jsou blíže popsány jen některé z typů tematických map, tato třída ale zahrnuje mnohem více způsobů ilustrace dat. Dalšími zástupci jsou například tematické mapy typu *isopleth*, metoda teček a další.

[2] Environmental Systems Research Institute

2.3.1 Kartogram

Kartogram (anglicky *choropleth map*) je typ tematické mapy vhodný zejména pro vizualizaci jedné diskrétní proměnné a její fluktuace napříč definovanými regiony nebo geopolitickými územními celky. Vzhledem ke konceptu mapy jsou data očekávána vždy v podobě jedné hodnoty pro danou plochu, v opačném případě mohou být do tohoto formátu agregována. Každá vyobrazená geografická jednotka (země, stát, okres, apod.) je vybarvena barvou různé intenzity, mírou saturace, stínováním či vzorem v závislosti na hodnotě statistické proměnné. Obvykle platí, že s rostoucí hodnotou proměnné roste i intenzita zvolené barvy, jak je tomu na příkladu 2.1 [8, 20].



Obrázek 2.1: Příklad tematické mapy typu kartogram[3]

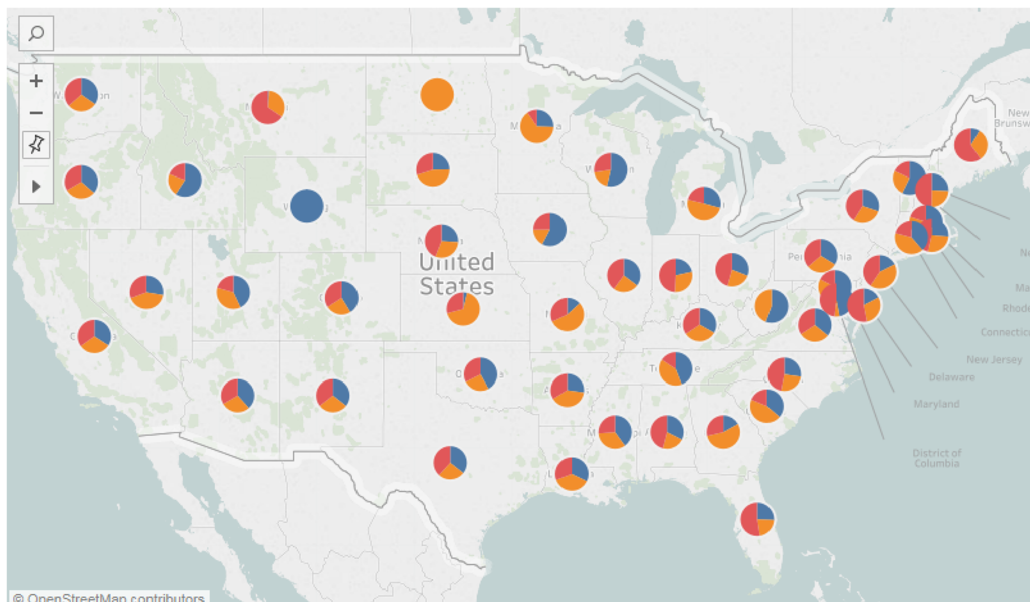
Kartogramy nemusí být nutně omezené na zobrazování jediného atributu (a tedy odstínů jedné barvy), ale je možné zahrnout atributů více a vytvořit z nich třídy. Užitečným příkladem mohou být výsledky prezidentských voleb, kde každému kandidátovi je přidělena jedna barva a jednotlivé územní celky jsou obarveny dle vítěze v konkrétní oblasti a saturovány v závislosti na velikosti rozdílu.

2.3.2 Mapa symbolů

Mapa symbolů (anglicky *symbol map*) umisťuje do středu zobrazovaných regionů typicky kruhovou značku, která může demonstrovat informaci o poměrném zastoupení jednotlivých veličin pro daný územní celek (například pomocí výsečového diagramu – viz obrázek 2.2) nebo promítá výši hodnoty statistické proměnné do velikosti zobrazované značky [8].

Zobrazované symboly jsou relativní k měřítku mapy. Se snižováním měřítka, a s tím souvisejícím snižováním detailů mapy, lze data shlukovat. Hodnoty se poté v závislosti na měřítku a vzdálenosti ostatních značek kumulují do větších celků a jsou zobrazovány do geografických územních celků vyšší domény [8]. Příkladem by mohla být mapa symbolů vyjadřující nezaměstnanost v okresech České republiky, při zmenšení měřítka mapy by se data agregovala a zobrazovala souhrnně pro jednotlivé kraje.

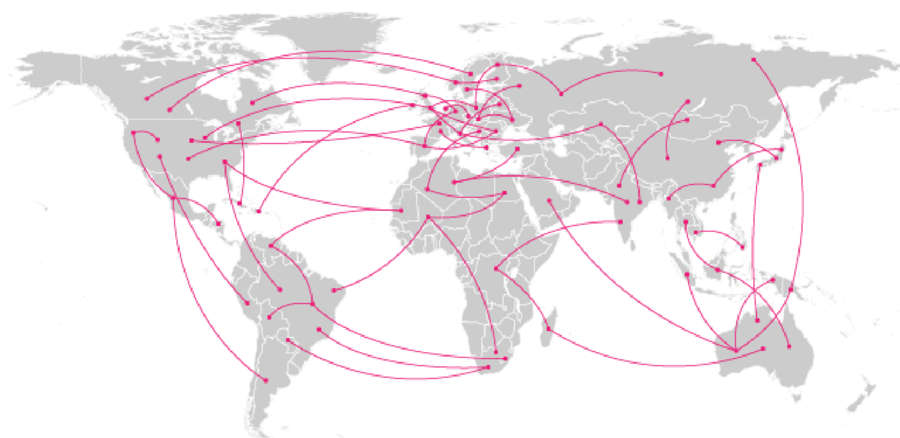
[3]https://datavizcatalogue.com/methods/choropleth.html



Obrázek 2.2: Příklad mapy symbolů s využitím výsečových diagramů[4]

2.3.3 Mapa spojení

Jak už název napovídá, mapa spojení (anglicky *connection map*) vizualizuje propojení mezi bodem A a bodem B definovanými zeměpisnou šířkou a délkou. Spojení může být realizováno přímkou nebo tzv. ortodromou. Ortodroma je nejkratší přímka spojující dva body na kulové ploše, v tomto případě na povrchu Země. V mapách spojení bývá ortodroma preferovanou variantou propojení bodů [42]. Příklad mapy spojení je k nahlédnutí na obrázku 2.3.



Obrázek 2.3: Příklad mapy spojení[5]

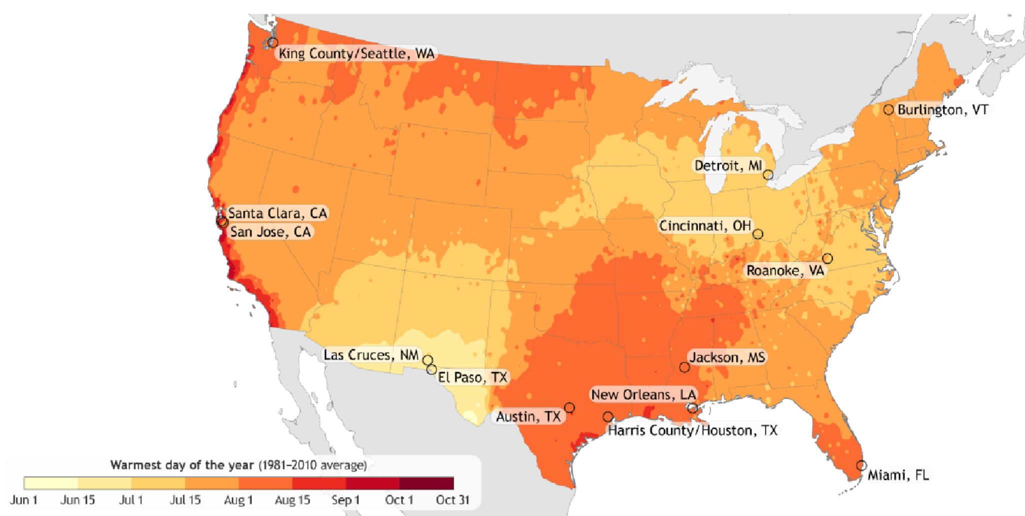
[4]https://vizzingdata.com/tableau-charts-symbol-filled-maps/
[5]https://datavizcatalogue.com/methods/connection_map.html

2.3.4 Teplotní mapa

Teplotní mapa (anglicky *heat map*) reprezentuje intenzitu výskytu události v rámci datové sady. Podobně jako kartogram využívají teplotní mapy variace zabarvení k vyjádření intenzity, na rozdíl od kartogramů ale řeší pouze frekvenci výskytu v určitém konkrétním bodě souřadného systému. Nejsou tak ohraničeny geografickými či geopolitickými hranicemi, jako je tomu v případě kartogramů, které navíc v rámci těchto území data seskupují [40].

Často nalézají využití ve vizualizování počasí (například zobrazení intenzity srážek, teploty, oblačnosti, apod.) a sledování přírodních fenoménů, kde za zástupce lze vybrat například rozsah zemětřesení či tornáda. V uvedených případech přirozeně není pro vizualizaci vyhovující respektovat politicky definované územní celky.

Jak lze vidět na příkladu níže, rozložení hodnot není ohraničeno geopolitickými hranicemi jako je tomu u kartogramu (2.1). Na obrázku 2.4 lze vidět teplotní mapu zobrazující kohorty časových intervalů, ve kterých průměrně nastaly v USA nejteplejší dny za období let 1981-2010.



Obrázek 2.4: Příklad teplotní mapy[6]

2.4 Nástroje k programové vizualizaci

Kapitola 2.3 popisuje argumenty, proč je vhodné geografická data vizualizovat. Nejčastěji volenou možností, jak toho docílit v moderních webových aplikacích, je využití některé z dostupných JavaScriptových knihoven, které vývojář zahrne do svého projektu. Konfiguraci knihovny s cílem dosáhnout žádoucího vykreslení pro koncového uživatele je nutné provádět ve zdrojovém kódu, jedná se tedy o programový přístup ke geovizualizaci.

Za předpokladu, že by se vývojář při vytváření tematických map vydal cestou nepoužívat žádný z podpůrných nástrojů, byl by nucen přímo zavádět SVG elementy do objektového modelu dokumentu a následně řešit, jak polygony definované geografickými souřadnicemi převést do rozdílného souřadného systému, který reprezentuje vykreslenou projekci. Tento přístup vyžaduje matematické znalosti a často i komplexní funkce. Je tak možné sáhnout

[6] https://www.weathernationtv.com/news/noaa-and-community-scientists-to-map-hottest-parts-of-13-cities-this-summer/

po řešení, které manipulaci s elementy usnadňuje a poskytuje sadu funkcí pro projekci bodů určených zeměpisnou šířkou a délkou do konkrétních 2D zobrazení jiného souřadného systému. Zástupcem takové skupiny je balíček d3-geo z rodiny D3.js, blíže popsaný v sekci 2.4.1.

Zmíněný přístup umožňuje velkou škálu personalizace výstupu, ale za cenu obrovské režie, neboť je implementačně velmi náročný. Uživatel musí zajistit všechny podpůrné mechanismy, včetně reakcí na uživatelskou interakci s vykreslenou mapou. Nabízí se tak použít řešení, které implementuje základní kostru operací, standardní projekci a abstrahuje práci s elementy mapy. Vývojář objekt mapy a jeho vrstvy nastavuje pomocí aplikačního rozhraní knihovny. Představiteli takových řešení jsou například knihovna Leaflet popsaná v sekci 2.4.2, Google Maps, OpenLayers a Mapbox GL JS.

Další vrstvu abstrakce přidává knihovna Geovisto (představená v sekci 2.4.3), která odstiňuje programátora od potřeby znalosti implementace konkrétních vrstev a umožňuje uživateli vrstvy tematické mapy vytvářet a konfigurovat deklarativním způsobem. Vývojář tak nemusí procedurálně přidávat polygony, vrstvy a geodata do objektu mapy a může se více soustředit na vizualizaci poskytnutých dat.

2.4.1 d3-geo

D3 (nebo také *Data-Driven Documents* či *D3.js*) je populární knihovna, která umožňuje ve webových aplikacích vizualizovat data a nad zobrazením zprostředkovávat interakci uživatele za pomocí manipulace objektového modelu dokumentu. Za tímto účelem využívá technologie SVG, Canvas a HTML [7]. Nad knihovnou D3 existuje velké množství rozšíření, která umožňují vykreslovat například interaktivní grafy, stromové struktury, nejrůznější objekty, křivky a další.

Jedním z rozšíření je i knihovna *d3-geo*. Tato knihovna se soustředí zejména na geografické projekce, kulovité tvary a sférickou trigonometrii. Lze s jejím využitím vytvářet tematické mapy. Nástroj *d3-geo* pracuje s objekty diskrétní geometrie: s mnohoúhelníky a lomenými čárami. Jejich projekce z kulovitého objektu do roviny je komplikovaná, protože hrany sférických geometrických objektů nejsou ve skutečnosti na promítané ploše čárami, ale geodetickými křivkami respektujícími tvar na který je mapa vykreslována (tzv. *ortodroma*). Popisovaná knihovna tuto problematiku řeší a nabízí škálu běžných i neobvyklých mapových projekcí [6].

K reprezentování geografických objektů je využíváno formátu GeoJSON. Při použití externího balíčku je možné použít i formát Shapefile [6].

2.4.2 Leaflet

Jedním z nejpopulárnějších JavaScriptových nástrojů pro vizualizaci map je v současnosti knihovna Leaflet. Za jejím vznikem stojí autor Vladimir Agafonkin a dle oficiální definice z dokumentace knihovny [22] nástroj cílí na tři faktory: jednoduchost, výkonnost a použitelnost. Leaflet umožňuje vytváření interaktivních map pomocí definování vrstev přidávaných nad bázovou mapou, přičemž počítá i s podporou mobilních zařízení.

Prvním krokem k zobrazení mapy je vytvoření kontejneru, do kterého bude mapa promítána. V HTML dokumentu je toho docíleno užitím blokového elementu `<div>`, důležité je pro něj nastavit parametry výšky a šířky. Propojení mezi JavaScriptovým objektem reprezentujícím definici mapy (poskytnutým knihovnou Leaflet) a kontejnerem v dokumentovém modelu je realizováno pomocí jeho identifikátoru (atributu `id`).

Vývojář musí objektu mapy poskytnout bázovou podkladovou rastrovou vrstvu, kterou může získat odkázáním na API od nějakého z nezávislých poskytovatelů (např. *OpenStreetMap*, *Google Maps*, *Esri* a další). Nad touto bázovou vrstvou poté programátor definuje své vlastní vrstvy, které jsou vysoce konfigurovatelné. Lze přidat značky (anglicky *marker*) pro konkrétní zeměpisné souřadnice, definovat pro ně vlastní ikonu, stylování, přidat vyskakovací popis a spoustu dalších možností personalizace. Další možností je přidání vrstvy s rastrovým nebo vektorovým obsahem překrývající podkladovou plochu. Pro všechny objekty je možné definovat reakce na uživatelské interakce s mapou.

Snahou je udržet jádro knihovny útlé, další možností jak posunout schopnosti knihovny je tak přidáním některého z doplňků třetích stran. Za jejich vývojem stojí programátorská komunita podporující tuto knihovnu. S využitím příslušných doplňků lze pomocí knihovny Leaflet animovat chování mapy, spravovat shlukování objektů či vytvářet tematické mapy jako kartogramy, teplotní mapy, a další.

Nativně knihovna Leaflet podporuje práci s formáty GeoJSON a WSM, přičemž do formátu GeoJSON umí i přímo stav mapy exportovat. Pro podporu některých dalších formátů nabízí knihovna doplňky, které vývojář do svého projektu zahrne a získá tak možnost s nimi pracovat. Příkladem jsou například SHP (shapefile), geoTIFF, TileJSON a další. Způsob definování geografických dat pomocí formátu GML (popsaný v sekci 2.2.2) není ke dni psaní práce knihovnou Leaflet oficiálně podporovaný.

V současnosti poslední vydanou verzí knihovny je verze 1.7.1, vývojáři ale již pracují na nové minoritní verzi 1.8.

2.4.3 Geovisto

Předchozí dva uvedené nástroje jsou zástupci čistě programového přístupu. Tento způsob ale klade vysoké nároky na jeho použití, neboť vyžaduje od uživatele znalosti programování. Knihovna Geovisto kombinuje programový přístup s možností úprav připravených šablon pomocí uživatelského rozhraní. Na technické vzdělání uživatele díky tomu nejsou kladeny tak vysoké nároky. Jedná se tedy o kompromis programového přístupu a autorského nástroje [17].

Geovisto očekává data od uživatele v objektově orientovaném formátu, která automaticky transformuje na obecný datový model používaný v implementaci. Tento model je uživateli následně prezentován prostřednictvím grafického uživatelského rozhraní. Nástroj umožňuje vizualizovat pouze některé atributy a vytvářet více vrstev, které lze ve výsledném zobrazení kombinovat. Vykreslené vrstvy zachytávají uživatelovu interakci s obsahem a jsou schopny na ni reagovat, je tak například možné se v rámci mapy pohybovat či měnit zobrazované měřítko [17].

Architektura knihovny je založena na principu strukturování do jednotlivých komponent, což umožňuje vývojářům zahrnout do projektu pouze funkcionalitu, kterou opravdu využijí. Navíc tato struktura implementace poskytuje prostor pro budoucí rozšíření.

Nástroj Geovisto nabízí balíčky pro tvorbu několika tematických map: kartogram, mapu spojení a mapu symbolů. V době psaní práce je připravována podpora pro teplotní mapy.

Podrobnější analýze knihovny Geovisto se věnuje kapitola 4.

Kapitola 3

Webové frameworky pro tvorbu UI

Dříve byly webové aplikace realizovány jako množina statických stránek, jejichž obsah se zřídka kdy měnil. Při každé interakci uživatele s webovou stránkou bylo nutné zaslat požadavek na server. Server událost zpracoval a sestavil nové vyobrazení webové stránky, následně vyhotovený soubor zaslal zpět klientovi. Veškeré komponenty shodné pro většinu zobrazení, obvykle zejména navigační panel, menu nebo záhlaví, se pak musely načítat znovu s každou změnou na stránce, a to i v případě bylo-li potřeba změnit pouze jediný element. Uživatel si zmíněného chování mohl všimnout „probliknutím" stránky v prohlížeči. V horším případě pomalého internetového připojení nebo nadměrné velikosti přijatých souborů se na koncovém zařízení webová stránka načítala postupně v rámci sekund, a pokud uživatel chtěl netrpělivě interagovat s doposud vykresleným obsahem, musel vyčkat na dokončení načítání. Uvedený poněkud nesofistikovaný přístup vedl k nepřívětivé uživatelské zkušenosti a pro čím dál náročnějšího koncového uživatele se postupem času stal nepřijatelný [28].

Vyřešit problémy popsané v předchozím odstavci si klade za cíl koncept vytváření jednostránkových aplikací s názvem SPA (zkratka pro anglické *Single Page Application*). Tento koncept se snaží v konzumentovi webové stránky navodit pocit jako by pracoval s nativní aplikací. Generuje HTML dokument dynamicky, a tedy odbourává nutnost obnovení stránky při změně obsahu, nabízí téměř okamžitou odezvu a zvyšuje rychlost celé aplikace. Aby toho však bylo možné docílit, musíme rozšířit portfolio použitých technologií, jednou z možností je klientský JavaScript [28].

Klientský JavaScript (anglicky *client-side JavaScript*) je dynamicky typovaný interpretovaný programovací jazyk spouštěný v prohlížeči klienta, který umožňuje programátorovi definovat celkové chování aplikace [14]. Dle statistik serveru W^{3}Techs k datu psaní této práce ve využitých klientských programovacích jazycích pro tvorbu dynamických webových stránek JavaScript jasně dominuje s téměř 97,6% podílem [41]. V současnosti je tak přirozeně interpret tohoto jazyka již součástí každého moderního prohlížeče [14]. Jádrem klientského JavaScriptu je aplikační rozhraní s jehož využitím lze přistupovat k jednotlivým elementům v objektově orientované reprezentaci HTML dokumentu se stromovou strukturou – odborně nazývané DOM (zkratka pro anglické *Document Object Model*). Umožňuje tak webovým vývojářům přistupovat ke konkrétním elementům HTML dokumentu, měnit jejich obsah, předefinovat vizuální stylování, restrukturalizovat dokument a spouštět různé akce na základě uživatelovy interakce s webovou aplikací [15].

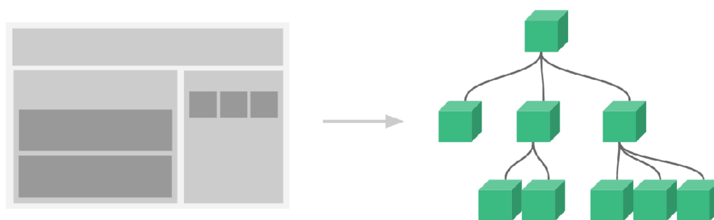
Řešení využít pouze HTML, CSS a *vanilla*[1] JavaScript k vytvoření dynamických webových stránek je naprosto v pořádku, vývojář však může narazit na problémy ve chvíli, kdy

[1]Označení pro prostý JavaScript bez přidaných externích knihoven

aplikaci bude potřeba rozšiřovat a její úroveň komplexity poroste. V tomto přístupu hrozí riziko, že i malá změna může ovlivnit velkou část aplikace, což povede k nutnosti přepsat velké množství kódu. Programátor bude nucen v implementaci manuálně měnit zobrazované hodnoty, neustále přistupovat k objektovému modelu dokumentu a provádět v něm operace čtení a zápisu. Dalším rizikem je kumulující se počet duplicit napříč nově přidávanými JS a CSS soubory, s každou přidanou funkcionalitou tak bude kód náročnější na údržbu a budoucí úpravy. Psaní takového kódu je velice časově náročné, udržet kód čitelný se stává nadlidským úkonem, a pokud by interní fungování aplikace chtěl pochopit nově příchozí kolega, bude velmi složité mu implementaci vysvětlit. Samozřejmě lze i takovýto projekt s dostatkem času a úsilí udržovat, vyvstává ale otázka, zda neexistuje nějaký alternativní výhodnější přístup [30].

Některé ze zmíněných problémů může autor aplikace vyřešit zahrnutím knihovny jQuery do svého projektu. Jedná se JavaScriptovou knihovnu, která klade důraz na snadnější manipulaci s objektovým modelem HTML dokumentu, opět za použití programového přístupu. Knihovna poskytuje abstrakci nad jazykem JavaScipt, díky čemuž může programátor psát kratší a lépe přehledný kód a snadněji manipulovat se stylováním elementů [21]. Stále ale přetrvávají problémy neustálého přistupování ke stromové struktuře dokumentu, manuálního zachytávání událostí, řízení přepisování hodnot, duplicit souborů a další.

Se snahou ulehčit vývoj komplexních aplikací a zamezit tomu, aby programátoři museli v každém projektu „znovuobjevovat kolo“ (implementovat společné mechanismy pro každý z vytvářených projektů) vznikly aplikační frameworky pro tvorbu uživatelských rozhraní. Jsou to předem implementované svazky JavaScriptového kódu, které poskytují základní řešení pro aplikaci v podobě podpůrných mechanismů. Jedním z takových mechanismů je systém pro detekci změn, kdy při aktualizaci dat dojde automaticky i k překreslení dané části uživatelského rozhraní a webový vývojář se tak nemusí starat o manipulaci s objektovým modelem dokumentu. Tyto frameworky staví na konceptu strukturování projektu do jednotlivých komponent, kde komponentou je jistá šablona pro deklarativní popis dokumentu, stylů a definici chování. Zda komponentu představuje jediný soubor, nebo souborů více, už záleží na použitém aplikačním rámci. Popisovaná architektura usnadňuje možnosti změn a dochází k odstínění mezi jednotlivými funkčními celky, vlivem čehož ovlivní změny pouze správnou část aplikace. Výhodou implementace v podobě komponent je jejich modularita a znovupoužitelnost. Díky konceptu zapouzdření lze jednotlivé komponenty do sebe zanořovat a opakovaně je využívat [30].



Obrázek 3.1: Organizace komponent v moderních frameworcích[2]

[2]https://v3.vuejs.org/guide/component-basics.html

Jak lze vidět na obrázku 3.1, základ aplikace tvoří jedna zdrojová komponenta, do které jsou vnořovány další a vytváří tak hierarchii reprezentovanou stromovou strukturou. Úroveň zanoření komponent není zpravidla omezena.

Popularita webových frameworků v posledních letech stále roste, protože v praxi s sebou přináší řadu výhod:

- *Rychlejší vývoj* – díky poskytnutým podpůrným mechanismům se mohou vývojáři soustředit na implementování aplikace a nemusí řešit nízkoúrovňové záležitosti.
- *Spolehlivý a udržovatelný kód* – zdrojový kód implementovaný s využitím webových frameworků dodržuje vhodné koncepty, kterými jsou organizace zdrojového kódu, členění projektu, snadnější testovatelnost a další.
- *Snadnější kolaborace* – webové frameworky spojují podobné architektonické a návrhové principy, takže není pro programátora obtížné se v nich orientovat.
- *Podpora komunity* – v případě problému je pravděpodobné, že stejné potíže měl i někdo jiný a je snadno dohledatelné řešení. S oblíbeností frameworku roste i počet různých doplňků třetích stran.

Samozřejmě využití kódu a principů, které nebyly vytvořeny na míru projektu, může přinést i zápory. Omezující může být dogmatická struktura konkrétního aplikačního rámce, čas nutný k pochopení konceptu rámce a obtíže může způsobit i aktualizace frameworku, která může narušit funkčnost vyvíjené aplikace [30].

Bližšímu popisu webových front-end frameworků pro tvorbu uživatelského rozhraní se věnují následující dvě sekce této kapitoly. Poslední, třetí, sekce popisuje podpůrné nástroje, které vývojářům mohou pomoci vytvářet kvalitnější kód při implementaci komponent.

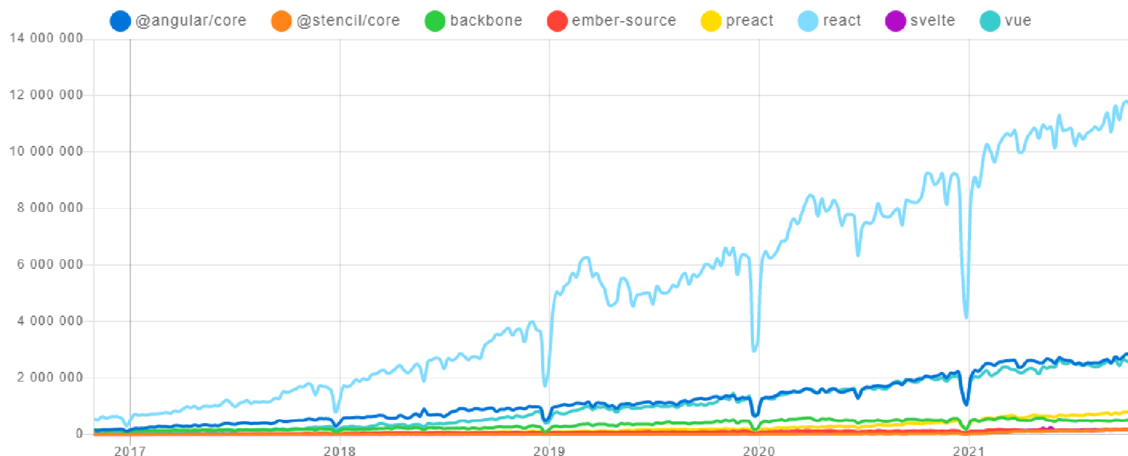
3.1 Porovnání UI frameworků

Vývojář webové aplikace má při rozhodování jaký UI framework použít na výběr ze široké škály možností a s každým dnem přibývají další. Přesto však existuje několik z nich, které jsou dlouhodobě u front-endových vývojářů oblíbené a lze je tak považovat za ty nejpopulárnější. Cílem této sekce je poukázat na rozdíly mezi nejznámějšími aplikačními rámci a z různých pohledů analyzovat faktory, které mohou být při výběru frameworku pro implementaci projektu klíčové. Všechna popisovaná řešení jsou volně dostupná open-source.

Podle popularity

Aplikační rámce jsou balíčky JavaScriptové implementace, které jsou exportovány k programátorům ve formě npm[3] balíčků, na jejichž základě následně vývojáři staví svoji aplikaci. Počty stažení těchto balíčků v závislosti na čase za interval posledních 5 let lze sledovat na následujícím grafu:

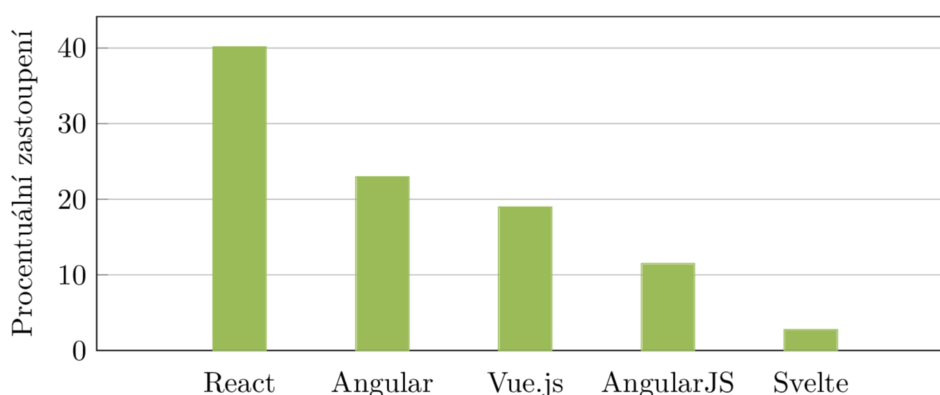
[3] https://www.npmjs.com/



Obrázek 3.2: Trend počtu stažení npm balíčků UI frameworků za dobu 5 let[5]. Zdrojem dat je správce balíčků npm, jehož prostřednictvím zahrnují vývojáři aplikační rámce do svých projektů. Graf vyjadřuje denní počet stažení knihoven v intervalu posledních 5 let.

Z výše uvedeného grafu 3.2 je možné vyčíst, že v současnosti v počtu stažení jasně dominuje framework React s 12 miliony stažení denně. O další dvě pozice soupeří aplikační rámce Angular a Vue.js, přičemž jejich křivka v grafu se v čase téměř prolíná. V době psaní práce byl na druhém místě Angular s téměř 3 miliony staženími denně a třetího místa se ujal framework Vue.js s 2,75 miliony. Žlutá křivka značí počty stažení pro odlehčenou a rychlejší alternativu Reactu – Preact, který se s číslovkou 900 tisíc umístil na čtvrtém místě, o místo níže je poté framework Backbone.js s půl milionem stažení každý den. Vzhledem k téměř lineárně rostoucím křivkám stažení lze pozorovat rostoucí oblibu webových UI frameworků.

Server Stack Overflow[6] provedl v srpnu roku 2021 průzkum popularity, do kterého se zapojilo více než 80 000 vývojářů z celého světa [38]. Jedním z výstupů průzkumu je seznam webových frameworků, se kterými programátor již pracoval a s jakými by se chtěl seznámit v následujícím roce. Na obrázku 3.3 je graficky vyobrazen výsledek ankety v podobě procentuálního zastoupení pro pět prvních zástupců z řad JavaScriptových UI frameworků.

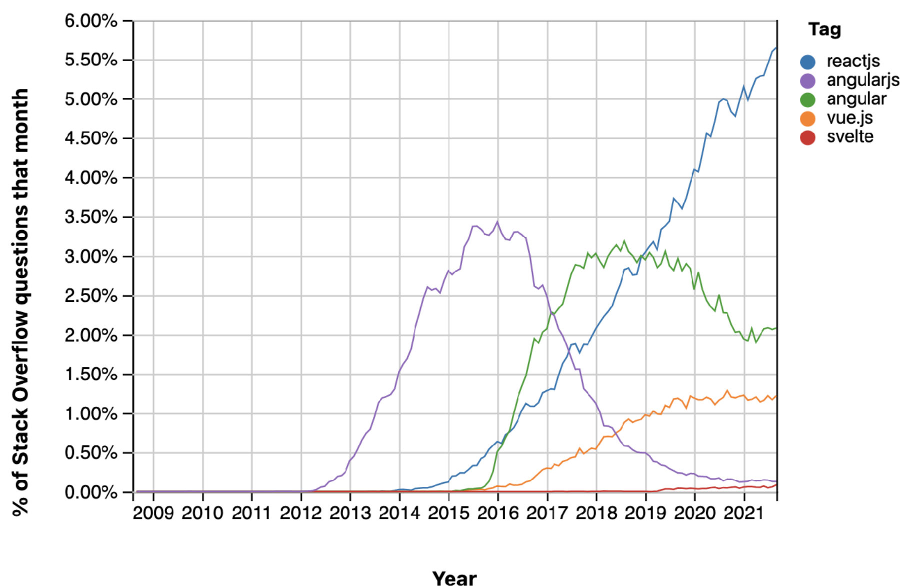


Obrázek 3.3: Popularita frameworků podle průzkumu serveru Stack Overflow[7]

[5]https://www.npmtrends.com/@angular/core-vs-@stencil/core-vs-backbone-vs-ember-source-vs-preact-vs-react-vs-svelte-vs-vue

[6]https://stackoverflow.com/

Výsledky průzkumu v případě prvních tří aplikačních rámců kopírují umístění v grafu 3.2, nejoblíbenějším UI frameworkem mezi programátorskou veřejností se tak stal React s 40,14 % hlasy, dále pak Angular s 22,96% podílem následovaný Vue.js s hodnotou nižší o 4 %. Na čtvrtém místě se umístil předchůdce frameworku Angular – AngularJS se 11,49 % hlasy a pátý skončil front-end překladač Svelte s 2.75% zastoupením. Tento trend potvrzuje i graf 3.4 zobrazující poměr dotazů za uplynulé roky v komunitě na serveru Stack Overflow.



Obrázek 3.4: Poměrné zastoupení dotazů položených na serveru Stack Overflow[8]

V tomto grafu je zajímavé sledovat fialovou křivku patřící webovému frameworku AngularJS, která od roku 2016 výrazně klesá. To je zapříčiněno vydáním zpětně nekompatibilního nástupce s novou koncepcí – rámce Angular. Podpora původního AngularJS byla ukončena k 31. červenci 2021.

Další metrikou, kterou by bylo možné porovnávat oblíbenost aplikačních rámců je počet „hvězdiček“ (tzv. *Github stars*) u repozitáře se zdrojovým kódem frameworku na serveru GitHub. V následujícím seznamu je uvedeno pět frameworků s největším množstvím hvězd u příslušících repozitářů společně s počtem nových změn týdně (hodnoty jsou průměrovány z dat za uplynulých 10 týdnů do data 31. 10. 2021).

- Vue.js[9] – 190 tisíc hvězd (průměrně 1 commit týdně).
- React[10] – 177 tisíc hvězd (průměrně 22 commitů týdně).
- Angular[11] – 77 tisíc hvězd (průměrně 49 commitů týdně).
- AngularJS[12] – 60 tisíc hvězd (podpora ukončena).

[7] https://insights.stackoverflow.com/survey/2021#most-popular-technologies-webframe
[8] https://insights.stackoverflow.com/trends?tags=reactjs%2Cvue.js%2Cangular%2Csvelte%2Cangularjs
[9] https://github.com/vuejs/vue
[10] https://github.com/facebook/react
[11] https://github.com/angular/angular
[12] https://github.com/angular/angular.js

- Svelte[13] – 52 tisíc hvězd (průměrně 7 commitů týdně).

Možná trochu překvapivě se ujímá prvního místa Vue.js. Zatímco za vývojem frameworků React a Angular stojí velké korporátní společnosti, podpora Vue.js do jisté míry stojí na programátorské komunitě, což by mohl být jeden z důvodů oblíbenosti na tomto serveru. Největší provoz změn však s přehledem zaznamenává repozitář Angularu s průměrně téměř půl sty commitů týdně.

Podle výkonnosti

Velké množství vývojářů je při volbě JavaScriptového frameworku pro nový projekt ovlivněno subjektivními faktory. Upřednostňují ty, se kterými již mají předchozí zkušenosti, případně ty, které jsou jim blízké. Jedna z přehlížených, avšak klíčových vlastností při vyhodnocování kvality frameworku, je však jeho výkon, zejména pokud tým plánuje implementovat rozsáhlý komplexní projekt. Uživatele aplikace nezajímá, jaké operace jsou prováděny na pozadí, důležité je, aby nedocházelo k nepřijatelným zpožděním a obsah s veškerými procesy na pozadí byl vyhotoven pokud možno v co nejkratším čase [27].

Jedním ze způsobů jak porovnávat rychlost je vytvoření nástroje, který provede srovnávací testy (takzvaný *benchmark*), kde smyslem je objektivně porovnat reagování testovaných subjektů. Premisou pro objektivní výstup jsou stejné vstupní podmínky. V případě testování webových frameworků se jedná o stejný vstup, použitý prohlížeč, hardware počítače a operační systém [27].

Německý programátor Stefan Krause je autorem nástroje `js-framework-benchmark`[14], který se testováním JavaScriptových UI frameworků zabývá. Benchmark vytváří tabulku s velkým množstvím záznamů a nad tabulkou vzápětí provádí různé operace. Pro jednotlivé aplikační rámce měří čas potřebný pro vykonání daných akcí včetně renderování. Výstupem je tabulka, kde sloupce vymezují testované frameworky, řádky provedené operace. Výsledek konkrétního testu je uveden v příslušné buňce tabulky, přičemž nese hodnotu času trvání operace v milisekundách. Výstup provedeného testování pro nejpopulárnější frameworky je zobrazen na obrázku 3.5. Identifikátor záznamu v podobě atributu `key` byl pro testování povolen.

[13] https://github.com/sveltejs/svelte
[14] https://github.com/krausest/js-framework-benchmark

Název Doba trvání pro..	vanillajs	solid-v1.1.0	inferno-v7.4.8	svelte-v3.42.1	vue-v3.2.1	preact-v10.5.14	angular-v12.0.1	react-v17.0.1	ember-v3.27.3
vytvoření řádků vytvoření 1000 řádků.	81.5 ± 1.2 (1.00)	84.5 ± 0.8 (1.04)	93.0 ± 0.7 (1.14)	96.5 ± 1.0 (1.18)	108.4 ± 1.4 (1.33)	108.0 ± 3.9 (1.32)	127.0 ± 2.3 (1.56)	135.7 ± 3.6 (1.66)	288.2 ± 6.2 (3.53)
nahrazení všech řádků aktualizace všech 1000 řádků.	79.5 ± 0.4 (1.00)	85.7 ± 1.4 (1.08)	85.1 ± 0.8 (1.07)	96.7 ± 2.0 (1.22)	93.1 ± 0.6 (1.17)	102.7 ± 1.0 (1.29)	108.6 ± 1.9 (1.37)	106.6 ± 0.8 (1.34)	173.9 ± 2.8 (2.19)
částečná aktualizace aktualizace každého 10. řádku pro 1000 řádků.	155.0 ± 3.3 (1.00)	160.8 ± 2.9 (1.04)	167.4 ± 3.7 (1.08)	190.4 ± 3.0 (1.23)	205.9 ± 3.9 (1.33)	182.8 ± 4.6 (1.18)	189.7 ± 4.6 (1.22)	223.9 ± 6.1 (1.44)	185.0 ± 3.6 (1.19)
výběr řádku označení vybraného řádku.	18.4 ± 1.0 (1.00)	20.5 ± 0.8 (1.11)	41.0 ± 2.1 (2.23)	29.0 ± 1.0 (1.58)	31.5 ± 0.6 (1.71)	60.8 ± 4.2 (3.31)	75.3 ± 1.7 (4.10)	115.4 ± 1.8 (6.28)	46.0 ± 1.2 (2.50)
výměna řádků výměna 2 řádků pro tabulku s 1000 řádky.	49.8 ± 0.5 (1.00)	50.5 ± 0.5 (1.01)	50.1 ± 0.4 (1.01)	51.9 ± 0.3 (1.04)	51.6 ± 0.7 (1.04)	52.7 ± 0.5 (1.06)	329.1 ± 1.0 (6.61)	328.6 ± 1.2 (6.60)	56.9 ± 1.7 (1.14)
odebrání řádku odebrání jednoho řádku.	20.4 ± 0.2 (1.00)	21.3 ± 0.8 (1.04)	20.5 ± 0.4 (1.01)	21.1 ± 0.3 (1.04)	22.2 ± 0.2 (1.09)	21.6 ± 0.3 (1.06)	20.8 ± 0.5 (1.02)	23.4 ± 0.3 (1.15)	22.5 ± 0.5 (1.10)
vytvoření více řádků vytvoření 10 000 řádků.	773.9 ± 15.7 (1.00)	815.2 ± 3.0 (1.05)	870.4 ± 3.8 (1.12)	984.4 ± 33.3 (1.27)	1,032.9 ± 17.8 (1.33)	1,108.9 ± 23.4 (1.43)	1,131.9 ± 18.3 (1.46)	1,390.4 ± 23.1 (1.80)	1,685.2 ± 27.9 (2.18)
přidání řádků do větší tabulky přidání 1000 řádků do tabulky s 10 000 řádky.	174.3 ± 2.0 (1.00)	182.5 ± 1.8 (1.05)	189.0 ± 1.4 (1.08)	212.3 ± 2.2 (1.22)	201.3 ± 2.7 (1.15)	254.2 ± 4.3 (1.46)	244.3 ± 1.5 (1.40)	255.7 ± 3.4 (1.47)	547.8 ± 5.0 (3.14)
smazání řádků smazání všech řádků z tabulky s 1000 záznamy.	47.8 ± 0.5 (1.00)	54.6 ± 0.8 (1.14)	63.1 ± 0.7 (1.32)	69.8 ± 0.8 (1.46)	69.1 ± 0.9 (1.44)	70.1 ± 0.7 (1.47)	156.0 ± 2.8 (3.26)	82.7 ± 0.8 (1.73)	227.9 ± 1.5 (4.77)
geometrický průměr všech faktorů v tabulce	1.00	1.06	1.19	1.24	1.27	1.42	1.99	2.07	2.14

Obrázek 3.5: Evaluace výkonu JavaScriptových frameworků pro jednotlivé operace[15]

Referenční hodnoty jsou stanoveny jazykem JavaScript bez jakékoliv přidané knihovny. Uvedeny jsou v nejlevějším sloupci a ohodnoceny jsou skóre 1 (hodnota v závorce). Vzhledem k faktu, že mechanismy frameworků jsou implementovány funkcemi Vanilla JavaScriptu, nemohou nikdy dosáhnout lepších hodnot než nativní JavaScript.

Zabarvení buňky značí velikost rozdílu oproti bázové hodnotě, kde s rostoucím rozdílem se buňka zabarvuje více do červena. Každá operace frameworku je ohodnocena skórem, které odpovídá podílu času vůči referenční hodnotě. Frameworky jsou ve sloupcích seřazeny podle geometrického průměru výsledného skóre jednotlivých operací uvedeného v posledním řádku tabulky. Řazení je podle výkonu realizováno zleva doprava, levý soused je tak podle testů celkově rychlejší než vybraný aplikační rámec.

Z obrázku 3.5 vyplývá, že nejrychlejším UI frameworkem se stal SolidJS, jehož výsledky testů se téměř blíží hodnotám prostého JavaScriptu. SolidJS cílí především na výkonnost a efektivitu a od ostatních frameworků se společně se Svelte (umístěném na 3. místě) liší i způsobem, jakým poskytují aplikaci koncovému klientovi. Oba ze zmíněných frameworků

[15]https://krausest.github.io/js-framework-benchmark/2021/table_chrome_95.0.4638.54.html

totiž fungují jako kompilátory, kdy s předstihem v době sestavování (*build time*) kompilují šablony implementované deklarativně do svazku optimalizovaného imperativního JavaScriptového kódu, který přímo aktualizuje DOM. Tento svazek následně distribuují koncovému uživateli [33, 35]. Dle oficiální dokumentace SolidJS za jeho rychlostí stojí především následující trojice vlastností [35]:

1. Sledovány jsou pouze elementy, pro které byla explicitně definována reaktivita.
2. Využití heuristiky ke snížení granularity a tedy snížení počtu provedených výpočtů.
3. Odložené vyhodnocování vlastností (anglicky *Lazy property evaluation*), což umožňuje odebrání nadbytečného obalujícího kódu a přebytečné synchronizace.

SolidJS nabízí téměř shodné funkce jako React, disponuje i velmi podobnou syntaxí (využívá také JSX), přičemž díky kompilování dosahuje vyšších rychlostí. Vzájemná kompatibilita však mezi těmito aplikačními rámci neexistuje, především kvůli odlišnému způsobu renderování komponent [35].

Velmi dobré skóre získala také knihovna Inferno[16], která opět cílí zejména na výkon. Na rozdíl od SolidJS je kompatibilní s Reactem, nepodporuje však jeho veškerou funkcionalitu [9]. Inferno, Vue.js, Preact, i React sdílí jednu společnou vlastnost. Všichni jmenovaní používají koncept virtuální dokumentového modelu k propagaci provedených změn do skutečného DOM. Každý má však vlastní implementaci, jakým způsobem změny zapisuje a porovnává mezi aktuálním skutečným a virtuálním objektovým dokumentovým modelem [9, 23].

Angular k aktualizaci DOM využívá odlišných praktik přepsáním implicitních mechanismů v prohlížeči, tento postup je podrobněji popsán v kapitole 3.2.2.

Podle obtížnosti

Navzdory tomu, že UI frameworky sdílí podobnou myšlenku a základní architekturu, jedná se stále o jiné produkty, které mají svá vlastní specifika. Jejich použití tak může být rozdílně obtížné. Obtížnost a časovou náročnost potřebnou k osvojení praktik definuje křivka učení (z anglického *learning curve*).

Mezi intuitivní a snadné k naučení se řadí framework Svelte, který by neměl být překážkou ani pro vývojáře začínající s jazykem JavaScript. Schéma komponent je ve Svelte soustředěno kolem HTML kódu a rozšiřuje ho o množinu direktiv. Relativně stále snadný na naučení je i Vue.js, který se chlubí především kvalitní dokumentací a snadnou integrací. Čas ušetří zejména vývojáři se zkušenostmi s aplikačními rámci Angular či React, neboť Vue některé části z obou světů přejímá. O kus obtížnějším zástupcem je například zmiňovaný React. Vzhledem k tomu, že React není kompletní framework, ale pokročilejší funkce vyžadují knihovny třetích stran, jeho složitost se liší v závislosti na využitých knihovnách. Obecně ale patří React k frameworkům, které jsou středně složité na naučení [23]. Mezi zástupce se strmou křivkou učení se poté řadí Angular a Ember.js. Hlavním důvodem větší časové náročnosti pro osvojení frameworku Angular je jeho komplexita a množství specifických programovacích technologií [12]. Ember.js disponuje vlastním ekosystémem s velkým množstvím vývojářských nástrojů, naučit se Ember.js by tak znamenalo naučit se používané metody, nástroje, konvence a zapojení do celého ekosystému, což může být pro programátory začínající v této doméně velmi obtížné [37].

Složitosti aplikačních rámců React, Angular a Vue se blíže věnuje kapitola 3.2.

[16] https://github.com/infernojs/inferno

Podle ostatních charakteristik

Některé z frameworků nechávají vývojářům volné ruce při tvoření struktury projektu, jiné jsou ohledně struktury dogmatické a vyžadují jejich dodržování. Vysoká flexibilita organizování kódu s sebou přináší své klady i zápory. Vyšší volnost, jakou poskytuje například Vue.js, dovoluje přizpůsobit lépe projekt jeho účelu, při komplexním projektu a špatně navržené struktuře však hrozí, že kód bude špatně udržovatelný a testovatelný. Striktní strukturu naopak vyžadují rámce Angular či Ember.js [12].

Méně populární z frameworků trpí jednou společnou nevýhodou, která může degradovat jejich potenciál: velikost komunity. Komunitou se rozumí programátorská veřejnost přispívající k prosperitě frameworku nahlašováním chyb, diskutováním a řešením problémů, testováním, rozšiřováním funkcionality za pomoci vytváření balíčků třetích stran a další. U méně populárních aplikačních rámců se vývojáři musí při implementaci většinou spolehat pouze na oficiální dokumentaci a její kvalita je tak pro vývoj klíčová. Dalším důležitým faktorem při rozhodování je i dostupnost knihoven, které vývojáři chtějí při implementaci využít.

Závěrem

I když frameworků existuje nepřeberné množství a každých několik dní přibude další, nelze z nich vybrat jeden nejlepší, který by byl ideálním řešením a bylo by možné ho použít na všechny projekty. Vždy je zapotřebí při výběru zohlednit účel projektu, jeho rozsah, velikost týmu programátorů, jejich zkušenosti a další důležité faktory.

3.2 Podrobná analýza

Tato sekce si klade za cíl představit tři z nejpopulárnějších frameworků, kterými jsou z dlouhodobého hlediska aplikační rámce React, Angular a Vue.js. Jejich dominantní pozici na trhu dokazuje i počet stažení na obrázku 3.2 z předchozí sekce.

3.2.1 React

Dle oficiálních webových stránek je React definován jako JavaScriptová knihovna pro tvorbu uživatelských rozhraní [13]. Na rozdíl od konkurenčního Angularu tak není kategorizován jako framework. Knihovna samotná poskytuje sady funkcí s dobře definovaným rozhraním, pomocí kterého vývojáři definují žádoucí chování aplikace [31]. Jedním z hlavních rozdílů oproti aplikačním rámcům je v poskytovaných mechanismech. Zatímco plnohodnotné frameworky poskytují vše, co by vývojář mohl potřebovat, knihovna React nabízí pouze jádro funkcí pro tvorbu uživatelského rozhraní. Další mechanismy, ku příkladu směrování v rámci aplikace, řeší programátorská komunita. Knihovna vznikla v roce 2011 a jako open-source byla zpřístupněna v roce 2013. Za jejím vývojem stojí společnost Facebook. Pro implementaci svých aplikací ji využívají velké společnosti jako Netflix, Paypal, AirBnB, Uber a další [31, 36].

React je založený na principu rozdělení uživatelského rozhraní do jednotlivých komponent. Aplikace je sestavena ze základních stavebních kamenů, kterými jsou právě komponenty. Ty jsou definovány deklarativně jako HTML elementy se zapouzdřenou funkcionalitou, vlastnostmi a stavem. Komponenty lze opakovaně používat a zanořovat do sebe, jejich seskupením pak lze vytvářet komplexní uživatelská rozhraní. Deklarativní způsob práce s daty činí chování aplikace lépe předvídatelné a pomáhá ke snadnějšímu ladění kódu [13].

Jednou z hlavních výhod Reactu je doporučený formát pro tvorbu komponent JSX. Jedná se o syntaktické rozšíření jazyka JavaScript, které přináší HTML syntaxi do JavaScriptu. V komponentách je tak na příslušných místech možné definovat HTML elementy společně s JavaScriptovým kódem. Syntaxe JSX činí kód mnohem přehlednější, úspornější a snazší na vývoj a údržbu. React striktně netrvá na používání JSX syntaxe a je možné kód implementovat kód v prostém JavaScriptu nebo i TypeScriptu. Variantou je i využít syntaxi JSX obohacenou o výhody TypeScriptu: TSX [13, 31]. Ukázka React komponenty je k náhledu ve výpisu 3.1, přičemž syntaxe JSX je demonstrována konstantou `avatar`.

```
function WelcomeBar(props) {

  const avatar = <img src={props.avatarUrl} />;
  return (
    <div>
        <div className='container'>{avatar}</div>
        <h3>Hello, {props.name}!</h3>
    </div>
  );
}
```

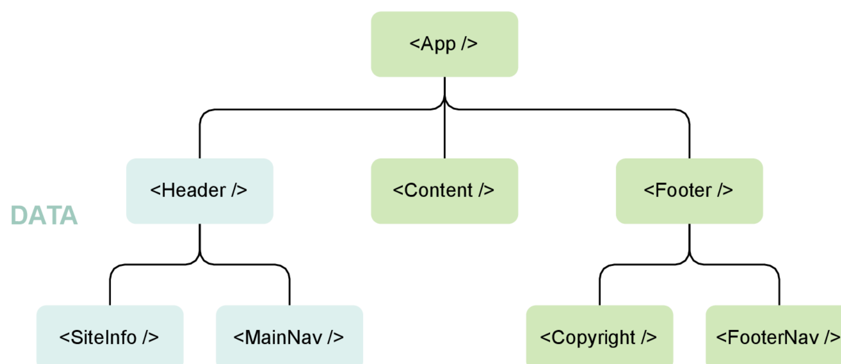
Výpis 3.1: Ukázka React komponenty s použitím syntaxe JSX

Komponenty závisí na datech, které jsou jim zprostředkovány. Ve skutečnosti jsou data tou dynamickou částí uživatelského rozhraní. Na jejich základě se mění obsah webové stránky, ať už je řeč o přímém obsahu elementů nebo podmíněném zobrazení komponent [3].

Tok dat je v Reactu realizován jako takzvaný *one-way binding*, tedy data mohou být propagována pouze jedním směrem. Tento přístup může vyžadovat odlišné vzory programování, na které programátor nemusí být zvyklý. V porovnání s konceptem obousměrného toku dat je zapotřebí psát větší množství kódu [26].

Data jsou propagována stromovou strukturou komponent pomocí tzv. *props*, což není nic jiného než objekt s daty. Ty může komponenta předávat pouze svým přímým potomkům, přičemž tok dat je možné kontrolovat (není nutné je propagovat všem uzlům o úroveň níže). Protože React aplikuje princip jednosměrného toku dat, předávaná data může příjemce pouze číst [13].

Rozsah toku dat je názorně demonstrován na obrázku 3.6.



Obrázek 3.6: Tok dat mezi komponentami v Reactu

Každý uzel na obrázku 3.6 reprezentuje jednu komponentu. Pokud by komponenta `Header` byla držitelem dat, přístup k nim může poskytnou pouze komponentám `SiteInfo` a `MainNav`. Nevýhodou je, že pokud by programátor potřeboval k datům přistoupit v odlišné větvi stromové struktury, bylo by nutné data distribuovat ze společného uzlu, který je oběma nadřazený. V praxi to vede k přístupu, kdy se tento problém vývojáři snaží vyřešit přidáváním dalších komponent, které slouží jako obálky zastřešující kontext dat. Tímto řešením ale narůstá počet komponent, přes které je nutné data propagovat, což má negativní vliv na kvalitu kódu. Uvedený problém se nazývá *props drilling* [34].

Řešení nabídla knihovna třetích stran Redux, která umožňuje jiný postoj k tomu, jak se stavem komponent a propagací dat pracovat. Redux centralizuje stav aplikace do jediného perzistentního skladu, do kterého mají přístup všechny komponenty. Stav aplikace se tak stává přehlednější i pro případy ladění během vývoje [5].

Tohoto nedostatku si byla vědoma i společnost Facebook vyvíjející React, protože s verzí 16.3 v roce 2018 představila novou generaci Context API. Programátoři nově mají možnost vytvoření kontextu přístupného z libovolné komponenty ve stromové struktuře. Samotný koncept je velmi podobný tomu, co nabízí Redux. Kontext udržuje stav a je přístupný pomocí jeho poskytovatele (anglicky *Provider*). Provider je komponenta umožňující inicializovat stav, definovat akce nad stavem a přihlásit se komponentám k odběru změn kontextu. Komponenty, které chtějí přistupovat k tomuto kontextu, musí být do komponenty poskytovatele obaleny. Na závěr je zapotřebí obalit cílovou komponentu, která chce ke stavu přistupovat komponenty konzumenta a injektovat tak kontext. Dále je možné s kontextem manipulovat stejně, jako v případě používání propagace vlastností [13]. Lze tedy přistupovat k datům přímo na úrovni cílové komponenty bez nutnosti je propagovat skrze potomky. Nevýhodou využívání kontextu je překreslení všech komponent, které konkrétní kontext využívají pokaždé, když se nějaká data změní. Proto je jeho použití výhodnější v aplikacích, kde se očekává méně změn stavu [34].

Komponenta v Reactu může být implementačně realizována jako funkce nebo třída. Pokud chtěl vývojář, aby mohl využít funkce životního cyklu komponenty nebo její stav, musel zvolit třídu. Tento přístup však zažil revoluci s *React Hooks* vydanými s verzí 16.8 počátkem roku 2019. React Hooks přidávají komponentě realizované jako funkce možnost pracovat se stavem (`useState`), definovat chování na základě jejího životního cyklu (`useEffect`), přistupovat ke kontextu (`useContext`) a další. Vývojář si navíc může definovat i vlastní akce tohoto typu [13].

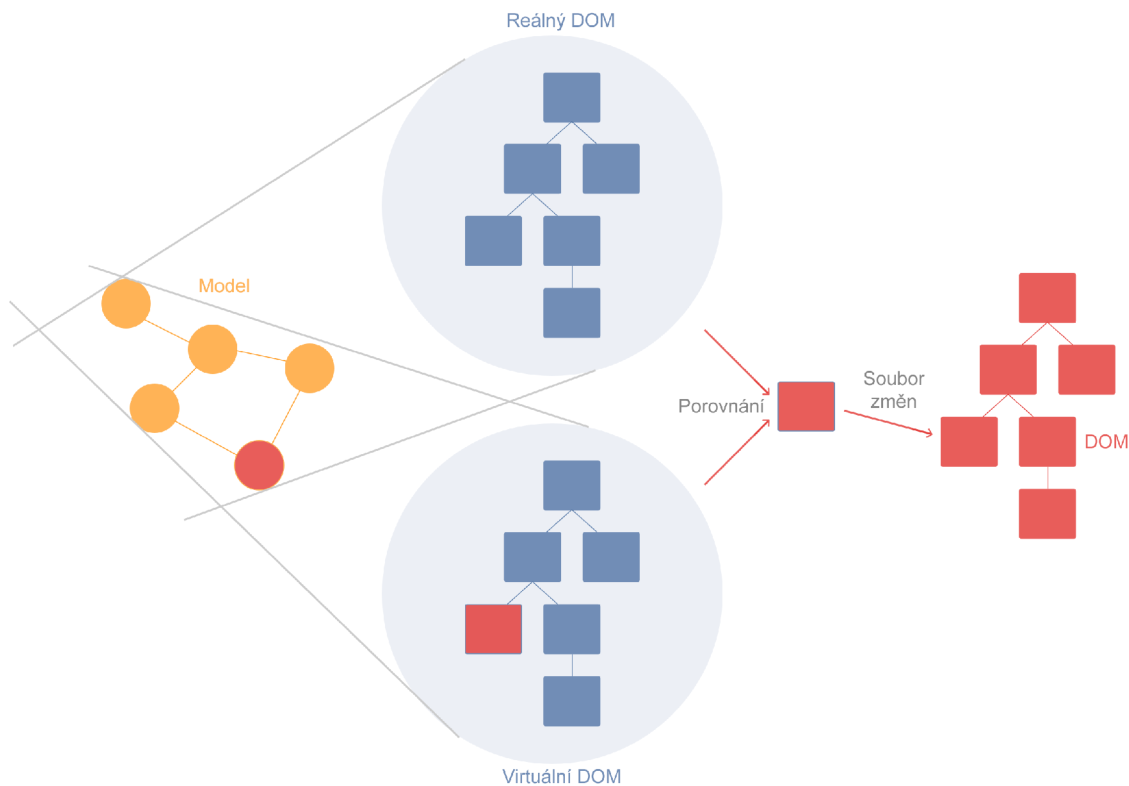
Manipulace přímo s objektovým modelem dokumentu je velmi drahá a výkonnostně náročná. React proto implementuje koncept virtuálního objektového modelu dokumentu (VDOM). Tento přístup umožňuje programátorovi deklarativně určit v jakém stavu chce uživatelské rozhraní mít a React zařídí, aby stav byl přenesen do DOM [13].

Virtuální DOM je stromová struktura React elementů, což jsou nejmenší stavební bloky aplikací používajících tuto knihovnu. Tyto elementy jsou nativní JavaScriptové objekty reprezentující komponenty/HTML elementy. Tedy virtuální DOM se od toho reálného velmi liší, oba však udržují informaci o obsahu, co se má vykreslovat uživateli. Manipulace s virtuálním DOM je mnohem rychlejší, protože React elementy jsou levné a VDOM se nikam nevykresluje. Další optimalizací je seskupování provedených změn do svazků, díky čemuž není nutné při každé zasahovat do reálného DOM a zvyšuje se výkonnost. Pokaždé, když se stav aplikace změní, je aktualizován virtuální DOM namísto toho reálného, cílem ale je udržovat virtuální DOM s tím reálným stále synchronizovaný [13].

React je odstíněn od místa, kam bude komponenty zobrazovat, jeho cílem je pouze vytvořit virtuální DOM. O zobrazení do konkrétního prostředí (web/Android/iOS/...)

se stará příslušná separátní knihovna. Pro zajištění vykreslování obsahu ve webovém prohlížeči slouží knihovna React DOM [3].

Proces synchronizace virtuálního objektového modelu dokumentu s reálným je ilustrován na obrázku 3.7. Diagram znázorňuje zdroj dat pro překreslení (datový model), dvě instance virtuálního DOM sloužící k účelům detekce změn a jejich propagaci do reálného DOM.



Obrázek 3.7: Princip virtuálního DOM v Reactu[17]

React v každou chvíli běhu aplikace udržuje v paměti dvě varianty virtuálního objektového modelu dokumentu. První varianta obsahuje aktuální změny, ta druhá poté předchozí stav. Na výše uvedeném obrázku (3.7) jsou objektové modely dokumentu vyjádřeny stromovou strukturou zvýrazněnou modrým kruhem. V horní polovině obrázku je zobrazen DOM nesoucí informaci o aktuálním zobrazovaném obsahu. V jeho spodní polovině se nachází virtuální DOM s červeně zvýrazněným uzlem reprezentujícím změnu v obsahu dokumentu.

V levé části obrázku 3.7 lze vidět, že některá z komponent v modelu změnila svůj stav (červeně označený uzel). Ve chvíli kdy dojde k aktualizaci stavu, je tato změna promítnuta do virtuálního DOM a ten je porovnáván s předchozím známým stavem. Algoritmus nalezení minimálního počtu modifikací mezi dvěma stromovými strukturami má kubickou složitost $\mathcal{O}(n^3)$, kde n je počet uzlů stromu. Takové porovnání by mělo velký negativní vliv na výkonnost celého procesu. Proto React používá heuristický přístup dosahující složitosti pouze $\mathcal{O}(n)$ založený na dvou předpokladech [13]:

1. Dva elementy odlišného typu budou produkovat rozdílný podstrom.

[17] https://auth0.com/blog/face-off-virtual-dom-vs-incremental-dom-vs-glimmer/

2. Vývojář užitím atributu `key` označí stabilní elementy napříč vykresleními: pokud dochází k renderování opakujících se potomků v DOM uzlu (například seznam `<ul>` s několika vnořenými `<li>` elementy) a bude zapotřebí přidat element na první pozici, dojde znovu k vykreslení celého rodičovského uzlu se všemi potomky. Tato operace je potencionálně drahá a výkonnostně náročná. Problém řeší podpora atributu `key`, který funguje jako identifikátor a React pak může operaci optimalizovat, neboť ví, že stávající elementy se pouze posunuly a není nutné je opětovně vykreslovat.

Výsledkem porovnání je soubor změn, které je nezbytné zanést do reálného objektového modelu dokumentu, aby byl stav aplikace synchronizován s vykreslovaným obsahem. Ve chvíli, kdy je proces propagování změn dokončen, React zajistí překreslení DOM webové aplikace a uživateli se zobrazí aktuální stav.

Jak už bylo zmíněno v předchozí kapitole 3.1, dobou potřebnou pro osvojení si praktik Reactu se knihovna řadí ke středně složitým, přičemž obtížnost velkou mírou závisí na využitých knihovnách třetích stran. React nabízí v rámci úvodního průvodce vytvoření aplikace piškvorek, jež by měla uvést vývojáře do praktik knihovny za dobu přibližně jedné hodiny. Dokumentace dostupná na oficiálních stránkách je podrobná a důkladná. Velkou výhodou je velikost komunity, díky které existuje velké množství veřejně dostupných vláken diskutujících nejrůznější potíže a vývojář začínající s touto knihovnou může velmi rychle nalézt řešení případných problémů [36]. Další výhodou je také mnoho dostupných nástrojů a knihoven třetích stran. Na pozadí tak k úspěchu Reactu přispívá komunita skýtající miliony vývojářů [13].

Společnost Facebook vyvíjející React vydává nové verze nepravidelně, aktuálně[18] je nejnovější verzí verze 17.0.2, vydaná v březnu roku 2021. Nová funkcionalita bývá součástí aktualizací s inkrementovaným číslem minoritního označení, nové majoritní verze poté obsahují změny, které mohou narušit fungování aplikací (tzv. *breaking changes*). V tuto chvíli už vývojáři Reactu připravují verzi majoritní verzi 18 [13].

3.2.2 Angular

Angular je komplexní řešení složené z knihoven pro tvorbu uživatelských rozhraní a vývojářské platformy pro vytváření efektivních a sofistikovaných jednostránkových aplikací. Jako platforma v sobě zahrnuje framework s architekturou založenou na komponentách, kolekci integrovaných knihoven pro podporu směrování v rámci aplikace (tzv. *routing*), správu formulářů, komunikaci mezi klientem a serverem a další. Nabízí také vývojářské nástroje, které usnadňují vývoj, sestavení kódu, jeho ladění a testování [16].

Vydán byl v roce 2016 a jedná se již o druhou generaci (nikoliv verzi) této platformy. Jeho předchůdcem byl AngularJS, který však stavěl na odlišných principech a dokonce i programovacím jazyce. Nová generace frameworku vytvořená od základů byla k odlišení označována jako Angular 2, ale vzhledem k sémantickému značení nových verzí bylo od tohoto označení upuštěno a nyní je znám jako pouze Angular. Mezi těmito aplikačními rámci neexistuje vzájemná kompatibilita. Za jejím vydáním stojí tým vývojářů společnosti Google, který i nadále pracuje na jeho podpoře [31].

Platforma staví na jazyce TypeScript, který lze chápat jako nadstavbu jazyka JavaScript s rozšířenou syntaxí, každý kód napsaný v JavaScriptu je tedy validní i v TypeScriptu. Staticky typovaný TypeScript s sebou přináší kontrolu typů, takový kód je poté bezpečnější, snadnější na údržbu a ladění během vývoje. Specifikovat lze také typy vstupních parametrů

[18]K datu 30. 10. 2021

funkcí a typ její návratové hodnoty. Vývojáři se navíc o kolizi datových typů dozvědí dříve již při překladu, a ne až za běhu aplikace. TypeScript je tak mnohem striktnější, ale pro vývoj bezpečnější volbou. Zdrojový kód napsaný v TypeScriptu není možné v současnosti spouštět přímo ve webových prohlížečích a je nutné ho přeložit do JavaScriptu. K tomu je využíván takzvaný transpiler, což je druh kompilátoru, který čte kód napsaný v jednom jazyce a produkuje kód v jazyce jiném. Oba kódy mají přitom shodnou funkcionalitu. Proces transpilace je ale oproti ostatním frameworkům krok navíc a tak vyžaduje v určité části vývoje při načítání strojový čas navíc [23].

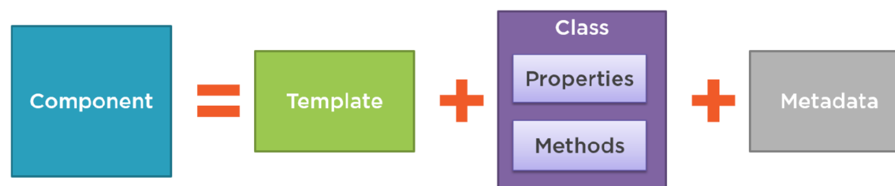
Vývojáři mají k dispozici nástroj příkazové řádky Angular CLI nabízející velké množství příkazů, které jim pomáhají udržovat a testovat kód dodržováním doporučených praktik. Nástroj umožňuje vygenerovat kostru projektu s již nakonfigurovaným směrováním, testovacím frameworkem a preprocesorem stylů. Užitím CLI lze dále generovat komponenty, vytvořit sestavení připravené pro produkci, přidat renderování na straně serveru a mnoho dalšího [16, 29].

Angular je vcelku dogmatický v ohledu jak organizovat projekt, všechny projekty tak mají velmi podobnou strukturu. Tento přístup snižuje míru flexibility, vývojáři ale čelí mnohem méně rozhodování než v případě Reactu. Hlavní výhodou striktní struktury implementovaných projektů je jejich dobrá škálovatelnost, což z Angularu činí dobrou volbu pro tvorbu uživatelských rozhraní rozsáhlých komplexních aplikací.

Základy aplikace vytvořené v Angularu tvoří komponentová architektura společně se službami. Komponenty jsou implementovány jako třídy, z čehož vyplývá, že Angular využívá základních principů objektově orientovaného programování. Komponenty jsou opět skládány do hierarchické struktury a vytvářejí komplexní uživatelské rozhraní.

Každá komponenta sestává ze 3 částí (viz. obrázek 3.8 a demonstrační ukázka zdrojového kódu 3.2) [16]:

- *Šablona* (Template) – šablony kombinují HTML kód rozšířený o direktiva Angularu a mohou tak modifikovat HTML elementy před tím než jsou zobrazeny. V šablonách je možné definovat direktiva poskytující programovou logiku a značky pro provázání s daty.
- *Třída* (Class) – třída je základem komponenty, každá komponenta implementuje třídu, která může obsahovat vlastnosti a metody.
- *Metadata* – definovány dekorátorem, který disponuje informacemi o navázané třídě, zejména nese informaci o příslušící šabloně a stylech. Dekorátor je v programu identifikován prefixem `@`.



Obrázek 3.8: Struktura komponenty[19]

[19]https://learnfrenzy.com/blog/top-25-angular-2-interview-questions–answers

Podle doporučených postupů je vhodné každou část implementovat do odděleného souboru, jak je demonstrováno na příkladu 3.2, je ale možné i zacílit logiku, šablonu i styly do jediného souboru.

welcome-bar.component.ts

```
@Component({
  selector: 'welcome-bar',
  templateUrl: './welcome-bar.component.html',
  styleUrls: ['./welcome-bar.component.css']
})
export class WelcomeBarComponent {
  name = 'John';
  avatarUrl = 'https://pathtoimage.com/px5ppBp.png'
}
```

welcome-bar.component.html

```
<div>
  <div class="container"><img src={{avatarUrl}} /></div>
  <h3>Hello, {{name}}!</h3>
</div>
```

welcome-bar.component.css

```
.container {
  width: 400px;
  height: 200px;
}
```

Výpis 3.2: Ukázka komponenty ve frameworku Angular

Služby (anglicky *services*), které existují po boku komponent poté představují aplikační logiku webu. Jejich nejčastější úlohou je komunikace a distribuce dat mezi klientskou aplikací a serverovým API. Angular implementuje návrhový vzor vkládání závislostí (anglicky *Dependency injection*), který umožňuje jejich funkcionalitu poskytovat dalším třídám. Služba je přidána do kontejneru, pokud je pro ni definovaný dekorátor `@Injectable`. V takovém případě lze konkrétní závislost (v tomto případě službu) vkládat do komponenty přidáním služby do jejího konstruktoru [16].

Logickým seskupením komponent vznikají tzv. moduly. Angular zajistí provázání komponent s využitými službami a vytvoří funkční jednotky. Pro tuto konfiguraci následně zajistí kompilační kontext [16].

Angular podporuje *two-way binding* a tedy kromě dat zobrazovaných aplikací uživateli je schopen změny v DOM způsobené interakcí uživatele reflektovat do dat aplikace. V typické aplikaci tedy existují dva způsoby provázání dat [16]:

- *Provázání na události* – umožňuje aplikaci reagovat na uživatelské vstupy a aktualizovat tak data aplikace.
- *Provázání na vlastnosti modelu* – dovoluje požadované hodnoty z výpočtů nad aplikačními daty projektovat do výsledného HTML dokumentu.

Konfigurace propojení dat je realizována v šabloně komponenty užitím direktiv. Ve chvíli, kdy jsou definována provázání nad daty, je zapotřebí změny detekovat a reflektovat je ve stavu aplikace či do zobrazovaného objektového modelu dokumentu (reálný DOM). K tomu Angular používá jiné mechanismy než jeho konkurence. Nepoužívá tedy koncept virtuálního DOM, namísto toho v Angularu fungují takzvané zóny [16].

Zóny implementované knihovnou *Zone.js* nejsou nic jiného než kontext, v jakém jsou vykonávány asynchronní operace. Ve chvíli, kdy se vyskytuje ve zdrojovém kódu asynchronní operace, je uložena do fronty událostí a čeká na vybrání smyčkou událostí a následné vykonání. Nelze zaručit, kdy se operace dokončí a je problematické na její dokončení následně reagovat. Smyslem zóny je tento kus kódu „obalit“, řízeně ho vykonávat a mít přehled přesně o tom, kdy jeho vykonávání začalo a skončilo. Nad životním cyklem asynchronní operace zóna navíc implementuje zpětná volání (tzv. *callback*), kterým může programátor přiřadit vlastní chování [32]:

- `onZoneCreated` – spouští kód ve chvíli, kdy je rozdvojením vytvořena zóna.
- `beforeTask` – spouští kód před vykonáním samotného obsahu zóny.
- `afterTask` – spouští kód po vykonání obsahu zóny.
- `onError` – provede definovanou operaci v případě, že při vykonávání obsahu zóny dojde k chybě.

Realizace popsaného chování knihovna docílila přepsáním implicitních mechanismů prohlížeče (JavaScriptových API), které naslouchají vzniklým událostem. Pokud je knihovna *Zone.js* v projektu zahrnuta, aplikace má přístup k objektu globální zóny a téměř pro každou asynchronní operaci automaticky vytváří novou zónu rozdvojením té původní. Popsaný mechanismus však není výhradou frameworku Angular, ve skutečnosti byl tento princip převzat z programovacího jazyka Dart. Angular v implementované aplikaci sleduje následující tři operace [16]:

- *Události* – události vyvolané uživatelskou interakcí, například `click`, `change`, `input`, `submit` a další.
- *HTTP požadavky* – získávání dat ze vzdálené služby.
- *Časovače* – JavaScriptové funkce `setTimeout()`, `setInterval()`.

Všechny tři zmíněné body mají jednu společnou vlastnost – jsou asynchronní. Ve skutečnosti jsou tyto operace jediným případem, kdy dochází ke změně stavu aplikace. Angular tedy používá zóny k detekci změn dat a ví tak přesně, kdy je vykonávaný asynchronní kód

dokončen a je nutné aktualizovat DOM. Nicméně zóny ve své původní implementaci nevyhovovaly požadavkům frameworku, a tak Angular zavádí vlastní upravenou verzi zvanou `NgZone`. Jedná se o kopii původní globální zóny s rozšířením jejího API o přidanou funkcionalitu. Důvody, proč Angular definuje svá vlastní zpětná volání, jsou zejména využití konceptu *observables* a sledování časovačů [16].

Angular má strmou učící křivku, což znamená, že v porovnání s ostatními frameworky je časově náročný na osvojení. Důvodem je zejména obsáhlost Angularu, který poskytuje kompletní monolitické řešení, a tak vývojář musí porozumět všem konceptům, co framework přináší. Rovněž jazyk TypeScript, ačkoliv je dobrou volbou pro velké týmy, může být náročnější pro začátečníky, proto učící složitost lehce zvyšuje [23]. Nabízí výukový program Hall of Heroes, který demonstruje stejnou projektovou strukturu shodnou i s projekty v produkčním prostředí. Absolvovat tento kurz a porozumět konceptu Angularu však trvá v průměru několik hodin, což potvrzuje vyšší složitost než v případě Reactu [36].

Společnost Google odpovědná za vývoj Angularu se k vydávání nových verzí staví odlišně než React a Vue.js. Každých 6 měsíců představí novou majoritní verzi zahrnující velké změny, přičemž v tomto intervalu společnost uvolní 1-3 minoritní verze obsahující menší změny. Záplatové aktualizace vycházejí téměř každý týden. Nejaktuálnější verze nese označení 13.0.1 [16].

3.2.3 Vue.js

Vue je progresivní webový framework vytvořený v roce 2013 bývalým zaměstnancem Googlu jménem Evan You [12]. Na rozdíl od ostatních monolitických frameworků bylo Vue od počátku navrženo s cílem být inkrementálně adaptovatelným a škálovatelným řešením. Knihovna, která tvoří jádro aplikačního rámce Vue se zaměřuje pouze na prezentační vrstvu a lze ji tak zahrnout do existujícího projektu realizovaného v jiné technologii a využít pouze v jedné části aplikace, která potřebuje bohatější, více interaktivní chování. Zároveň je framework perfektně způsobilý fungovat v kombinaci s ostatními moderními nástroji a podpůrnými knihovnami jako kompletní řešení schopné pohánět sofistikované jednostránkové aplikace [12].

Vzhledem k topologii frameworku není řešení směrování, správy stavu, renderování na straně serveru a podobných mechanismů implicitně zahrnuto. Dostupné jsou ale balíčky s oficiální podporou (autory jsou vývojáři Vue), které tyto koncepty implementují a lze je do projektu zahrnout. Pro směrování v rámci implementované aplikace existuje balíček *vue-router*, pro správu stavu – *vuex* a za účely jednotkové testování Vue komponent nástroj *vue-test-utils*. Všechny ze zmíněných knihoven mají oporu v dokumentaci. Další funkcionalitu do projektu lze přidat díky obrovskému ekosystému balíčků od vývojářů třetích stran tvořící komunitu Vue [25].

Podobně jako v případě Angularu i Vue nabízí nástroj příkazové řády s názvem Vue CLI. Projekty vygenerované pomocí tohoto nástroje obsahují elementární kostru projektu, doporučené strukturování souborů a prvotní konfiguraci. V možnostech tohoto nástroje je také spravovat sestavení zdrojového kódu, jeho testování a nasazení aplikace. V porovnání s Angularem je škála příkazů nižší, a tak je možností, které má vývojář k dispozici užíváním tohoto nástroje, o poznání méně. Vue CLI například neumožňuje generovat komponenty. Jako jediný z trojice nejrozšířenějších frameworků však Vue nabízí i grafické uživatelské rozhraní, ve kterém si programátor může vyzkoušet jaké závislosti v projektu potřebuje a projekt nakonfigurovat. Spuštění grafického uživatelského prostředí je realizováno použitím příkazu `vue ui` [31, 36].

Obdobně jako u ostatních zmíněných frameworků i Vue sází na dělení struktury projektu do jednotlivých znovupoužitelných komponent. V případě Vue jsou komponenty striktně omezeny vždy pouze na jediný soubor. Ten totiž nemá obecný formát Javascriptu nebo Typescriptu, ale sází na vlastní speciální formát souborů nazývaný Vue SFC používající speciální syntaxi se zaměřením na HTML specifickou pro Vue. Název SFC je zkratkou z anglického *Single File Component* a takový soubor nese příponu `.vue`. Implementace jedné komponenty pak odpovídá jednomu souboru s touto příponou. Aby mohl webový prohlížeč kód zpracovat, musí být implementace předkompilována do standardního JavaScriptu a stylů CSS [12].

Struktura komponenty je rozdělena do 3 logických bloků [12]:

- sekce `<script>` – standardní JavaScriptový modul, místo pro implementaci logiky komponenty. Úlohou modulu je exportovat definici Vue komponenty.
- sekce `<template>` – zde programátor definuje šablonu v HTML s užitím Vue direktiv. Direktiva mají prefix `v-`, jsou vkládány přímo do značek HTML a umožňují vytvořit provázání na model, iterovat kolekcí komponent, přiřadit akcím události a další.
- sekce `<style>` – obsahuje definici stylů pro danou komponentu.

Jedna z vlastností Vue je jeho reaktivita. Pokaždé, když se změní hodnota modelu, dojde k synchronizaci a automaticky je překresleno i uživatelské rozhraní. Vzhledem k tomu, že Vue se řadí mezi frameworky implementující nad modely obousměrné provázání (*Two-way binding*), i změny z uživatelského rozhraní se okamžitě reflektují do stavu aplikace. Aby bylo možné změny promítat, musí v aplikačním rámci existovat mechanismus pro jejich zachycení [12].

Způsob detekce změn se liší pro jednotlivé verze frameworku. Vue 2 převádělo všechny vlastnosti objektu data reprezentující stav aplikace na funkce getter/setter. Pro každou instanci komponenty je definován vlastní hlídač (anglicky *watcher*), který sleduje změny závislostí. Ve verzi 3 prošel systém reaktivity kompletní rekonstrukcí a volí odlišný přístup. Využívá přitom novinek uvedených ve standardu ECMAScript 6, Proxy a Reflect API [12].

Každá komponenta implementuje funkci `data`, která vrací prostý JavaScriptový objekt s daty dané komponenty. `Proxy` je objekt, který obaluje jiný objekt a umožňuje zachytit jakékoli interakce nad ním provedené. Umožňuje tak definovat speciální funkce, které reagují na konkrétní operace a potlačují vestavěné chování objektů JavaScriptu. V případě Vue jsou pro objekt vrácený funkcí data předefinovány obslužné rutiny pro funkce `get` a `set`. Jsou tak zachycena veškerá volání, kdy má dojít ke čtení či zápisu dat a lze nad nimi implementovat vlastní logiku, což je přesně to, co udělali vývojáři Vue. Pokud se přistupuje k objektu za účelem čtení (metoda `get`), rozšířili implementaci o funkci `track`, která zaznamenává závislosti na danou vlastnost. Operace `set` je obohacena o funkci `trigger`, jejíž úkolem je spustit všechny funkce, které měly na konkrétní vlastnosti nastavenou závislost. Díky tomuto aplikační rámec ví, kdy má spustit překreslení zobrazovaného obsahu [12].

V upraveném chování je ale stále žádoucí získat a nastavit hodnotu objektu, tedy je zapotřebí stále vyvolat implicitní metody pro přístup k objektu, které ovšem byly v předchozím kroku byly předefinovány. Problémem je reference klíčového slova `this` momentálně odkazujícího na objekt Proxy, nikoliv na „obalený“ objekt. Řešením je využití rozhraní Reflect API, rovněž představeného v ES6, které snadno provede volání vestavěného chování JavaScriptu [12].

Pro propagaci změn ke konzumentovi webové aplikace využívá podobně jako React (a další řada framewroků) konceptu virtuálního objektového modelu dokumentu (VDOM),

který byl již popsán v sekci 3.2.1. Hledání upraveného uzlu v reálném DOM je drahé, a proto i Vue seskupuje volání na změnu DOM do svazků a změny provádí najednou. Virtuální DOM je přitom upraven pokaždé, když je zavolána v komponentě funkce pro její renderování [12].

Pokud by chtěl vývojář vybrat ze tří nejpopulárnějších frameworků ten nejsnazší na naučení, bude to bezesporu Vue.js. Pomáhá mu v tom především snadná integrace a jeho schéma, kdy uživateli stačí pouze počáteční HTML soubor, který bude doplňovat o funkcionalitu Vue a nemusí tak kód přepisovat například do JSX syntaxe, jak tomu je v případě Reactu. Dalším styčným bodem je kvalitní, přehledná a dostatečně podrobná dokumentace. Vue sdílí některé principy s Reactem i Angularem, a proto pro vývojáře, kteří již mají předchozí zkušenosti s jedním z uvedených dvou frameworků, je velmi snadné se v něm zorientovat. Další výhodou je uživatelská zkušenost z vytváření a správy aplikace, kdy Vue uživatelům za tímto účelem poskytuje mimo konzolového nástroje i grafické uživatelské rozhraní [36].

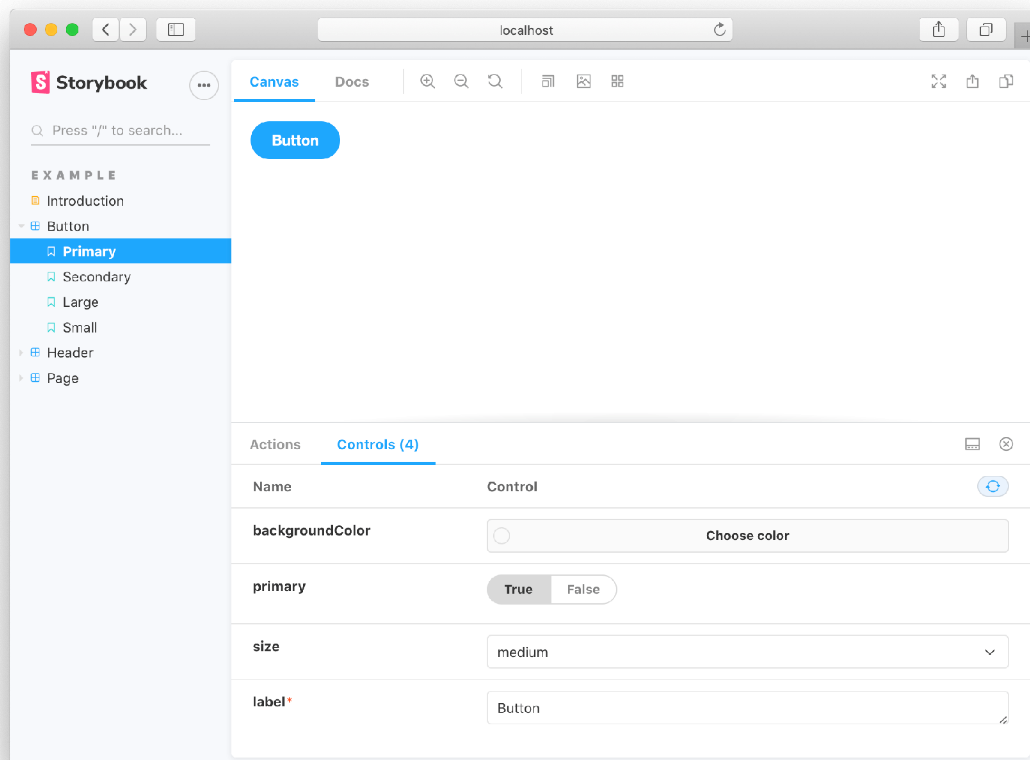
Vývojáři Vue.js vydávají nové verze nepravidelně. Od samého počátku vznikly pouze tři majoritní verze a tou poslední je Vue 3, vydané 18. září 2020. Poslední vydání Vue 3 nese označení 3.2.21 [12]. Mezi webové aplikace implementované platformou Vue.js se řadí weby Alibaba, GitLab, Nintendo a další [36].

3.3 Podpůrné nástroje pro vývoj komponent

Jak již bylo zmíněno v úvodu této kapitoly, většina moderních webových frameworků pro tvorbu uživatelských rozhraní staví na konceptu strukturování zobrazení aplikace do jednotlivých komponent. Ve složité aplikaci může existovat velké množství komponent a jejich zobrazení může mít více podob na základě kontextu, ve kterém se vyskytují. Tato sekce si klade za cíl představit některé z podpůrných nástrojů, které programátorům usnadní jejich vývoj.

3.3.1 Storybook

Prvním, a zároveň v současnosti jedním z nejpopulárnějších podpůrných nástrojů při vývoji komponent ve webovém frameworku, je JavaScriptový nástroj Storybook. V komplexních projektech může existovat velké množství komponent a jejich pozice a výsledné grafické zobrazení na webové stránce se může lišit v závislosti na aktuálním stavu. Udržet přehled o tom, v jakých variacích se bude komponenta zobrazovat není snadným úkolem. Zde přichází na scénu Storybook, který umožňuje vizuálně testovat komponenty pro jejich rozdílné konfigurace. Ať už se jedná o malé atomické komponenty nebo komplexní kusy aplikace, pokud je subjektem část uživatelského rozhraní, lze ho v grafickém prostředí Storybooku promítnout a testovat [39]. Grafické uživatelské rozhraní nástroje pro demonstrační komponentu tlačítka je ukázáno na obrázku 3.9.



Obrázek 3.9: Ukázka grafického uživatelského rozhraní nástroje Storybook[21]. Levý panel zobrazuje zahrnuté komponenty včetně implementovaných rozdílných konfigurací, hlavní část obrazovky je rozdělena do dvou částí. Horní polovina vykresluje výsledné zobrazení komponenty v závislosti na jejích parametrech, zmíněné parametry je poté možné měnit v konfiguračním panelu ve spodní polovině okna prohlížeče.

Vývojář pro každou komponentu, kterou chce vyvíjet/testovat, musí vytvořit takzvanou *story*. Story zachycuje jednu UI komponentu v konkrétním, předem definovaném, stavu. Obvykle je zapotřebí zobrazit komponentu v různém kontextu, a tak pro každou komponentu vznikne několik takovýchto story, kde každá definuje nějaký jiný, pro vývoj zajímavý, stav. Tyto konfigurace a zahrnutí komponenty do testování definuje vývojář ve zdrojovém kódu, samotné zobrazení a testování jednotlivých variací je pak přirozeně dostupné v grafickém uživatelském rozhraní, jež Storybook nabízí. Má však možnost v rámci testování měnit hodnoty parametrů komponent, v čemž spočívá hlavní síla Storybooku. Související komponenty je možné pro lepší organizaci soustředit do skupin, které je možné i zanořovat a vytvářet tak jejich hierarchii [39].

Mimo vizuálního testování komponent Storybook nabízí možnosti jejich dokumentace. Rozšířit jeho funkcionalitu jde navíc některým z celého ekosystému doplňků, ty potom umožňují například zhodnotit jejich přístupnost či ladit jejich responzivní rozložení v kontejneru. Storybook v současnosti podporuje každý z majoritních webových frameworků jako jsou React, Vue, Angular, Svelte, Ember a mnoho dalších. Tento nástroj používají při vý-

[21] https://storybook.js.org/docs/react/get-started/whats-a-story

voji i korporátní společnosti jako Microsoft, Atlassian, Airbnb a další, což svědčí o jeho užitečnosti [39].

3.3.2 Styleguidist

Nástroj Styleguidist se snaží vyřešit problém, kdy při realizaci velkých projektů nemá celý vývojový tým stejné povědomí o již vytvořených komponentách. Aby nevznikal zbytečně duplicitní kód a byly efektivně využívané již implementované komponenty, Styleguidist generuje centralizovanou dokumentaci vytvořených komponent dostupnou celému týmu [1].

Hlavní myšlenkou tohoto nástroje bylo zvýšit napříč projektem adopci UI vzorů a lépe tak škálovat uživatelské rozhraní. Za zrodem nástroje Styleguidist stála myšlenka větší motivace znovupoužití již vytvořených vzorů (což šetří peníze při vývoji a zvyšuje konzistenci), pokud jsou informace dobře zdokumentované a centralizované [10].

Informace pro generování jsou čerpány z komentářů ve zdrojovém kódu, deklarací vlastností a metod nebo souborů ve formátu značkovacího jazyka Markdown (soubory `Readme.md` nebo soubory nesoucí název shodný s názvem komponenty). Na základě použitých značek jazyka Markdown poté vykresluje v dokumentaci interaktivní plochu pro práci s komponentami nebo příslušný formátovaný text [1].

Ačkoliv se s nástrojem Storybook překrývají v některých dostupných funkcích, oba mají rozdílnou perspektivu na vývoj UI komponent. Storybook se soustředí na vytváření a testování komponent, Styleguidist pomáhá zejména s jejich dokumentováním [10]. Achillovou patou nástroje může být jeho podpora, neboť je dostupný pouze pro frameworky React a Vue, aplikační rámec Angular není podporovaný.

3.3.3 Bit

Ušetřit čas vývojářům použitím již implementovaných komponent z jiných projektů bez nutnosti duplicity kódu se snaží platforma Bit. Jedná se o kolaborační platformu, která poskytuje ekosystém pro sdílení komponent mezi aplikacemi napříč repozitáři. Svým způsobem je možné ho chápat jako jakýsi „obchod s komponentami“, což vývojářům znatelně šetří čas, protože nemusí všechny komponenty implementovat od začátku [2].

Bit přidává sémantickou vrstvu nad repozitáře, které mapují soubory na komponenty. Tato přidaná vrstva poskytuje platformě možnost učinit komponentu znovupoužitelnou napříč projekty. Bit komponenta je poté označení pro jednu znovupoužitelnou jednotku. Můžou jí být komponenty některého ze tří nejpopulárnějších frameworků, sdílené CSS/SCSS styly nebo nějaká z užitečných funkcí aplikace [2].

Namísto toho, aby vývojáři vytvářeli komponentu jako součást aplikace, importují už některou z existujících, kterou jim Bit poskytuje. Bit komponenty procházejí aktualizacemi, vývojář využívající některou ze sdílených komponent potom není vázán na nejnovější verzi komponenty, ale v případě, že například obsahuje chybu, může použít některou ze starších verzí. Do projektu jsou naimportovány pouze komponenty, které skutečně ve zdrojovém kódu využívá a nezvyšuje tak velikost své aplikace kompletním balíčkem [2].

Bit je kompatibilní se všemi z popsané trojice frameworků z předchozí sekce: React, Angular i Vue.

Kapitola 4

Analýza

Tato kapitola se věnuje bližšímu představení knihovny Geovisto a podrobněji zkoumá její rozdíly oproti konkurenčním nástrojům. Dále jsou popisovány principy použité v implementaci a jejich vliv na použitelnost nástroje. Na závěr kapitola pojednává o nedostatcích této knihovny a definuje požadavky programátorů na použití knihovny se současnými webovými frameworky pro tvorbu uživatelských rozhraní.

V současnosti je možné dělit nástroje pro interaktivní vizualizaci do dvou různých skupin v závislosti na využitém přístupu při vytváření. První skupinou jsou programátorské nástroje, které pro realizaci vizualizace kladou vysoké nároky na znalosti uživatele. K jejich použití musí mít uživatel programátorské, někdy i matematické, schopnosti. Toto kritérium tak vymezuje cílovou skupinu pouze na vývojáře, a pro uživatele z řad laické veřejnosti je velmi obtížné tyto nástroje využít. Jejich volba tak může spadat na nějaký z nástrojů, které volí vcelku opačný přístup. Nástroje tvořící druhou skupinu nabízí paletu šablon, ze kterých si uživatel může vybrat. Poskytne tak pouze data a má možnost konfigurovat základní parametry zobrazení. Jedná se o takzvané autorské nástroje, tedy grafické nástroje umožňující vytvářet interaktivní vizualizace s možností konfigurace výsledného výstupu. Tento výstup je možné exportovat jako webové komponenty a umožňuje tak v aplikaci zahrnout interaktivní obsah bez nutnosti znalosti programování.

Kompromis mezi uvedenými skupinami hledá knihovna Geovisto[1], která vznikla v roce 2020 se snahou nabídnout nový přístup k realizaci geovizualizace v moderních aplikacích. Nástroj je realizován ve formě programové knihovny a nabízí odlišná rozhraní pro laické uživatele i zkušené vývojáře. Umožňuje přitom nastavovat data a vlastnosti zobrazení deklarativním přístupem. Cílí však zejména na programátory, kteří chtějí rychle prototypovat tématické mapy deklarativně bez nutnosti implementovat mechanismy pro zpracování dat. Mohou k tomuto účelu použít předpřipravené tématické mapy. Knihovna je dostupná pod licencí MIT jako open-source a tím pádem má programátorská veřejnost přístup ke zdrojovým kódům. Je tak možné ji volně užívat, šířit a modifikovat.

4.1 Charakteristiky knihovny Geovisto

V porovnání s ostatními volně dostupnými autorskými nástroji Geovisto řeší odlišně některé vlastnosti klíčové pro vizualizaci geografických tematických map.

První odlišností je formát vstupních dat, který knihovna očekává. Konkurenční nástroje obvykle pracují s tabulkovými formáty, jakými je například formát CSV. V současnosti jsou

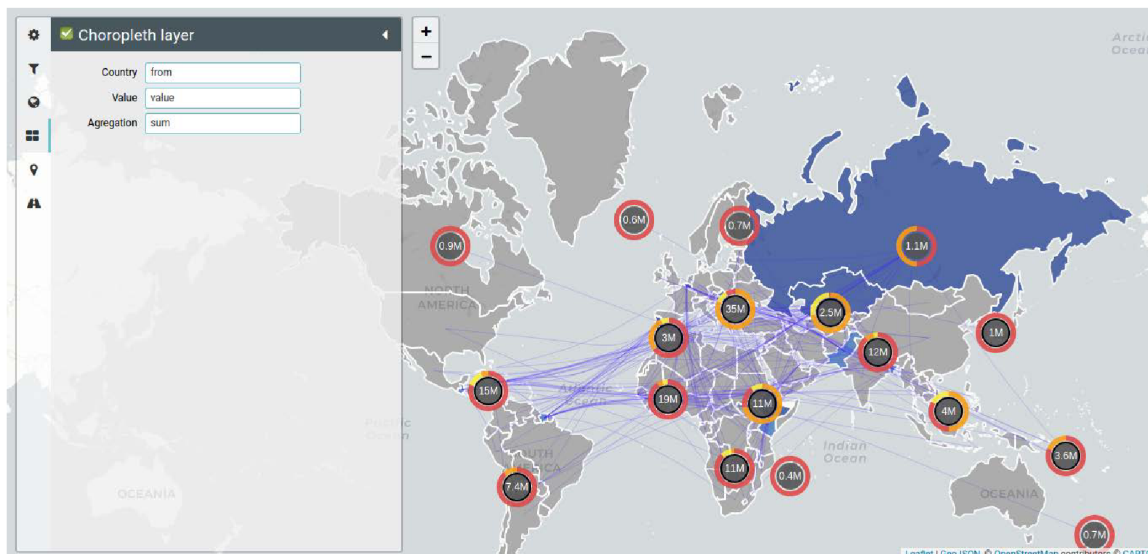
[1] https://github.com/geovisto

užívané datové sady reprezentovány v hierarchických formátech (např. JSON) nebo jsou ukládány v NoSQL databázích. Geovisto nabízí možnost uživateli poskytnou libovolná geoprostorová data v objektově-orientovaném formátu. Taková data je však zapotřebí za účely jejich projekce předzpracovat do interní reprezentace knihovny, která vyžaduje zploštělou datovou strukturu. Za tímto účelem knihovna disponuje automatickou transformací vstupních dat na obecný datový model, je tak možné přímo pracovat se serializovanými daty poskytnutými například webovým aplikačním rozhraním.

Další nevýhodou konkurenčních nástrojů jsou omezené možnosti konfigurace a interakce, které nabízí. Tyto nástroje se obvykle soustředí na obecnou 2D vizualizaci a povolují zobrazení základních tematických map s nízkým počtem atributů. Knihovna Geovisto nabízí možnost vytvářet projekce více vrstev a kombinovat je do jednoho zobrazení. Knihovnu lze navíc rozšířit připojením vlastních vrstev a nástrojů, filtrováním či selekcí atributů, nebo má vývojář možnost přetížením upravit implicitní mechanismy knihovny. Jednotlivé vrstvy mezi sebou komunikují a zajišťují tak potřebnou míru interakce vyobrazené mapy.

Motivací, která stála za vznikem knihovny Geovisto, bylo vytvořit snadno použitelnou knihovnu pro personalizovanou vizualizaci geografických dat, kterou by si však současně mohl pokročilý uživatel snadno rozšířit a modifikovat. Výsledkem je tak nástroj dostupný ve formě programové knihovny, který může být použit i jako autorský nástroj. Vývojáři mají možnost využívat poskytované aplikační rozhraní knihovny a pomocí přetěžování funkcí si chování knihovny rozšířit nebo modifikovat. Druhý způsob jak konfigurovat vykreslované vrstvy tematické mapy je použitím bočního panelu v grafickém uživatelském rozhraní (k vidění na obrázku 4.1). Konfigurace pomocí bočního panelu je možná díky již zmiňovanému předzpracování dat a uživateli jsou tak zpřístupněny jednotlivé domény. Se stavem mapy je možné pracovat a knihovna nabízí možnost konfiguraci mapy exportovat nebo importovat.

Boční panel ale není implicitně součástí knihovny, je to jeden z nástrojů, který je možné ke knihovně připojit, neboť struktura knihovny je rozdělena do jednotlivých modulů, kde přídavné moduly tvoří vrstvy mapy. V době psaní práce jsou oficiálně dostupné vrstvy pro boční panel, filtrování, selekci a výběr tématu. Z vrstev pro vytváření tematických map jsou to poté kartogramy, mapy značek, vrstva polygonů, mapa spojení a v přípravě je také podpora pro teplotní mapy. Na obrázku 4.1 je možné v levém bočním panelu vidět zástupce pro konfiguraci přítomných vrstev.



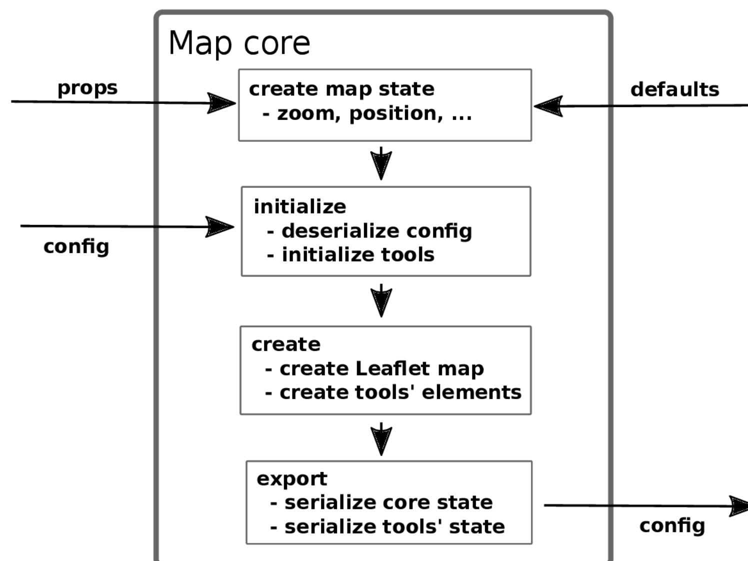
Obrázek 4.1: Ukázka rozhraní widgetu knihovny Geovisto zobrazující projekci více vrstev tematických map (zdroj: [17])

Pokud by chtěl uživatel využít například populární knihovnu Leaflet, musel by podrobně nastudovat nabízené aplikační rozhraní knihovny, poskytnout soubory s geografickými daty (například polygony pro kartogram) a na nižší úrovni vytvářet jednotlivé prvky, které se na mapě mají vyskytovat, a definovat jejich chování. V případě knihovny Geovisto jsou mu k dispozici vrstvy, se kterými jsou provázána poskytnutá data a výstupem je již hotová vrstva. Stále však nepřichází o rozsáhlé možnosti nízkoúrovňové personalizace, neboť má možnost přistupovat k objektu `L.Map`, který všechny Leaflet elementy mapy spravuje.

4.2 Architektura knihovny Geovisto

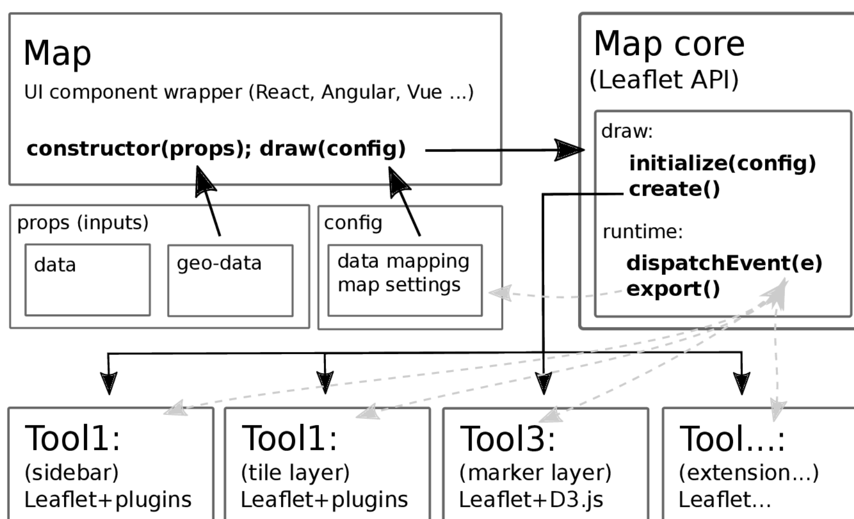
Knihovna Geovisto staví na principu modularity a její implementace je tak strukturována do jednotlivých logických celků (obrázek 4.3). Architekturu je tak možné rozdělit do dvou hlavních částí: jádro knihovny a nástroje.

Jádro knihovny je vstupním bodem a obstarává zpracování vstupních dat poskytnutých uživatelem a následnou inicializaci globálního stavu mapy. Chybějící vstupní data jsou doplněna implicitními hodnotami. Pokud je přítomen konfigurační soubor, obsažené informace mají nejvyšší prioritu a přepisují tak poskytnutá inicializační data. Tento proces postupného zpracování vstupních dat uvažující priority vstupů je zachycen na obrázku 4.2 v kontextu inicializace stavu mapy. Jádro knihovny je poté možné obalit do komponenty dle principů příslušného webového rámce pro tvorbu uživatelských rozhraní, podmínkou je však předání dat nezbytných pro inicializaci mapy a vyvolání vykreslení nad objektem mapy.



Obrázek 4.2: Schéma priority zpracování vstupů jádrem knihovny Geovisto (zdroj: [17])

Nástroje umožňují vytvářet vrstvy mapy s vizuální grafickou reprezentací nebo lze jejich užitím manipulovat s daty a zobrazení tak přímo ovlivňovat. Jedná se například o postranní panel nebo jednotlivé vrstvy mapy realizující grafickou projekci dat. Díky modulární architektuře je možné do projektu zahrnout pouze vybrané vrstvy, knihovna navíc umožňuje aktivovat/deaktivovat jejich zobrazení.



Obrázek 4.3: Abstraktní model architektury knihovny Geovisto (zdroj: [17])

Mezi nástroji reprezentujícími jednotlivé vrstvy probíhá komunikace, díky čemuž jsou všechny vrstvy schopné reagovat na změny a je tak zajištěna interakce mapy. Nástroje mezi sebou nekomunikují přímo, ale zasílají události jádru, které události zachytává, řadí je do fronty a zajišťuje jejich distribuci mezi nástroje. Každý registrovaný nástroj poté obdrží událost a v závislosti na identifikátoru je rozhodnuto, zda vrstva implementuje reakci na tuto událost nebo ji má ignorovat. Popsané chování tak implementuje návrhový vzor *observer*.

Vývojář si může rozšířit knihovnu o další nástroje, které budou tvořit nové vrstvy. Tento nástroj však musí implementovat příslušné rozhraní a respektovat tak způsob komunikace mezi již dostupnými vrstvami.

4.3 Použití Geovisto s webovými UI frameworky

V současnosti při implementaci front-end vrstvy webové aplikace vývojáři často volí některý z JavaScriptových webových frameworků pro tvorbu UI, neboť umožňují dekomponovat grafické uživatelské rozhraní do logických celků nazývaných jako komponenty. Rozdělení na komponenty umožňuje vytvářet jejich hierarchii a zapouzdření, hlavní výhodou je však jejich znovupoužitelnost. Každá komponenta má svůj vlastní životní cyklus a příslušné metody pro správu stavu umožňují synchronizaci mezi modelem a zobrazovaným obsahem (objektovým modelem dokumentu).

Geovisto v aktuální verzi postrádá vrstvu reprezentovanou zmíněnými komponentami a vývojář zapojení knihovny do projektu musí řešit sám, což vyžaduje jistou míru znalosti poskytovaného API a nutnost dodatečného implementování vrstvy propojující jádro knihovny s konceptem programování komponent. Pokud by existovala tato vrstva, vývojář by byl od dodatečného programování oproštěn a mohl by tak přímo tyto komponenty zahrnout do svého zdrojového kódu mezi ostatní obsah určený k vykreslení. To by uživateli značně usnadnilo proces zahrnutí knihovny do projektu.

V aktuálním stavu knihovny je řešením pro úspěšné vykreslení tematické mapy zapouzdření jádra knihovny do jedné komponenty příslušného aplikačního rámce (otestováno pro frameworky React a Angular). K vytvoření počáteční konfigurace knihovny lze využít vstupy komponenty a s jejich pomocí předat veškerá nastavení, včetně zahrnutých nástrojů, dat a geografických dat. Tento přístup ale vzhledem k bohatým možnostem přizpůsobení knihovny ústí v obsáhlý a nepřehledný kód. Pro inicializaci a promítnutí mapy do objektového modelu dokumentu je nutné využít logiky pro životní cyklus komponenty a zařídit vytvoření objektu mapy a její vykreslení voláním funkcí z jádra knihovny.

K datu psaní práce tato knihovna pro tvorbu tematických map v implementaci obsahuje jeden demonstrační příklad pro její zahrnutí do klientské aplikace. Tento kód je zasazen do paradigmatu aplikačního rámce React a vizualizace výstupu knihovny je demonstrována užitím podpůrného nástroje Storybook (popsaného v kapitole 3.3.1). Vývojář tak ze zdrojového kódu uvedeného příkladu může čerpat inspiraci pro práci s objektem mapy a API knihovny. Stále se však od uživatele očekává vytvoření zdrojového kódu realizujícího propojovací vrstvu mezi knihovnou a cílovou aplikací.

Zmíněný postup se jeví jako složitý, uživatelsky nepřívětivý a má negativní dopad na kvalitu zdrojového kódu vyvíjené aplikace. Řešením by bylo popsané chování abstrahovat a poskytnout vývojáři předdefinovanou sadu UI komponent, se kterými by mohl přímo pracovat bez nutnosti spravovat životní cyklus objektu mapy.

4.4 Shrnutí

Při integraci knihovny do projektu využívajícího UI framework je v současném stavu uživatel nucen spravovat životní cyklus objektu mapy a vytvářet si vlastní komponenty sloužící k reprezentaci mapy v objektovém modelu dokumentu za účelem grafického zobrazení výstupu. Je nutné získat všechny informace nezbytné ke konfiguraci mapy, objekt mapy v zápětí inicializovat, pomocí identifikátoru provázat s blokovým elementem předem definovaných

rozměrů a s každou aktualizací komponenty mapu překreslit. K dosažení popisovaného chování je nezbytné využít interních funkcí knihovny spravující daný objekt. Od vývojáře se tak očekává jistá míra znalosti vnitřního fungování implementovaných mechanismů nástroje Geovisto a základní znalost práce s poskytovaným aplikačním rozhraním.

Vzhledem k tomu, že komponenty jsou obvykle vytvářeny pro reprezentaci grafického celku ve výsledném zobrazení aplikace, je vhodné vytvořit abstrahující komponenty zejména pro nástroje knihovny, neboť renderují nějaký vizuální výstup. V případě, že by uživatel chtěl zahrnout vícekrát jednu z vrstev s odlišnými nastaveními, může použít vícekrát příslušnou komponentu s rozdílnými vlastnostmi. Vývojář by tak pomocí hierarchie komponent mohl nastavovat zahrnuté nástroje a jejich konfiguraci.

Vrstva se sadou komponent abstrahující přístup k objektu mapy musí být specifická pro příslušný aplikační rámec, neboť musí respektovat syntaxi a principy konkrétního frameworku. S cílem usnadnit použití knihovny co největšímu počtu vývojářů se nabízí popisovanou vrstvu abstrakce implementovat pro nejvíce užívané frameworky. V kapitole 3.1 byl proveden rozbor nejpopulárnějších frameworků, přičemž výsledkem průzkumu je zřejmá dominance aplikačního rámce React. Je tedy vhodné řešení realizovat pro zmíněný framework.

Po provedení analýzy současného stavu knihovny Geovisto byly zjištěny následující nároky programátorů na její rozšíření pro práci s UI frameworky:

1. Abstrahovat zahrnutí využitých nástrojů pomocí předdefinovaných komponent, které uživatel může přímo vložit do svého zdrojového kódu.

2. Projektovat životní cyklus mapy do životního cyklu komponenty odpovědné za správu objektu mapy.

3. Deklarativně definovat parametry nástrojů užitím vstupů jednotlivých komponent.

4. Odstínit uživatele od nutnosti pracovat s aplikačním rozhraním knihovny Geovisto při inicializaci modulů.

5. Zajistit nástrojům knihovny Geovisto schopnost dynamicky reagovat na změny vstupů reprezentujících komponent.

6. Neomezit možnosti personalizace, které knihovna Geovisto nabízí.

Kapitola 5

Návrh rozšíření

V reakci na předešlou kapitolu a v ní popisované nedostatky knihovny se tato kapitola zabývá bližším popisem navrhovaných nástrojů, jejichž záměrem je diskutované problémy vyřešit. Nejprve se zaměřím na vymezení problému v kontrastu s aktuálním řešením a popíši různé pohledy na možnosti řešení, včetně jejich kladů a záporů. Jádro kapitoly tvoří detailní popis teoretického navrhovaného rozšíření v podobě abstraktní vrstvy tvořené sadou komponent, které usnadní práci s nástrojem Geovisto. Na závěr popisuji způsob, jakým bude tato vrstva prezentována a distribuována koncovému uživateli.

Odstranění problémů popsaných v sekci 4.3 a zefektivnění práce s knihovnou je hlavní motivací pro realizaci navrhovaného rozšíření. Cílem je vytvořit vrstvu sloužící jako prostředník mezi knihovnou Geovisto a aplikací uživatele tak, aby uživatel mohl přímo využívat sadu předem definovaných značek a nebyl nucen kvůli integraci tematické mapy implementovat žádný přebytečný zdrojový kód. Tyto značky jsou v moderních webových frameworcích pro tvorbu UI realizovány ve formě komponent. Vrstva abstrahující rozhraní nástroje Geovisto bude řešena jako samostatná knihovna odstíněná od implementace knihovny Geovisto, pouze se závislostí na její ucelené balíčky veřejně dostupné prostřednictvím nástroje npm[1].

5.1 Uvažované přístupy

Původní myšlenkou byla snaha co nejvíce oprostit uživatele od nutnosti využít funkce poskytovaného aplikačního rozhraní a abstrahovat veškerou konfiguraci nástroje a jednotlivých vrstev užitím pouze předem definované sady komponent a jejich vlastností. Tento způsob však není vhodný zejména ze tří hlavních důvodů:

- Vznikla by silná závislost na stavu a možnostech knihovny v době implementace rozšíření. Knihovna Geovisto je konstruována způsobem, aby byla pro vývojáře pomocí přetížení funkcí snadno rozšířitelná a modifikovatelná. V případě využití tohoto přístupu by byly uživateli vymezeny striktní hranice jejího použití a knihovna by tak přišla o jednu ze svých hlavních výhod. Pokud bych například uvažoval o vytvoření komponenty pro konfiguraci dat mapy, pro každý způsob předzpracování dat by musela vzniknout nová komponenta. Tento způsob není přívětivý k budoucím rozšířením knihovny, ani k personalizaci vývojářem. Řešením by mohlo být vytvoření jedné univerzální komponenty pro data, předávání funkcí a zavedení zpětných volání, v tomto scénáři se však již vytrácí původní cíl tohoto přístupu – zjednodušení přístupu a odstínění od aplikačního rozhraní.

[1] https://www.npmjs.com/

- Aplikační rámce pro tvorbu uživatelských rozhraní využívají konceptu komponent s cílem dekomponovat grafické uživatelské rozhraní do menších znovupoužitelných bloků. Jedná se tak o celky reprezentující grafický výstup ve výsledném zobrazení. Pokud bych se rozhodl zavést komponenty i pouze pro účely konfigurace nástroje, neměly by ve vykresleném obsahu žádné vizuální zastoupení a to by znamenalo porušení zmíněného zavedeného konceptu.
- Testovatelné (například pomocí nástroje Storybook) jsou pouze komponenty, které mají přímý vliv na výslednou projekci v zobrazení aplikace klienta. Dříve zmíněnou komponentu pro data by tak nebylo možné testovat.

Z výše uvedených důvodů jsem se tak rozhodl do komponent zapouzdřit pouze prvky knihovny, které reprezentují vizuální prvky ve výsledném grafickém uživatelském rozhraní. Soustředit se tak budu pouze na vytvoření komponent pro nástroje, neboť právě ony tvoří jednotlivé zobrazované vrstvy mapy, a samotný objekt mapy. Obecné konfigurace, které nemají grafické zastoupení ve vykresleném obsahu, budou konfigurovány užitím vlastností komponenty zapouzdřující objekt mapy s využitím funkcí API. Myšlenkou tak je do komponent zapouzdřit nástroje, které je možné do knihovny libovolně zapojovat díky tomu, že každý nástroj je implementovaný jako modul.

Každý modul je však specifický a uvažoval jsem dva rozdílné přístupy, jak se k realizaci jejich abstrakce postavit:

- Prvním způsobem je vytvoření jedné obecné komponenty společné pro všechny moduly. Tento přístup s sebou nese ale řadu nevýhod. Vzhledem k tomu, že každý nástroj má jiné vlastnosti a s tím související odlišná nastavení, nebylo by možné zajistit typovou kontrolu. Případně by bylo možné od sebe odlišit obecnou komponentu pro nástroje a vrstvy mapy, neboť zástupci těchto skupin sdílejí společné vzory implementace, což by umožnilo alespoň částečný dozor nad typy předávaných dat. Tato komponenta by tak abstrahovala neurčitý nástroj a jeho typ by musel být určen uživatelem. Realizace předávání typu pomocí výčtu není vhodná, zejména kvůli intoleranci vůči budoucím rozšířením knihovny, neboť výčet by musel být pravidelně doplňován o typy nových modulů. Nabízí se určovat typ nástroje pomocí řetězce, tento přístup je však náchylný k překlepům a snižuje uživatelskou přívětivost řešení. Výhodou je odolnost řešení vůči nově přidaným modulům (nezávisle na jejich původu – od autorů knihovny či programátora, který si knihovnu rozšiřuje), neboť obecná komponenta by byla schopna zpracovávat informace pro předem neznámé typy nástrojů.
- Opačným pojetím je projekce každého nástroje do samostatné komponenty. Vytvořená komponenta abstrahující nástroj by tak byla velmi specifická a schopná zajistit typovou kontrolu všech možností konfigurace. Řešení by současně otevřelo prostor pro další zanořování komponent. Například konfigurace bočního panelu by mohla být rozdělena na jednotlivé dílčí komponenty, kde každá z nich by představovala konkrétní záložku na panelu a měla své vlastní nastavení. Nevýhodou přístupu je jeho závislost na aktuálním stavu knihovny, neboť je nutné, aby mezi poskytovanými moduly a nabízenými komponentami existovalo izomorfní zobrazení. Současně je však nutné zachovat možnost rozšíření knihovny externími nástroji, které si může programátor implementovat. Vzniká tak požadavek poskytovat i obecnou komponentu. Vzhledem k předem neznámé struktuře rozhraní modulu by však komponenta nebyla schopná provádět kontrolu typů. Bylo by ji ale možné dočasně využít i pro nově im-

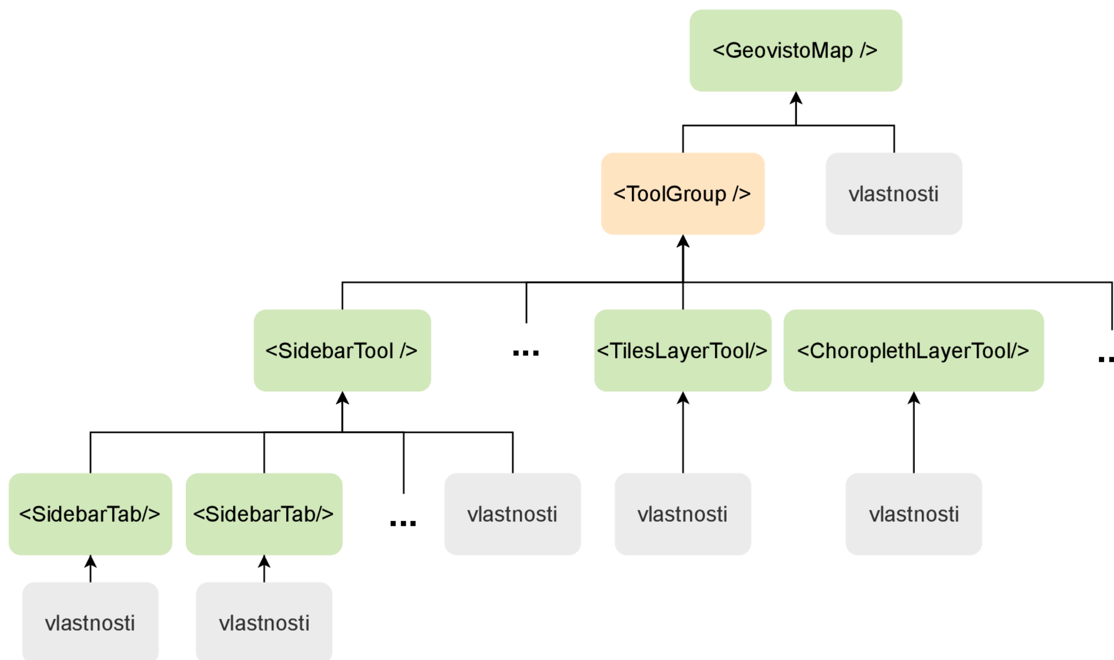
plementované moduly autory knihovny bez nutnosti ihned přidat novou komponentu reprezentující modul do již existující sady.

Vzhledem k charakteru požadovaného rozšíření jsem vyhodnotil druhý přístup jako uživatelsky přívětivější a další návrh řešení se řídí tímto přístupem. Vývojář tak bude mít dobrý přehled o parametrech a požadovaných typech, které konkrétní modul na vstupu očekává.

5.2 Návrh sady komponent

Navrhované rozšíření knihovny Geovisto tedy spočívá ve vytvoření mezivrstvy, kde každý modul bude abstrahován a bude mít svoji vlastní reprezentaci v podobě komponenty. Současně je však nutné zachovat veškeré vlastnosti knihovny a nijak neomezit možnosti, které nabízí. Cílem není jakkoliv obohatit funkcionalitu knihovny, ale vytvořit vrstvu abstrakce, která by usnadnila její použití v aplikacích zasazených do některého ze tří nejpopulárnějších frameworků.

Na obrázku 5.1 je možné vidět schéma hierarchie navržené sady komponent a různé úrovně jejich zanoření. Zelenou barvou jsou vyobrazeny komponenty, které mají úzkou spojitost s implementací knihovny, zejména se jedná o jednotlivé moduly projektované do komponenty. Šedé uzly vyjadřují vstupy/vlastnosti příslušné komponenty. Kořenový uzel tvoří komponenta zapouzdřující celý objekt mapy, je tvořena blokovým elementem promítnutým v objektovém modelu dokumentu a slouží jako kontejner pro veškerý obsah mapy. Její starostí je správa životního cyklu objektu mapy, volá tak funkce implementované přímo v jádru knihovny a zařizuje vytvoření mapy, její vykreslení a reakce na změnu stavu. Oranžovou barvou je v diagramu 5.1 vyjádřena komponenta, která nemá přímou oporu v implementaci knihovny, ale byla vytvořena s myšlenkou logicky seskupit nástroje mapy a moduly vykreslující jednotlivé vrstvy tematické mapy. Tato „skupinová komponenta“ zastřešuje všechny své přímé potomky, seskupuje od nich informace o jejich vstupech a je zodpovědná za propagaci všech informací kořenovému uzlu. Výhodou přidané úrovně je také distribuce logiky v rámci vrstvy a komponenta spravující objekt mapy tak není zbytečně zatěžována zpracováním všech potomků.



Obrázek 5.1: Hierarchie navrhovaných komponent

Přímými potomky „skupinové komponenty" jsou komponenty zapouzdřující konkrétní moduly, které knihovna nabízí, nebo vlastní nástroje vytvořené uživatelem. Komponenty na této úrovni jsou znovupoužitelné a mohou se v rámci skupiny vyskytovat vícekrát. V případě, že by tak uživatel chtěl vytvořit například dvě vrstvy pro zobrazení kartogramu, zvolí unikátní identifikátory a obě vrstvy mohou nést naprosto odlišnou konfiguraci. Další úrovně už jsou poté pouze komponenty reprezentující určitou přizpůsobitelnou vlastnost modulu.

Výpis 5.1 převádí hierarchii z diagramu 5.1 do podoby pseudokódu blízkému podobě zdrojového kódu, který by implementoval uživatel s využitím navrhovaného rozšíření.

```
<GeovistoMap
    id="my-geovisto-map"
    data={Geovisto.getMapDataManagerFactory().json(data)}
    config={Geovisto.getMapConfigManagerFactory().default(config)}
    ... >
    <ToolGroup>
        <SidebarTool id="tool-sidebar" ... >
            <SidebarTab name="Choropleth" enabled=false ... />
            <SidebarTab name="Markers" icon="fa fa-map-marker" ... />
            ...
        </SidebarTool>
        ...
        <TilesLayerTool id="tool-layer-map" ... />
        <ChoroplethLayerTool id="tool-layer-choropleth" ... />
        ...
    </ToolGroup>
</GeovistoMap>
```

Výpis 5.1: Pseudokód demonstrující návrh rozdělení do komponent

Jak je možné vidět na výše uvedeném příkladu, zahrnutí a přizpůsobení jednotlivých nástrojů probíhá pouze užitím dostupných komponent a jejich vlastností, přičemž zdrojový kód uživatele zůstal jednoduchý a přehledný.

Je zapotřebí brát v potaz, že samotné komponenty abstrahující moduly nevykreslují žádný obsah a jsou tak označovány jako takzvané *renderless* komponenty. Jejich funkce spočívá v zapouzdření volání API a slouží jako nosič informace o konfiguraci příslušného nástroje. Tyto informace musí být následně vhodně seskupeny, zapsány do globální konfigurace mapy a předány jádru knihovny ke zpracování. Grafické vykreslení zařizuje samotný modul, komponenta jej pouze abstrahuje a reprezentuje v konceptu komponent příslušného frameworku.

Vývojář využívající Geovisto je schopen přetížením metod aplikačního rozhraní modifikovat chování knihovny a užívat vlastní nástroje. Pro moduly vytvořené uživatelem bude nutné uvažovat generickou komponentu, která bude schopna provádět zpětná volání, neboť uživatel je v tomto případě zodpovědný i za inicializaci nástroje. Vstupy komponenty poté mohou nést různou strukturu a není tak možné kontrolovat dodržení typů.

5.3 Výstup

Výstupem realizace uvedeného návrhu bude sada předdefinovaných komponent, přičemž uvedený přístup umožňuje implementaci v různých aplikačních rámcích. Není však možné vytvořit jeden univerzální balíček pro všechny frameworky, neboť implementace je závislá na programovacím jazyce, syntaxi a konceptech konkrétního aplikačního rámce. Zmíněná sada bude ucelena a exportována jako samostatná knihovna, přičemž ke koncovému uživateli bude distribuována ve formě npm balíčku. Po jeho stažení bude mít vývojář přímý přístup ke komponentám. Protože popisované sady komponent tvoří abstraktní vrstvu nad knihovnou Geovisto a implementačně na ni přímo závisí, musí uživatel do svého projektu zahrnout i jádro této knihovny, rovněž dostupné ve formě npm balíčku[2].

[2] https://www.npmjs.com/package/geovisto

Kapitola 6

Implementace

Tato kapitola se věnuje detailnímu popisu fungování implementovaného řešení pro aplikační rámec React. Nejprve obecně popisuji architekturu vytvořené sady komponent, způsoby jakými mezi sebou jednotlivé komponenty komunikují a postupy jakými může uživatel nakládat s knihovnou. Dále kapitola pojednává o toku dat, jejich zpracování a klíčových mechanismech potřebných k inicializaci a aktualizaci mapy, zejména v návaznosti na implementaci jádra knihovny Geovisto a její jednotlivé moduly. Na závěr jsou popsány zavedené prvky optimalizace, které zlepšují uživatelskou zkušenost při práci s tematickou mapou vytvořenou užitím implementované sady komponent.

React je v současnosti dle počtu stažení (viz graf 3.2) dlouhodobě nejpopulárnějším aplikačním rámcem a s úmyslem zacílit na co největší množinu programátorů je tak výsledné řešení realizováno pro uvedený framework. Programátor využívající React má k dispozici dva koncepty jak vytvářet komponenty, pomocí třídního modelu nebo realizovat komponenty jako funkce. V řešení využívám druhý zmíněný koncept a výstupem je tak sada funkcionálních komponent. Uživatel je však může využít bez omezení i v případě, že ve svém zdrojovém kódu využívá třídní model.

Řešení bylo implementováno v jazyce TypeScript s využitím syntaxe TSX, neboť nabízí kontrolu datových typů, využití rozhraní a dalších podpůrných mechanismů pro kvalitnější zdrojový kód výsledného řešení. Tento jazyk byl navíc zvolen, aby mnou vytvořené řešení bylo konzistentní s knihovnou Geovisto, která rovněž využívá TypeScript. Volba tak přispěje k udržitelnosti knihovny. Koncový uživatel využívající implementovanou knihovnu není programovacím jazykem omezen a může ve svém projektu využívat i nativní JavaScript.

V popisu implementace hojně využívám pojmy *props* a *React Hook* specifické pro tento aplikační rámec, přičemž byly blíže vysvětleny v kapitole 3.2.1.

6.1 Architektura řešení

Implementace řešení respektuje navrženou strukturu v předchozí kapitole a je koncipována do hierarchie dle schématu 5.1. Programová vrstva abstrakce je tak tvořena sadou komponent, pomocí které je uživatel schopen ve zdrojovém kódu své webové aplikace (implementované s použitím frameworku React) přímo vytvářet a konfigurovat tématické mapy a efektivně spravovat nastavení jednotlivých vrstev, tedy instancí nástrojů. Komponentu každého modulu lze ve zdrojovém kódu použít vícekrát, přičemž každé použití produkuje jednu instanci nástroje a lze tak vytvářet více různých vrstev mapy stejného modulu.

Základní komponentou řešení, která vytváří instanci objektu mapy a zaštiťuje veškerý obsah konfigurace, je element `GeovistoMap`. Prostřednictvím jejich vstupů lze nastavit parametry zobrazení tematické mapy, které knihovna Geovisto uživateli nabízí. Jak je možné vidět na ukázce 6.1, mimo zmíněné *props* je od uživatele vyžadován identifikátor nezbytný pro provázání objektu mapy s blokovým elementem `div`, do kterého je obsah vykreslován, a seznam tříd kaskádových stylů, přičemž prostřednictvím stylů se od uživatele očekává vymezení rozměrů kontejneru mapy na webové stránce.

```
<GeovistoMap
    id="my-geovisto-map"
    className="geovisto-map-container"
    props
>
    <ToolGroup>
        komponenty nástrojů
    </ToolGroup>
</GeovistoMap>
```

Výpis 6.1: Zanořování komponent

Jediným potomkem kořenové komponenty je komponenta `ToolGroup`, která neočekává od uživatele žádná *props*, jedná se o komponentu spravující manažera nástrojů a centrální logiku pro zpracování potomků – komponent reflektujících konfigurace instancí jednotlivých nástrojů.

Použití komponent ke konfiguraci jiného objektu není však jejich typickým využitím, neboť jak bylo popsáno v kapitole 3, komponenty jsou zamýšleny k vytváření vizuální reprezentace pomocí elementů zasazených do objektového modelu dokumentu. V případě mého řešení však funkci veškerého renderování obsahu zastává knihovna Geovisto a komponenty hrají roli stavebních kamenů, prostřednictvím kterých je možné jednotlivé nástroje pouze konfigurovat. Vzhledem k tomuto faktu tak bylo zapotřebí vytvořit takzvané *renderless* komponenty, tedy takové komponenty, které nemají vlastní reprezentaci v objektovém modelu dokumentu, ale současně je aplikační rámec při generování/vykreslování obsahu neignoruje. Všechny komponenty tedy renderují prázdný element `<></>` (případně s vnořenými potomky), neboť v takovém případě metoda pro vykreslení obsahu není prázdná a aplikační rámec React s takovým kódem pracuje jako s běžnými komponentami. Stále tak uchovává informace o jejich stavu a poskytuje možnost reagovat na události.

Při prvotním načtení komponenty `GeovistoMap` je kontrolována validita potomků (vnořených elementů). V mém řešení je jediným akceptovaným přímým následníkem komponenta `ToolGroup` (viz schéma 5.1), jakýkoliv jiný vnořený element komponenty `GeovistoMap` bude ignorován a tato skutečnost bude zapsána do vývojářské konzole. Informaci o podporovaných přímých potomcích uchovává konstanta `supportedTopLevelComponentTypes`, implementace tak počítá s dalšími možnými budoucími rozšířeními konfigurace mapy s užitím komponent. Pokud nemá komponenta spravující objekt mapy žádný platný vnořený element, je ihned přistoupeno k vykreslení tematické mapy.

Programátor má možnost přistupovat k objektu mapy, který reprezentuje a udržuje jako svůj stav komponenta `GeovistoMap`. Ve svém zdrojovém kódu toho uživatel docílí vytvořením reference *Ref* pro přístup k React elementům pomocí React Hook `useRef`, či funkce `React.createRef()` v případě třídních komponent. Dále je nezbytné nastavení atributu

`ref` zmíněné komponenty `GeovistoMap`. Pro uživatele je poté prostřednictvím vytvořeného odkazu připravená metoda `getMap()`, která jako návratovou hodnotu poskytuje aktuální objekt mapy shodný s vykresleným zobrazením. Uživatel tak může k tomuto objektu přistupovat a modifikovat ho bez omezení tak, jak nabízí knihovna Geovisto.

Klíčovou komponentou řešení je komponenta `ToolGroup` zapouzdřující všechny využité nástroje a jejich nastavení. Tato komponenta spravuje manažer nástrojů jádra knihovny Geovisto a implementuje veškerou logiku nezbytnou pro inicializaci a aktualizaci nástrojů tematické mapy. Od svých potomků však očekává již připravená data ve formátu odpovídajícímu objektu *props* nástrojů v interní reprezentaci knihovny Geovisto. Potomky komponenty `ToolGroup` jsou komponenty jednotlivých nástrojů, ty jsou odpovědné za přípravu dat, přičemž mohou být dále konfigurovatelné pomocí vnořených elementů (viz schéma 5.1). Každý platný element uvedený jako přímý potomek komponenty `ToolGroup` reprezentuje jednu instanci konkrétního modulu. Uživatel má možnost pomocí implementované sady komponent konfigurovat i své vlastní nástroje, které nejsou součástí oficiálně podporovaných modulů.

Pokud má komponenta `ToolGroup` ve zdrojovém kódu uživatele definované neplatné potomky, tedy potomky, kteří nepředstavují komponentu některého z nástrojů (určeno konstantou `supportedComponentTypes`), jsou stejně jako v případě kořenové komponenty elementy ignorovány a oznámeny uživateli prostřednictvím vývojářské konzole. Knihovna dále pracuje pouze s podporovanými typy komponent.

Tok dat a komunikace mezi komponentami

Jak již bylo zmíněno v kapitole 3.2.1, React podporuje pouze *one-way binding*, tedy aplikační rámec je konstruován tak, aby byla data předávána od rodičovské komponenty potomkovi ve formě již zmíněných *props*. V případě konfigurace objektu mapy pomocí komponent je však zapotřebí, aby jednotlivé komponenty reprezentující nastavení konkrétních nástrojů předávaly data svému rodiči a data tak byla propagována až do kořenové komponenty, tedy zapouzdřující komponenty `GeovistoMap` spravující objekt mapy. Bylo tedy nezbytné vyřešit otázku, jak může komponenta své vstupy zpracovat a předat je své nadřazené komponentě situované v hierarchii o úroveň výše.

Při řešení uvedeného problému je třeba brát v potaz smysl realizované programové vrstvy, zdrojový kód aplikace programátora nesmí být nijak zatížen a musí se soustředit na deklaraci vrstev mapy, je nepřípustné, aby uživatel musel komponentám předávat nějaké předem specifikované parametry (*props*) vyžadované pro fungování knihovny. Uživatel musí nastavovat přímo pouze parametry ke konfiguraci jednotlivých nástrojů, případně samotného objektu mapy. Současně není vhodné implementačně zatěžovat ani jednotlivé komponenty reprezentující nástroje mapy, neboť je vyžadována maximálně možná jednoduchost přidání nového nástroje. O zahrnutí podpory nového nástroje do implementace řešení pojednává příloha A. Knihovna Geovisto je stále ve vývoji, expanduje a autoři knihovny stále přidávají nové moduly (nástroje/vrstvy) rozšiřující funkcionalitu této knihovny.

Jedním z východisek předávání dat mezi potomky a jejich rodiči je použití *React Context API*, které umožňuje vytvořit centralizovaný stav přístupný z více různých komponent. Problémem v případě takového řešení by byla reakce na programovou změnu vstupů a následná aktualizace konkrétního nástroje, neboť by docházelo ke ztrátě informace o zdroji vyvolání změny a musel by na tento problém existovat externí mechanismus. Zejména v případě, kdy by existovalo více instancí jednoho nástroje. Dalším problémem je granularita nastavení jednotlivých nástrojů, například `SidebarTool` komponenta je dále konfigurovatelná

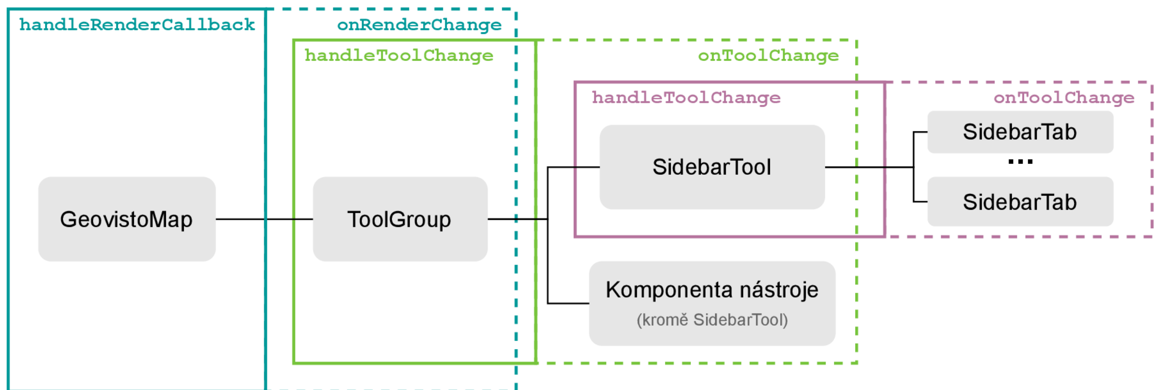
pomocí potomků – komponent `SidebarTab`, tedy při změně potomka by musel existovat mechanismus pro detekci a vyvolání aktualizace rodičovské komponenty, což by efektivní aktualizaci zobrazení mapy opět komplikovalo.

Jako efektivnější se jeví využité řešení spoléhající na funkce zpětného volání, nazývané jako *callback*, kde potomek v případě potřeby volá definovanou funkci a data předává jako parametr. Rodičovská komponenta poté zachytává volání této funkce a získává tak přístup k datům od potomka a může s nimi dále libovolně pracovat. Omezením v případě využití tohoto způsobu propagace dat je nutnost předat funkci obslužné rutiny události (takzvaný *event handler*) komponentě jako vstup společně s ostatními *props*. Objekt *props* je přitom pouze pro čtení a není ho možné modifikovat v jiné části zdrojového kódu, než při iniciální deklaraci vstupů komponenty. Uživatel by tedy musel při konfiguraci mapy předávat každé komponentě další parametr s funkcí obslužné rutiny, což porušuje stanovené cíle nezatěžovat uživatele interním fungováním implementované knihovny.

Byl tedy vymezen cíl rozšířit objekt *props* o funkce zpětného volání a nezasahovat přitom žádným způsobem do zdrojového kódu uživatele. Realizace cíle bylo v implementaci dosaženo duplikováním komponent potomků. Pro každou vnořenou komponentu představující nelistový uzel je vytvořen její klon a jsou jí předány kopie vstupů, vždy rozšířené o určitý *callback*. Jejich duplikace tak umožní kontrolovaně rozšířit objekt *props* v implementovaných komponentách. Původní komponenty specifikované uživatelem jsou nahrazeny klony, které jsou nadále knihovnou zpracovány. Své potomky duplikují a rozšiřují všechny komponenty očekávající alespoň jeden vnořený element. Vyvolat obsluhu v rodičovském uzlu prostřednictvím *callbacku* lze ku příkladu následovně:

`props.onToolChange(`*`data`*`)`

Komponenta `GeovistoMap` rozšiřuje programátorem specifikované vstupy svých potomků o deklaraci funkce `onRenderChange`. V případě zavolání této funkce podřízenou komponentou `ToolGroup` je vyvolána obslužná metoda zajišťující vykreslení/překreslení tematické mapy. Jako parametr přitom očekává již zkompletovaný manažer nástrojů. Vnořené komponenty, které očekávají další potomky (tedy `ToolGroup` a `SidebarTool`) *props* svých vnořených elementů rozšiřují o *callback* `onToolChange`, prostřednictvím kterého si komponenty předávají informace o změnách svého stavu. Signatura metody definuje jeden povinný a druhý nepovinný parametr. Prvním parametrem jsou data komponenty, druhý poté může nést informaci o tom, který vstup byl modifikován. Tímto způsobem jsou propagována data mezi komponentami od listových uzlů až po ten kořenový. Využíváno je toho při procesu inicializace mapy a programové změny parametru některého z nástrojů ve chvíli, kdy tematická mapa je již vyhotovena. Procesu sestavení konfigurace a inicializace mapy se blíže věnují následující sekce této kapitoly. Přiřazení metod zpětného volání a obslužné rutiny jsou vizualizovány na obrázku 6.1:



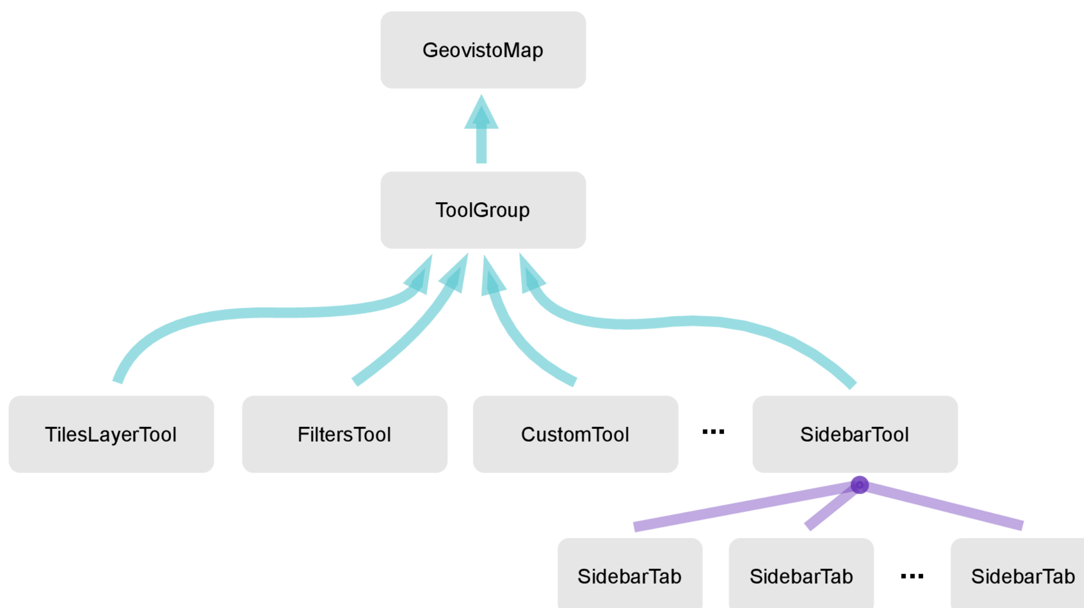
Obrázek 6.1: Přiřazení metod zpětného volání komponentám

Obdélníkem s plným obrysem je na obrázku 6.1 vždy ohraničena komponenta, která poskytuje obslužnou rutinu pro zpětné volání. Obdélníkem s přerušovaným ohraničením jsou poté seskupeny komponenty, jejichž *props* byly o příslušnou metodu rozšířeny a mohou tak daný *callback* vyvolat. Související zpětná volání a obslužná rutina je vždy vyznačena shodnou barvou.

6.2 Inicializace objektu mapy

Předtím, než je uživateli tematická mapa poprvé vykreslena, je nutné sestavit objekt reflektující interní reprezentaci parametrů mapy v knihovně Geovisto. Zmíněný objekt je následně předán funkci pro vykreslení. V rámci tohoto procesu dochází k seskupení informací ze všech listových komponent, jejich zpracování a postupné propagaci v hierarchii o úroveň výše.

Reakce na osazení komponenty je řešena pomocí vlastního React Hook `useToolEffect`. Využívá přitom oficiálně poskytované implementace `useEffect`, která umožňuje spouštět kód v závislosti na životním cyklu komponenty. Vlastní React Hook `useToolEffect` vyvolá při osazení komponenty nebo změně některé z jeho závislostí zpětné volání `onToolChange` popsané v předchozí sekci a jako parametr předává formátované *props*. Komponenty nástrojů takto mají kontrolu nad svými *props*, mohou je upravovat do potřebného tvaru vyžadovaného implementací knihovny Geovisto a dále je předávají rodičovské komponentě `ToolGroup`, jak je na obrázku 6.2 znázorněno modrými šipkami.



Obrázek 6.2: Tok dat při inicializaci nástrojů a vytváření tématické mapy

Jak lze pozorovat na obrázku 6.2 demonstrujícím směr toku dat při inicializaci instance mapy, zpracování komponenty `SidebarTool` se od ostatních nástrojů lehce liší. Zatímco ostatní komponenty nástrojů předávají přímo svá *props* komponentě `ToolGroup`, komponenta `SidebarTool` nejprve zpracovává své potomky – jednotlivé záložky postranního panelu. Při osazení komponenty tak nejprve iteruje přes své potomky, přistupuje k jejich *props* a pro každého vytváří novou instanci třídy `SidebarTab` (jedná se o instanci třídy z implementace knihovny Geovisto, nikoliv o instanci komponenty). Očekávané *props* komponenty `SidebarTab` musely být rozšířeny o parametr `tool` s cílem získat identifikátor nástroje, ke kterému záložka postranního panelu přísluší. Identifikátor nástroje je poté spojen s instancí záložky a je uložen do pomocného pole k dalším, již zpracovaným, komponentám `SidebarTab`. Popsané zpracování komponent záložek postranního panelu do interní reprezentace knihovny Geovisto má na starosti metoda `getProcessedTabs`. Metodě `onToolChange` poté jako parametr předává objekt obsahující *props* komponenty `SidebarTool` rozšířený o zpracované záložky. Zpracování záložek postranního panelu je na obrázku 6.2 znázorněno fialovými čárami, záměrně se nejedná o šipky, neboť `SidebarTool` při inicializaci mapy přistupuje ke svým potomkům přímo ze své implementace, nepřijímá přitom data od svých potomků pomocí zpětných volání.

Komponenta `ToolGroup` zachytává zpětná volání od všech vnořených komponent a pro jejich konfigurace vytváří instance tříd jednotlivých nástrojů. Manažer nástrojů, jež je udržován jako stav komponenty `ToolGroup`, vždy zahrne nově vytvořenou instanci modulu do svého seznamu. Ve chvíli, kdy je počet zpracovaných nástrojů shodný s počtem validních potomků, které komponenta `ToolGroup` má, znamená to, že byly zpracovány konfigurace všech nástrojů a může být přikročeno k renderování mapy. Je tedy vyvolán *callback* `onRenderChange` s manažerem nástrojů jako jediným parametrem. Kořenová komponenta zpětné volání zachytí, zkompletuje manažer nástrojů se svými *props* a přechází k vykreslení tematické mapy.

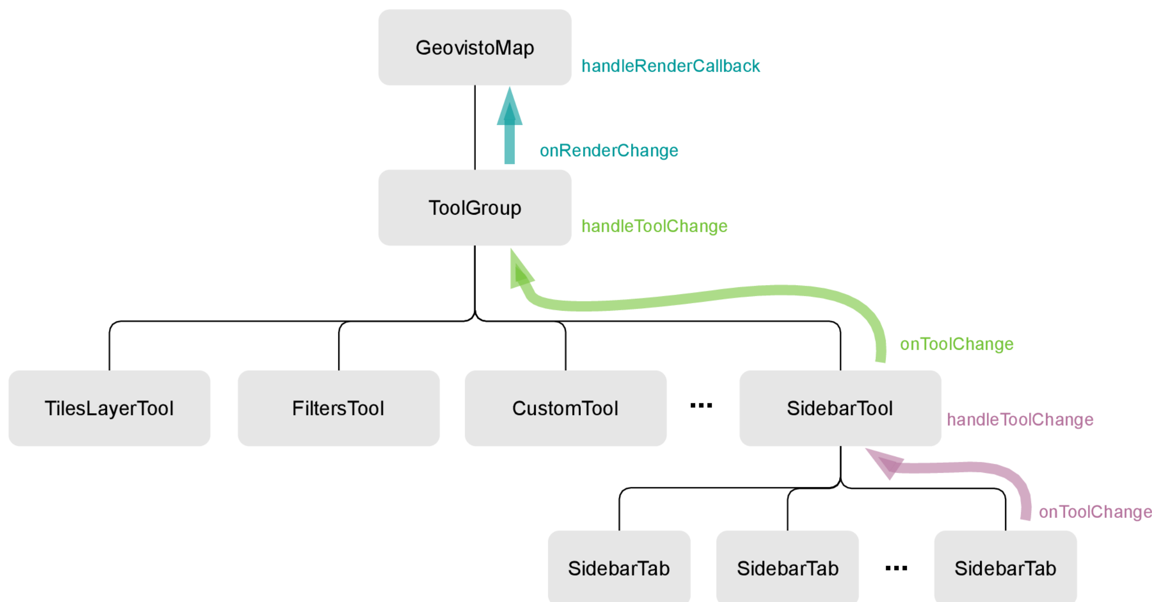
6.3 Aktualizace konfigurace komponenty

K aktualizaci vstupů komponenty může dojít v případě, kdy uživatel pro konfiguraci komponenty nepoužívá konstanty, ale proměnné nebo funkce. Podobně jako v případě inicializace, i v případě modifikace některého ze vstupů, komponenty detekují změny komponenty samotné a po vykonání vlastní logiky předávají řízení společně s daty rodičovským komponentám.

Mohou přitom využívat již představený React Hook `useToolEffect` využívaný pro inicializaci, neboť reaguje na všechny události v rámci životního cyklu komponenty. Jeho implementace ale v níže představených případech nemusí být dostačující, a tak bylo zapotřebí vytvořit nové React Hooks, které by komponenty mohly využívat: `useDidUpdateEffect`, `useDidToolIdUpdate`, `useDidToolEnabledUpdate`. Poslední dva uvedené se týkají aktualizace v případě změny identifikátoru nebo povolení vykreslení nástroje. Tyto případy včetně popisu implementace zmíněných React Hooks jsou blíže představeny v sekcích 6.3.1 a 6.3.2. Všechny varianty implementace přitom využívají stejný callback `onToolChange`, liší se tím, v jaké fázi životní komponenty figurují, jaké parametry předávají a jaké podmínky musí splňovat pro vykonání zpětného volání.

Zmíněný React Hook `useDidUpdateEffect` implementuje obdobné chování jako funkce `componentDidUpdate` v případě realizace komponent užitím konceptu tříd, funkcionální komponenty však jeho obdobu nemají, a tak bylo nutné vytvořit vlastní alternativu. V porovnání s dostupným řešením `UseEffect` nereaguje při iniciálním nasazení komponenty, tělo funkce předané v podobě parametru je vyvoláno v okamžiku, když se hodnota některé ze specifikovaných závislostí změní. V implementaci k tomu využívá React Hook `useRef`, který v tomto případě plní funkci uchování pravdivostní hodnoty napříč přerenderováními komponenty. Hodnota je iniciálně nastavena na hodnotu `false` a po nasazení komponenty je překlopena na hodnotu `true`. Před tím, než se má vykonat tělo funkce, je testována podmínka na tuto hodnotu, tím je dosaženo, že je vynecháno spuštění kódu při zavedení komponenty.

Pro detekci změny hodnot vstupů využívá React Hook `useDidUpdateEffect` například komponenta `SidebarTab`. Inicializace instancí této komponenty řeší rodičovská komponenta `SidebarTool`, reakci na změny vstupů poté už řeší samostatně, z tohoto důvodu je nutné nereagovat na událost při nasazení komponenty. Při změně vstupů komponenty poté volá callback `onToolChange`, který zachytí nadřízená komponenta `SidebarTool`. Ta provede zpracování svých potomků metodou `getProcessedTabs` stejně jako v případě inicializace a informaci o změně společně s aktuálními daty postranního panelu zasílá komponentě `ToolGroup`. Tento postup je znázorněný na schématu 6.3.



Obrázek 6.3: Demonstrace propagování programové změny parametrů instance komponenty `SidebarTab`

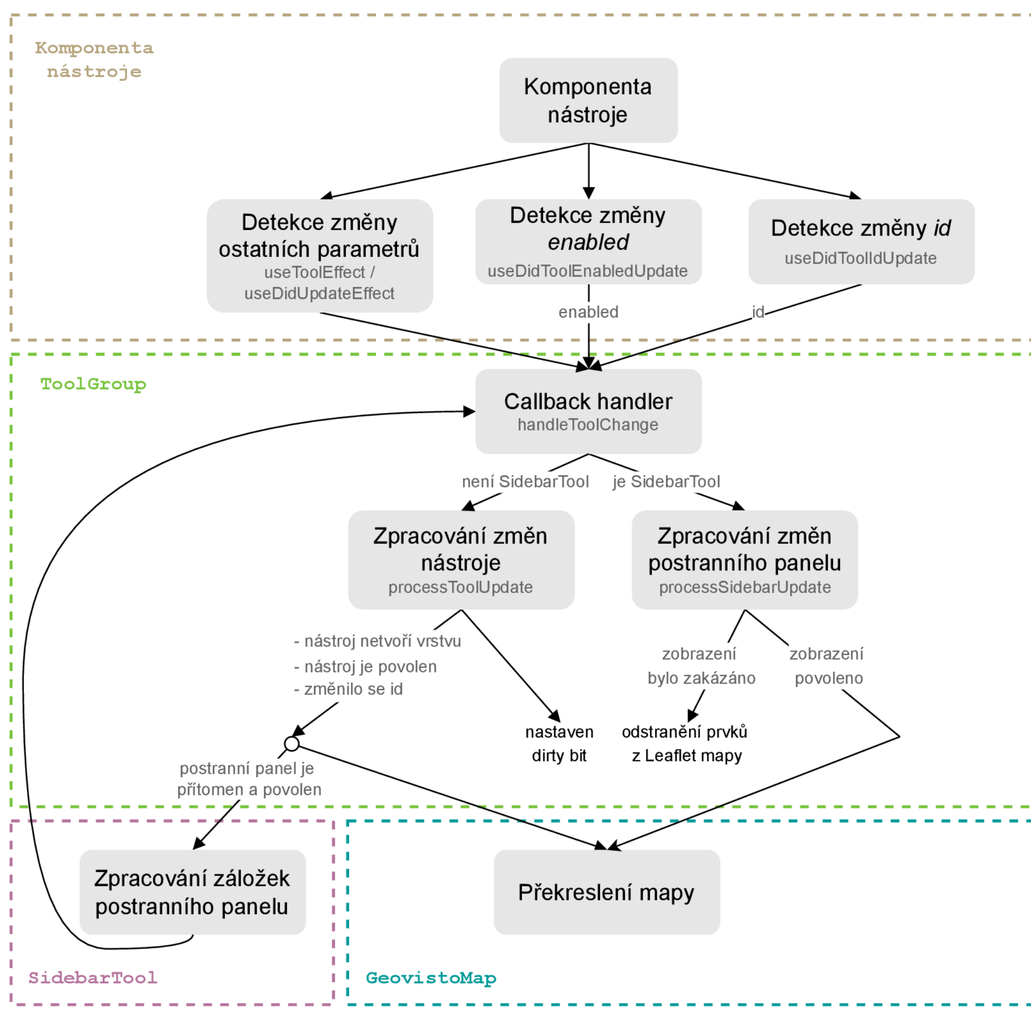
Komponenty reprezentující přímo některý z modulů využívají pro detekci změny vstupů React Hook `useToolEffect`. Jeho prvotní zpětné volání `onToolChange` je tak využito pro inicializaci nástrojů, každé další volání je klasifikováno jako aktualizace stavu některé z komponent. Při jakékoliv změně vstupů tak předává svá aktuální data komponentě `ToolGroup`, která s daty dále pracuje.

Spuštěná obslužná rutina se liší dle komponenty, která ji vyvolala, obecně ale platí rozdělení do dvou skupin – odlišný přístup vyžaduje komponenta `SidebarTool`, ostatní komponenty nástrojů jsou poté zpracovávány stejným způsobem.

V případě všech nástrojů ale bylo zapotřebí vyřešit otázku, jak při změně parametrů komponenty aktualizovat *props* a stav nástroje (anglicky *state*) v objektu mapy a posléze promítnout aktualizovanou konfiguraci do vykresleného zobrazení. Již samotná změna *props* interního objektu knihovny Geovisto nachází problémy, neboť aplikační rozhraní knihovny neposkytuje žádnou metodu, která by to umožňovala. Bázová třída pro všechny nástroje `MapObject` implementuje metodu `setProps` umožňující nastavení *props*. Ta je však opatřena modifikátorem přístupu `protected` znemožňujícím přístup k metodě. Musel by tak být proveden zásah do implementace knihovny a modifikátor přístupu by musel být změněn za klíčové slovo `public`. Změna stavu objektu je možná pouze užitím metody `initialize`, která je spuštěna v rámci konstruktoru instance nástroje při jeho vytváření. I promítnutí změn do vykreslené mapy by bylo problematické, neboť i metody manipulující s prvky knihovny Leaflet, která stojí za vytvořením jednotlivých elementů mapy, jsou opatřeny modifikátorem přístupu `protected`.

Z uvedených důvodů je proces reflektování změn vstupů komponenty řešen odstraněním a opětovným zavedením instance nástroje. Konstruktoru nástroje jsou předány aktualizované *props*, inicializuje se stav s novými hodnotami a jsou přidány žádoucí elementy knihovny Leaflet. Následně už je nutné pouze synchronizovat stav objektu mapy s vykresleným obsahem.

Metoda odpovědná za aktualizaci vstupů postranního konfiguračního panelu včetně jeho záložek (tedy komponent `SidebarTool` a `SidebarTab`) nese název `processSidebarUpdate`. Její logiku tvoří popsané nahrazení objektu bočního panelu novou instancí a následný pokyn k překreslení mapy pomocí zpětného volání `onRenderChange`. Pokud však obslužná rutina byla vyvolána z důvodu změny parametru `enabled` na hodnotu `false`, tedy zakázání zobrazení postranního panelu ve vykresleném obsahu, jsou pouze z mapy odebrány příslušné elementy knihovny Leaflet. Stav komponenty je změněn a uživateli se již zobrazuje tematická mapa bez postranního panelu, a to bez nutnosti překreslit celý obsah mapy. Rovněž pokud dojde dále k jakékoliv změně parametru, k překreslení nedochází do chvíle, než je zobrazení opět povoleno. Tento rozhodovací proces je znázorněn v diagramu 6.4.



Obrázek 6.4: Předávání řízení při aktualizaci konfigurovatelných dat nástrojů[1]

Jak je možné vyčíst z výše uvedeného diagramu 6.4, za předpokladu, že zpětné volání `onToolChange` vyvolá jakýkoliv jiný nástroj nežli postranní panel, o obsluhu aktualizace nástroje se stará metoda `processToolUpdate`. Obdobně jako v předchozím případě ze seznamu manažera nástrojů odstraní starou instanci nástroje a nahradí ji novou. Pokud změ-

[1]Poznámky u šipek v rámci rozhodovacího stromu signalizují příslušnou podmínku. Pokud poznámka není uvedena, předpokládají se všechny ostatní případy, které jsou v dané situaci platné.

něný nástroj vytváří vrstvu tématické mapy a tedy implementuje rozhraní `ILayerTool`, změna povolení zobrazení (parametru `enabled`) lze řešit efektivně bez nutnosti překreslení mapy. Tomuto řešení se věnuje sekce 6.3.2. V opačném případě, kdy modul implementuje pouze základní rozhraní nástroje `IMapTool`, je nutné i při změně parametru `enabled` vyvolat opětovné vykreslení mapy. K překreslení musí dojít i za předpokladu, že se změnil identifikátor nástroje nebo v případě, že se změnil jakýkoliv jiný parametr a zobrazení nástroje je povoleno.

Nástroje však mohou být provázané se záložkou konfiguračního panelu, a i když stará instance nástroje byla odstraněna, záložka stále obsahuje informaci o její zastaralé verzi. Metoda se tedy snaží zjistit, zda uživatel mezi nakonfigurovanými nástroji využil i postranní panel. Pokud je komponenta postranního panelu nakonfigurována a jeho zobrazení je povoleno, musí být aktualizovány i záložky nástrojů. Komponenta `ToolGroup` za tímto účelem vytváří odkaz `ref` na komponentu `SidebarTool`, která pomocí odkazu zpřístupňuje metodu `getTabs`. Uvedená metoda má jediný úkol, vyvolá zpětné volání `onToolChange` s aktuálními daty komponenty `SidebarTool`. Komponenta `ToolGroup` poté zpětné volání zachytí, aktualizuje postranní panel a vyvolá překreslení mapy (tzn. provede se pravá větev rozhodovacího stromu na diagramu 6.4). V tuto chvíli je tak synchronizovaný stav nástroje i zobrazení tematické mapy s postranním konfiguračním panelem.

6.3.1 Aktualizace identifikátoru

Chce-li uživatel změnit identifikátor komponenty nástroje, knihovna se k této změně musí stavět jinak, než jak je tomu u ostatních vstupů. Jak bylo zmíněno v předchozí sekci, aktualizace nástrojů probíhá pomocí nahrazení instance modulu se zastaralými daty za nově vytvořenou instanci. Zastaralý nástroj je přitom vyhledán a odebrán užitím identifikátoru a metodou `removeById`, kterou implementuje manažer nástrojů. Vznikl tak požadavek uchovávat hodnotu identifikátoru před jeho změnou, aby mohla být odebrána stará instance nástroje a nahrazena novou instancí s novým identifikátorem.

Reakce na změnu identifikátoru nástroje je řešena implementací vlastního React Hook `useDidToolIdUpdate`, který ukládá hodnotu identifikátoru do stavu komponenty a reaguje na změnu *props*, konkrétně pouze parametru `id`. Využívá přitom `useEffect`, kdy při nasazení komponenty pouze ukládá hodnotu identifikátoru. Při každé další změně závislostí volá zpětné volání `onToolChange`, kdy předaná *props* rozšiřuje o parametr `prevId` nesoucí hodnotu identifikátoru před jeho změnou. Jako druhý parametr předává konstantu `ID_PROP`, metoda `handleToolChange` díky tomu ví, že je v datech přítomna i původní hodnota identifikátoru a může se tomu při aktualizaci nástroje přizpůsobit. Po dokončení aktualizace je identifikátor uchovaný ve stavu komponenty nahrazen tím aktuálním, aby při další změně byla opět dostupná informace o identifikátoru v současnosti přítomného nástroje v objektu mapy. Hodnota nového identifikátoru je při každé změně podrobena validaci, pokud nová hodnota není platný řetězec znaků, knihovna zobrazí uživateli obrazovku s chybou.

Změna identifikátoru samozřejmě vyžaduje překreslení mapy, neboť nástroj může být propojen s dalšími mechanismy knihovny, ku příkladu s některou ze záložek postranního panelu, která nese informaci o identifikátoru nástroje, ke kterému patří. Uživatel si při konfiguraci nástrojů sám zodpovídá za změnu identifikátoru na všech místech výskytu ve svém zdrojovém kódu.

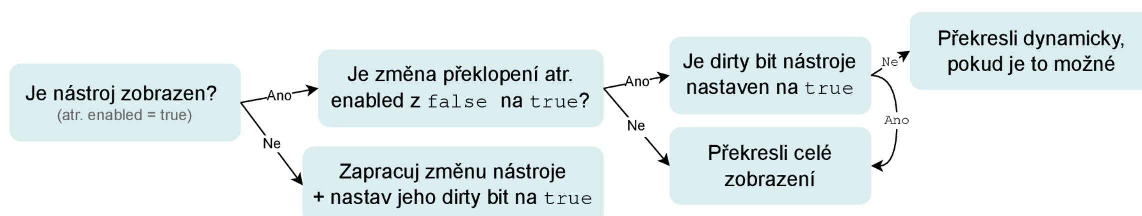
6.3.2 Aktualizace stavu zobrazení

Každé překreslení tematické mapy má negativní vliv na celkovou uživatelskou zkušenost s příslušnou aplikací. Implementovaná vrstva abstrakce se snaží minimalizovat počet překreslení na nejnutnější minimum, které knihovna Geovisto umožňuje. Zaměřuje se tak na atribut `enabled` definovaný v rozhraní `IMapToolProps`. Atribut je implementačně povinný pro všechny nástroje a určuje, zda je pro daný nástroj povolené jeho zobrazení. Konfigurace nástroje tak může být přítomna v objektu mapy, ale jeho elementy vytvořené knihovnou Leaflet, prostřednictvím knihovny Geovisto, nejsou vykresleny. Optimalizace spočívá v přistupování přímo k prvkům mapy Leaflet v okamžicích, kdy to implementace knihovny Geovisto povoluje.

Důvodem, proč se v implementaci zaměřuji na tento atribut, je možnost pracovat přímo s Leaflet elementy prostřednictvím aplikačního rozhraní knihovny Geovisto. Z jejího rozhraní je totiž možné nad instancí nástroje volat metodu `setEnabled` odpovědnou za dynamické zobrazování/skrývání nástroje. Tato metoda v závislosti na aktuálním stavu komponenty přistupuje k funkcím `showLayerItems` pro zobrazení a `hideLayerItems` pro skrytí. Obě metody přímo pracují s elementy knihovny Leaflet a dle potřeby je přidávají a odebírají. Tímto dochází ke změně zobrazení tematické mapy vykreslené uživateli, přitom však není nutné překreslit mapu jako celek tak, jak je tomu u ostatních atributů. Zmíněnou funkcionalitou však disponují pouze nástroje, které vytváří novou vrstvu tématické mapy a implementují tak rozhraní `ILayerTool`. Synchronizaci aktuálního zobrazení se stavem nástroje v objektu mapy zařizuje již mnou implementovaná knihovna.

Dalším zavedeným optimalizačním mechanismem s cílem minimalizovat potřebu překreslení je zavedení pomocného pole `toolDirtyBitArray`, které uchovává informaci o tom, zda byl nástroj změněn během toho, co nebyl zobrazen. Pokud nástroj není vykreslen v tematické mapě a změní se některý z jeho vstupů, není potřeba iniciovat změnu v zobrazení, neboť vykreslený obsah by se nezměnil. V takovém případě je nutné pouze aktualizovat stav instance, aby data instance nástroje byla aktuální, a uchovat informaci o nutnosti překreslení, pokud má být nástroj zobrazen. To umožňuje odložit překreslení až do chvíle, kdy je to nezbytně nutné.

Jak bylo popsáno v úvodu této sekce, nástroje lze dynamicky zobrazovat/skrývat užitím metod `showLayerItems` a `hideLayerItems`. Dynamické vykreslení vizuálních prvků nástroje však není možné v případě, pokud se během období, kdy byl nástroj skryt, změnil některý z jeho vstupů a nedošlo k jeho překreslení. V uvedeném případě je nutné překreslit celý objekt mapy, neboť nová instance nástroje do něj musí být nejprve zavedena. Tento proces je zachycen na obrázku 6.5. Aby bylo možné rozlišit stav, kdy je možné prvky dynamicky vykreslit a kdy je nutné překreslení celého objektu mapy, bylo zavedeno zmíněné pomocné pole `toolDirtyBitArray`.



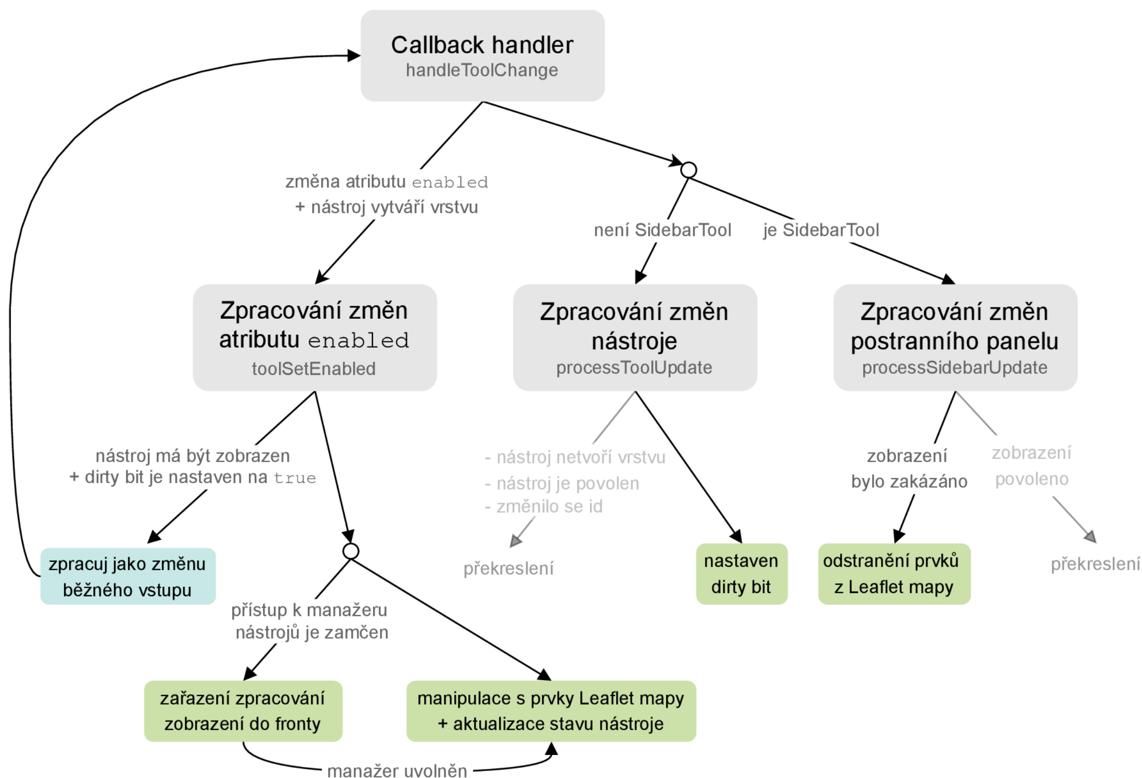
Obrázek 6.5: Rozhodovací proces při optimalizaci zpracování změny vstupu

Pomocné pole je realizováno jako seznam dvojic identifikátor–pravdivostní hodnota. Identifikátor nese hodnotu atributu `id` komponenty, pomocí které se instance v knihovně prokazuje, pravdivostní hodnota uchovává informaci, zda byl nástroj změněn během období, kdy byl skryt. V implementaci je nazývána jako takzvaný *dirty bit*. Při inicializaci instancí nástrojů je pro každou instanci modulu vytvořen záznam v tomto poli s hodnotou nastavenou na `false`. Pokud byl změněn některý ze vstupů komponenty nástroje s výjimkou atributu `enabled`, je nástroj zpracováván standardní cestou nahrazením instance v seznamu manažera nástrojů. Optimalizační krok přichází až ve chvíli, kdy má dojít k překreslení objektu mapy. Pokud jsou vizuální elementy nástroje aktuálně skryty, je nástroj vyhledán dle identifikátoru a jeho *dirty bit* je nastaven na hodnotu `true` (viz obrázek 6.6). Díky tomu je uchována informace, že mapa musí být překreslena až ve chvíli, kdy bude nástroj znovu zobrazen. Hodnota *dirty bit* je rovněž resetována pro všechny dvojice při každém překreslení tematické mapy, neboť aktuální stav všech nástrojů byl inicializován v objektu mapy a promítnut do obsahu zobrazení.

Na počátku procesu změny zobrazení nástroje je potřeba, aby komponenta nástroje zachytila událost o modifikaci vstupu `enabled`. Detekci změny má na starosti vlastní React Hook `useDidToolEnabledUpdate`, který reaguje na změnu atributu `enabled` objektu *props*. Pokud uvedený atribut nabude jiné hodnoty, je vyvoláno zpětné volání `onToolChange`, přičemž je (v podobě druhé argumentu) předána informace o změně pravdivostní hodnoty pro zobrazení (vyjádřeno konstantou `ENABLED_PROP`). V těle funkce je pro detekci změn využita již dříve popsaná metoda `useDidUpdateEffect`.

Obslužná metoda `handleToolChange` zachytí volání a díky specifikovanému argumentu ví, že došlo ke změně parametru `enabled`. Pokud nástroj, jehož vstup byl změněn, vytváří vrstvu tematické mapy (implementuje rozhraní `ILayerTool`), předá dále řízení metodě `toolSetEnabled`, v opačném případě se pokračuje ve standardním zpracování nástroje. Tento postup je znázorněn na obrázku 6.6.

Metoda `toolSetEnabled` je zodpovědná za dynamické skrývání či zobrazování nástrojů. Uživatel má možnost měnit povolení zobrazení vrstvy nástroje programově nebo prostřednictvím grafického uživatelského rozhraní záložky postranního panelu. Pokud uživatel má ve svém zdrojovém kódu přítomnou konfiguraci komponenty postranního panelu, je tak nutné změnit stav i této komponenty. Metoda tedy nalezne záložku příslušící danému nástroji a voláním funkce `setChecked` z aplikačního rozhraní Geovisto zajistí přepnutí všech závislostí. Dále je pro nástroj získán *dirty bit* a v návaznosti na jeho hodnotu je rozhodnuto, zda je možné vrstvu nástroje dynamicky vykreslit nebo je nutné nástroj inicializovat a vykreslit tak mapu jako celek. Popsaný proces rozhodování je vizualizován na následujícím schématu 6.6:



Obrázek 6.6: Předávání řízení při aktualizaci atributu `enabled`

Zelenou barvou jsou na obrázku 6.6 vyznačeny optimalizační kroky, které v implementaci byly učiněny s cílem omezit nutnost překreslení. Je možné pozorovat, že pokud funkce `toolSetEnabled` může dynamicky pracovat s obsahem Leaflet mapy, může narazit na zamítnutí přístupu k potřebným zdrojům.

Pokaždé, když dojde k překreslení mapy, kořenová komponenta `GeovistoMap` odpovědná za překreslení vrací svým potomkům aktuální objekt mapy. Komponenta `ToolGroup` z něj extrahuje manažer nástrojů a ten ukládá jako svůj stav. Uložení stavu (užitím metody `useState`) je v Reactu realizováno asynchronně, což otvírá prostor pro riziko nekonzistentního stavu, pokud na něm závisí další mechanismy. Pokud se změnil vstup některého z nástrojů a v krátkém časovém intervalu byl změněn i atribut `enabled`, může dojít k problému, kdy metoda `toolSetEnabled` bude přistupovat k zastaralé instanci nástroje, která již v nové konfiguraci mapy neexistuje. K vyřešení uvedeného problému byl zaveden „zámek“, který znemožňuje přístup k manažeru nástrojů, pokud je momentálně v procesu aktualizace.

Zámek je realizován jako globální proměnná nesoucí pravdivostní hodnotu `true`/`false` indikující, zda je možné ke stavu komponenty přistoupit či nikoliv. Než je tedy aktualizován manažer nástrojů, je zámek nastaven na hodnotu `true` a pokud by v tuto chvíli chtěla metoda `toolSetEnabled` dynamicky měnit zobrazení nástroje, je jí znemožněn přístup. Místo toho vloží požadavek na změnu zobrazení do fronty `enabledToolQueue` a ve chvíli, kdy bude asynchronní změna stavu komponenty `ToolGroup` dokončena, jsou všechny požadavky provedeny.

Ve zdrojovém kódu je tento mechanismus realizován užitím React Hook `useEffect`, který reaguje na změnu manažera nástrojů. Ve chvíli, kdy se stav změní, komponenta může

pracovat s aktuálními daty. Fronta `enabledToolQueue` je tak postupně vyprazdňována a požadavky na zobrazení zpracovány.

Optimalizace překreslení je dostupná i při aktualizaci vstupů komponenty `SidebarTool`. Pokud postranní panel změnil svůj stav zobrazení a má být skryt, jsou prostřednictvím aplikačního rozhraní knihovny Geovisto odstraněny Leaflet elementy z objektového modelu dokumentu. Uživatel ve svém obsahu tak již ten nástroj nevidí, současně však nebylo nutné znovu renderovat mapu jako celek.

Modrou barvou je na obrázku 6.6 zvýrazněn stav, kdy musí být překreslena zobrazená mapa. Došlo ke změně atributu `enabled` z hodnoty `false` na hodnotu `true`, instance nástroje tak byla doposud v zobrazení skryta a nově má být uživateli zobrazena. Současně ale během toho, co byl nástroj uživateli skryt, došlo ke změně některého z jeho dalších parametrů. Hodnota příslušného bitu v poli `toolDirtyBitArray` je nastavena na `true`. Nástroj je nezbytné znovu zavést s novou konfigurací a je nutné zobrazenou tematickou mapu překreslit. Toho je docíleno voláním obsluhy `handleToolChange`, již však bez uvedeného druhého argumentu specifikujícího, který vstup byl změněn. Nástroj tak bude zpracován standardní cestou nahrazením instance v manažeru nástrojů a bude provedeno překreslení mapy.

Všechny uvedené optimalizační mechanismy popsané v této sekci však mohou být diskreditovány nevhodným použitím knihovny uživatelem. Pokud programátor ve svém zdrojovém kódu deklaruje funkci jako některý ze vstupů komponent knihovny, dochází při každé změně jakéhokoliv aspektu rodičovské komponenty (včetně změny vstupů jiného potomka) k vyvolání deklarované funkce, což způsobuje změnu objektu komponenty *props*. Jedná se o implicitní chování knihovny React a implementovaná sada komponent ho respektuje. Každá komponenta nástroje v mém řešení reaguje na změny svých vstupů a vyvolává proces obsluhy. Ve scénáři, kdy by uživatel deklaroval funkci jako parametr nástroje, by při změně jakéhokoliv vstupu libovolné konfigurační komponenty docházelo k překreslení celé mapy, tedy i v případech, kdy by to teoreticky nebylo nutné. Obecným doporučením pro použití funkcí jako atributů komponent je využití React Hook `useMemo`, který ukládá do proměnné návratovou hodnotu funkce a je schopen hodnotu aktualizovat při změně některé ze závislostí. Proměnná je vzápětí předána komponentě jako vstup.

6.4 Uživatelské moduly

Jak již popisovala kapitola 4, knihovna Geovisto staví na modulární architektuře, přičemž poskytuje uživateli možnosti si přidat vlastní rozšíření včetně možnosti implementace vlastních nástrojů. Za tímto účelem bylo nutné vytvořit komponentu `CustomTool`, která dokáže zpracovávat jakékoliv vstupy předávané v objektu *props* komponenty. Nevýhodou je v tomto případě ztráta typové kontroly, neboť nelze předem určit jaké parametry vlastní nástroj očekává.

Knihovna Geovisto implicitně předpokládá, že vlastní nástroj uživatele implementuje minimálně rozhraní `IMapTool`, mé řešení tak tento předpoklad přebírá a uvažuje naprogramované veškeré chování specifikované tímto rozhraním. Nástroj tak musí disponovat například identifikátorem a proměnnou `enabled` určující, zda je zobrazení nástroje ve vykreslené mapě povoleno.

Aby bylo možné takový nástroj přidat do objektu mapy, je nutné jeho vytvořenou instanci zahrnout mezi ostatní nastavené nástroje, které manažer nástrojů spravuje. Vzhledem k tomu, že ale o modulu uživatele nejsou předem známy žádné informace, musí programátor předat komponentě informaci o vytvoření instance. Komponenta `CustomTool` z uvedeného

důvodu vyžaduje povinný parametr `createTool`, který očekává od programátora předanou funkci vytvářející instanci nástroje, demonstrovanou na příkladu 6.2.

```
<CustomTool
    id="custom-image-layer-tool"
    enabled={true}
    url="https://pathtoimage.com/px5ppBp.png"
    bounds={[[37.3374, -5.5017], [57.4484, 47.5482]]}
    createTool={(props) => new ImageLayerTool(props)}
/>
```

Výpis 6.2: Příklad použití CustomTool komponenty

Jak je možné vidět na příkladu 6.2, vlastní nástroj může mít různá *props*, kde v tomto případě jsou zde demonstrovány vlastnosti `url` a `bounds`. Na rozdíl od oficiálně poskytovaných modulů tak musí být mechanismus pro detekování změn nastavitelných parametrů řešen jiným způsobem, neboť nelze předem specifikovat závislosti některému z vlastních React Hooks implementovaných knihovnou, jako je tomu u ostatních komponent.

Řešením je reagovat na veškeré změny objektu *props*. Na změnu tohoto objektu reaguje React Hook `useMemo`, ve kterém jsou vyextrahovány a uloženy do proměnné pouze hodnoty představující konfigurovatelné parametry nástroje. Eliminovány jsou pomocné mechanismy knihovny (parametry `onToolChange`, `createTool`) i atributy `id` a `enabled`. Ty jsou, obdobně jako u ostatních nástrojů, zachycovány odděleně, neboť je reakce na jejich změnu a zpracování odlišné.

Na změny vyexportovaných hodnot dále reaguje kód, který zajišťuje porovnání extrahovaných parametrů s aktuálním stavem komponenty, ve kterém jsou uloženy parametry z předchozího volání. Změna vstupů komponenty je tak detekována pomocí porovnání dvou objektů.

Za tímto účelem byl doinstalován npm modul *deep-equal*[2], který umožňuje porovnání objektů pomocí jeho hodnot, nikoliv referencí do paměti počítače, a to včetně komplexních objektů, funkcí a několika úrovní zanoření. Užitím zmíněného porovnání pomocí externí knihovny je docíleno, že uživatel není limitován určitými typy hodnot parametrů vlastních nástrojů, ale může předávat i komplexní datové typy různě zanořené objekty i funkce.

Pokud výsledek porovnání značí, že se uživatelské vstupy komponenty `CustomTool` změnily, je vyvolán callback `onToolChange`, jako je tomu u ostatních komponent. Po zachycení volání komponentou `ToolGroup` se s ní pracuje stejně jako s oficiální komponentou poskytovanou autory knihovny Geovisto. Podporována je i optimalizace v podobě změn parametrů bez nutnosti překreslení mapy, stejně jako v případě oficiálně nástrojů poskytovaných autory knihovny Geovisto, nástroj však i v tomto případě musí implementovat minimálně rozhraní `ILayerTool`.

[2]https://www.npmjs.com/package/deep-equal

Kapitola 7

Testování

Vytvořená sada komponent, jež byla předmětem této práce, byla po celý čas vývoje testována. Snahou testování bylo ověřit splnění požadavků stanovených v sekci 4.4, kontrola realizace návrhu popsaného v kapitole 5 a ladění implementovaného zdrojového kódu, jehož popis byl předmětem předchozí kapitoly. S každou přidanou dílčí částí byla podrobně testována její funkčnost, v delších časových intervalech zdrojový kód jako celek, zejména činnost ostatních mechanismů, které mohla změna ovlivnit. Popsaná kontrola správného chování knihovny byla prováděna programátorem.

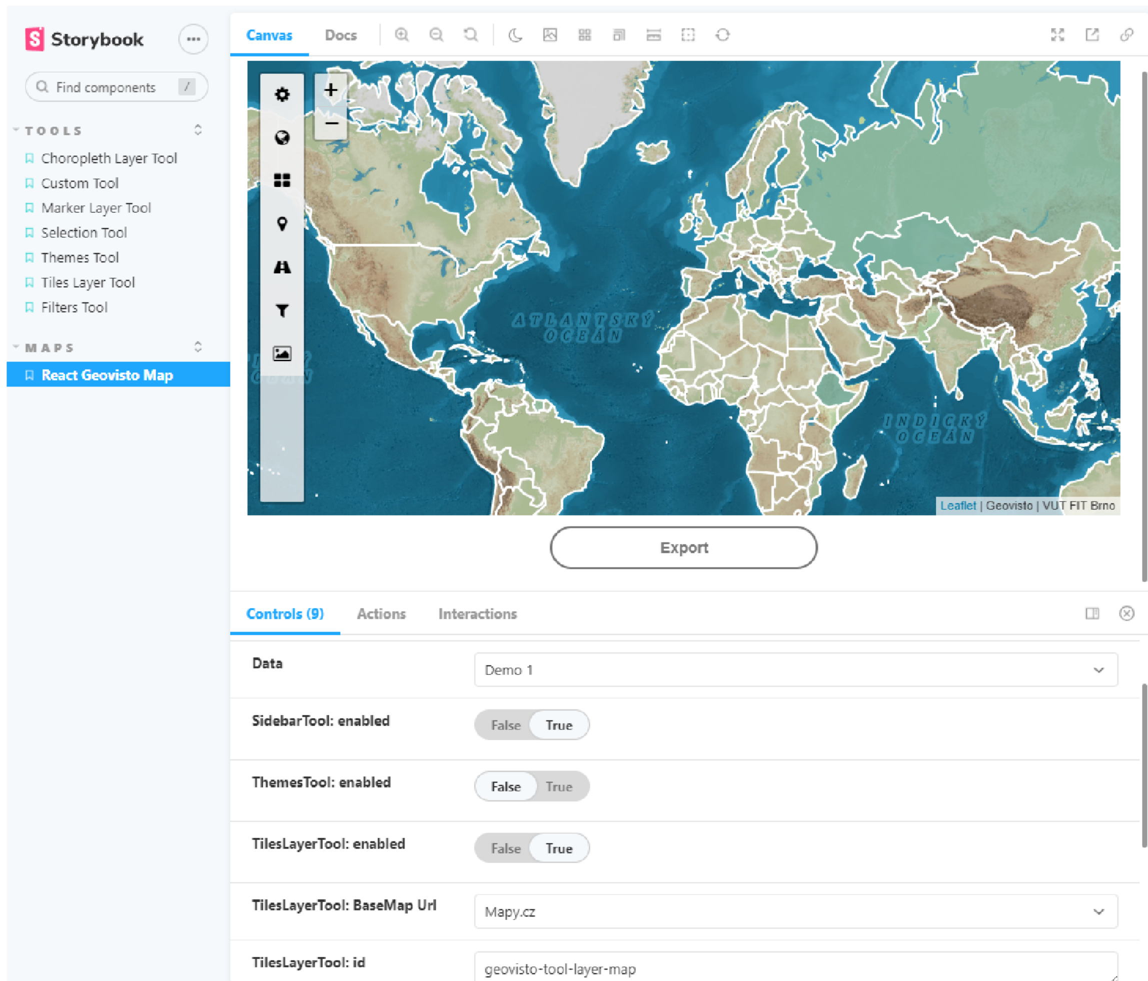
Vyvíjený zdrojový kód byl nejprve implementován s přímou závislostí na plnou implementaci jádra knihovny Geovisto[1] a všech oficiálně poskytovaných modulů. Tento přístup umožnil při vývoji důkladné ladění zdrojového kódu a experimenty s úpravami implementace zmíněné knihovny, zejména pak úpravy modifikátorů přístupu. Během testování přitom byla díky tomuto postupu odhalena a následně opravena implementační chyba knihovny Geovisto způsobující selhání procesu inicializace fragmentů prostřednictvím *props*. Později byl extrahován pouze zdrojový kód implementovaného řešení a byla přidána závislost na npm balíček knihovny Geovisto[2] a všechny dostupné moduly.

Pro účely testování byla využita knihovna Storybook, blíže představena v kapitole 3.3.1. S užitím tohoto pomocného nástroje byly vytvořeny odlišné varianty počáteční konfigurace mapy demonstrující různá nastavení přítomných modulů. Pro každý modul byla vyvinuta ve zdrojovém kódu jedna reprezentace (v podobě takzvané *story*), která umožňuje podrobně testovat možnosti nastavení komponent jednotlivých nástrojů. Na obrázku 7.1 je možné vidět snímek z prostředí nástroje Storybook, kde v levém panelu jsou záložky pro testování jednotlivých modulů. Pro každý modul lze ve spodní liště nastavit parametry, které jsou pro danou komponentu (a tedy i instanci nástroje) platné.

Správné chování komponent při dynamických změnách jejich vstupů tak byly simulovány a otestovány prostřednictvím změn vstupů definovaných v ovládacím panelu grafického rozhraní nástroje Storybook. Současně bylo ověřeno korektní chování nástrojů a dynamické změny vstupního objektu *props* a jeho vlastností ve zdrojovém kódu za použití proměnných a funkcí. Na snímku obrazovky 7.1 je zachycena testovací konfigurace pro souhrnné nastavení tematické mapy.

Prostředí bylo doplněno o tlačítko umožňující export současného stavu mapy ve formátu JSON do konfiguračního souboru.

[1]https://github.com/geovisto/geovisto-map
[2]https://www.npmjs.com/package/geovisto



Obrázek 7.1: Ukázka souhrnné konfigurace mapy využité při testování komponenty

V rámci provedeného testování byly vyzkoušeny různé kombinace přítomných komponent včetně závislostí pořadí, ve kterých jsou ve zdrojovém kódu uvedeny. Dle očekávání různé kombinace a pořadí neovlivňují očekávanou funkčnost výsledné aplikace. Byly však zjištěny nedostatky v případě, kdy uživatel v kódu uvedl komponentu `ToolGroup`, současně jí ale jako potomky nezadal žádné validní elementy. Tento scénář je demonstrován na následujícím příkladě:

```
<GeovistoMap
    vstupy
>
    <ToolGroup>
    </ToolGroup>
</GeovistoMap>
```

Uvedený zdrojový kód vedl k tomu, že se mapa nikdy neinicializovala, neboť proces inicializace spouští poslední zpracovaný nástroj. Tato chyba byla opravena a v aktuální verzi knihovny je proces zavedení mapy spuštěn i pokud element `ToolGroup` v kódu existuje a není specifikovaný žádný nástroj.

Ověřen byl i případ, kdy uživatel vloží mezi otevírací a uzavírací značky komponent neplatné elementy (elementy nereprezentující předdefinovanou komponentu žádného z nástrojů). V takovém případě je skutečnost zapsána do vývojářské konzole.

Velký vliv na chování aplikace má přítomnost komponenty `SidebarTool` (a vnořených elementů `SidebarTab`) v konfiguraci tematické mapy, neboť v určitých případech musí při změně nástroje být změny promítnuty i do zobrazení postranního panelu. Panel však není povinnou součástí konfigurace mapy, a tak kód nesmí být na jeho přítomnosti závislý, byly tak testovány změny vstupů komponent v obou případech, kdy postranní panel byl i nebyl v kódu specifikován.

Testování byly podrobeny i případy, kdy došlo ke změně identifikátoru nástroje. V takovém případě musí být nástroj se starým identifikátorem odstraněn z objektu mapy, nahrazen novou instancí a za předpokladu, že je přítomen i postranní panel, provést jeho překreslení. Synchronizaci změny identifikátoru u nástroje a příslušné záložky postranního panelu si musí řešit vývojář sám. Dojde-li ke změně identifikátoru nástroje a atribut `tool` náležící komponenty `SidebarTab` se nezmění, po vykreslení nebude záložka postranního panelu zobrazena – nenalezne v kódu nástroj, ke kterému by dle identifikátoru náležela. Popsaný mechanismus se ukázal v implementované knihovně jako plně funkční.

Velmi podrobně jsem se věnoval ověřování funkcionality dynamického překreslování, které bylo zavedeno jako optimalizační krok. K simulaci případu, kdy nemá dojít k okamžitému překreslení mapy, byla použita vrstva `TilesLayerTool`, jejíž zobrazení bylo prostřednictvím proměnné zakázáno a dynamicky byla měněna podkladová mapa (atribut `baseMap`). K překreslení mapy došlo až ve chvíli, kdy bylo zobrazení nástroje opět povoleno, což dokazuje, že zmíněný optimalizační krok funguje.

Chybu v implementaci odhalilo použití podmíněného osazení komponenty do objektového modelu dokumentu způsobem demonstrovaným na následujícím příkladě:

```
{condition &&
    <TilesLayerTool id="base-map" enabled={true}/>
}
```

Vytvářená knihovna při inicializaci přímo pracuje s množstvím potomků a změna jejich počtu vedla k nesprávnému chování. Po objevení chyby byl implementován mechanismus, který změnu počtu potomků ošetřuje.

Podrobeny testování byly i případy užití sady komponent při nesprávném nakládání s knihovnou. Příkladem mohou být stejné identifikátory různých instancí nástrojů nebo hodnota `undefined` na pozici identifikátoru, tedy v případě, kdy uživatel identifikátor nespecifikuje. Jsou-li identifikátory více instancí nástrojů shodné, knihovna dále uvažuje pouze komponentu specifikovanou v pořadí na první pozici a všechny další komponenty se stejným identifikátorem ignoruje. Vývojáře informuje o této skutečnosti prostřednictvím konzole prohlížeče. Popsaná situace splňuje předpokládané chování. V případě chybějícího identifikátoru nástroje knihovna dle očekávání kontrolovaně havaruje a vypíše uživateli na obrazovku chybové hlášení o nenalezeném parametru `id`.

Uživatel může rozšířit nástroje knihovny Geovisto o vlastní moduly a tento fakt bylo potřeba simulovat. Z uvedeného důvodu byl implementován vlastní nástroj `ImageLayerTool` přidávající vrstvu s obrázkem do výsledného vykreslení mapy. Nástroj očekává kromě povinných parametrů dva vlastní vstupy pro zdrojovou adresu obrázku a jeho pozici v mapě. Díky tomu mohla být ověřena funkčnost komponenty `CustomTool` včetně ověření, že v rámci interního fungování knihovny je s uživatelským modulem nakládáno stejně jako s oficiálně podporovaných nástrojem publikovaným autory knihovny Geovisto.

Konfigurace objektu tematické mapy a jednotlivých modulů užitím implementované sady komponent v prostředí nástroje Storybook (obrázek 7.1) slouží současně jako návod pro programátory, jak knihovnu správně používat. V grafickém uživatelském prostředí aplikace navíc může pro každý nástroj nalézt záložku s vygenerovanou dokumentací, která podrobněji popisuje jednotlivé nastavitelné vlastnosti nástroje. Uživatel může spustit připravené příklady pomocí příkazu `npm start` v kořenovém adresáři repositáře řešení.

Kapitola 8

Závěr

Cílem této diplomové práce bylo vytvořit programovou vrstvu abstrakce efektivně propojující zdrojový kód webové aplikace uživatele s možnostmi vytváření a konfigurace tematických map, které poskytuje knihovna Geovisto. Hlavní motivací bylo usnadnit uživateli nakládání s knihovnou a zjednodušit proces zahrnutí tematické mapy do projektu vývojáře využívajícího aplikační rámec pro tvorbu uživatelských rozhraní.

K dosažení požadovaného výstupu bylo nejprve potřeba se seznámit problematikou vytváření tematických map v moderních aplikacích. Před započetím implementace bylo žádoucí pochopení kontextu a vznikla tak potřeba podrobně prostudovat knihovnu Geovisto zabývající se vytvářením tematických map a pochopit její fungování na implementační úrovni. Vykreslení grafických prvků do výsledného zobrazení zajišťuje knihovna Leaflet, bylo tedy nezbytné se seznámit i s používáním aplikačního rozhraní uvedené knihovny. Pro realizaci implementační stránky práce bylo nezbytné provést podrobnou analýzu technologií používaných k vytváření uživatelských rozhraní moderních webových aplikací a vybrat nejvhodnějšího kandidáta pro výsledné řešení. Na základě provedeného srovnání byl vybrán aplikační rámec React, jehož praktiky si bylo nutné důkladně osvojit praktiky, zejména z důvodu, že cíl práce využívá koncept komponent vcelku netradičním způsobem. K vytvoření testovacího grafického rozhraní bylo zapotřebí porozumět konceptu a nabýt vědomosti o používání pomocného nástroje Storybook.

Záměr práce se podařilo naplnit a výsledkem je tak funkční sada komponent realizovaná pro aplikační rámec React. Pomocí vytvořených komponent může programátor vytvářet a konfigurovat tematické mapy a to zejména prostřednictvím vytváření a specifikace parametrů instancí jednotlivých modulů. Komponenty jsou schopné reagovat na dynamické změny svých vstupů a iniciovat překreslení zobrazeného grafického obsahu ve chvíli, kdy je to nezbytné. V rámci implementace bylo zavedeno několik optimalizačních mechanismů, které si kladou za cíl uživatelskou zkušenost s knihovnou Geovisto vylepšit. Vyvinutá knihovna komponent splňuje kladené požadavky plnit funkci prostředníka mezi programátorem a knihovnou Geovisto a jakkoliv neomezovat možnosti personalizace při nakládání s knihovnou. Současně je implementovaný zdrojový kód strukturován způsobem, který umožňuje snadné rozšíření o nově podporované moduly a zvyšuje tak udržitelnost knihovny.

Vytvořená vrstva abstrakce byla realizována pro aplikační rámec React s úmyslem zacílit na co největší množinu programátorů. Práce má potenciál stát se předlohou pro implementaci obdobné sady komponent pro další z populárních webových frameworků a bylo by vhodné programátorům usnadnit práci s knihovnou Geovisto i za předpokladu že využívají jiný ze zástupců aplikačních rámců pro tvorbu uživatelského rozhraní, tedy například Angular či Vue.js.

Literatura

[1] Artem Sapegin. *Getting started with React Styleguidist* [online]. 2021 [cit. 2021-11-18]. Dostupné z: https://react-styleguidist.js.org/docs/.

[2] Bit community. *How bit Works?* [online]. 2021 [cit. 2021-11-18]. Dostupné z: https://docs.bit.dev/docs/how-bit-works.

[3] Boduch, A. a Derks, R. *React and React Native: A complete hands-on guide to modern web and mobile development with React.js, 3rd Edition*. Packt Publishing, 2020. ISBN 9781839212437.

[4] Butler, H., Daly, M. a Doyle, A. *The GeoJSON Format*. RFC 7946. RFC Editor, srpen 2016. Dostupné z: https://www.rfc-editor.org/rfc/rfc7946.txt.

[5] Dan Abramov and Andrew Clark. *Redux: Getting started with Redux* [online]. 2021 [cit. 2021-11-09]. Dostupné z: https://redux.js.org/introduction/getting-started.

[6] Data-Driven Documents. *D3-geo* [online]. 2021 [cit. 2021-12-11]. Dostupné z: https://github.com/d3/d3-geo.

[7] Data-Driven Documents. *Home* [online]. 2021 [cit. 2021-12-10]. Dostupné z: https://github.com/d3/d3/wiki.

[8] Dent, B., Torguson, J. a Hodler, T. *Thematic map design*. 6. vyd. McGraw-Hill New York, New York, NY, 2009 [cit. 2021-11-22]. ISBN 978–0–07–294382–5.

[9] Dominic Gannaway. *Inferno: Get Started* [online]. 2021 [cit. 2021-11-08]. Dostupné z: https://www.infernojs.org/docs/.

[10] Dominic Nguyen. *Storybook vs Styleguidist* [online]. Chromatic, Květen 2018 [cit. 2021-11-17]. Dostupné z: https://www.chromatic.com/blog/storybook-vs-styleguidist/.

[11] Environmental Systems Research Institute, Inc.. *ESRI Shapefile Technical Description* [online]. červenec 1998 [cit. 2021-11-28]. Dostupné z: https://www.esri.com/content/dam/esrisites/sitecore-archive/Files/Pdfs/library/whitepapers/pdfs/shapefile.pdf.

[12] Evan You. *Vue.js: Introduction* [online]. 2021 [cit. 2021-11-02]. Dostupné z: https://v3.vuejs.org/guide/.

[13] Facebook Inc.. *React: Getting Started* [online]. 2021 [cit. 2021-10-30]. Dostupné z: https://reactjs.org/docs/getting-started.html.

[14] Flanagan, D. *JavaScript: The Definitive Guide*. O'Reilly Media, 2011. Definitive Guides. ISBN 9780596805524.

[15] Frisbie, M. *Professional JavaScript for Web Developers*. Wiley, 2019. ISBN 9781119366447.

[16] Google. *Introduction to the Angular Docs* [online]. 2021 [cit. 2021-10-31]. Dostupné z: https://angular.io/docs.

[17] Hynek., J., Kachlík., J. a Rusňák., V. Geovisto: A Toolkit for Generic Geospatial Data Visualization. In: INSTICC. *Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - IVAPP,*. SciTePress, 2021, s. 101–111. DOI: 10.5220/0010260401010111. ISBN 978-989-758-488-6.

[18] International Organization for Standardization. *ISO 19136* [online]. 2020 [cit. 2021-11-27]. Dostupné z: https://www.iso.org/standard/75676.html.

[19] International Organization for Standardization. *ISO 3166* [online]. 2020 [cit. 2021-11-25]. Dostupné z: https://www.iso.org/iso-3166-country-codes.html.

[20] Kraak, M. a Ormeling, F. *Cartography: Visualization of Geospatial Data*. Prentice Hall, 2003 [cit. 2021-11-25]. ISBN 9780130888907.

[21] Kumar, A. a Chaudhary, M. *Practical jQuery*. Apress, 2015. ISBN 9781484207871.

[22] Leaflet. *Leaflet API reference* [online]. 2021 [cit. 2021-11-22]. Dostupné z: https://leafletjs.com/reference-1.7.1.html.

[23] Levlin, M. *DOM benchmark comparison of the front-end JavaScript frameworks React, Angular, Vue, and Svelte*. 20500 Turku, Finsko, 2020. Diplomová práce. Åbo Akademi University.

[24] MacEachren, A. a Kraak, J. Research challenges in geovisualization. Cartography Geograph. *Inf. Sci.* Leden 2001, sv. 28, s. 3–12, [cit. 2021-11-24].

[25] Macrae, C. *Vue.js: Up and Running: Building Accessible and Performant Web Apps*. O'Reilly Media, 2018. ISBN 9781491997215.

[26] Mardan, A. *React Quickly: Painless web apps with React, JSX, Redux, and GraphQL*. Manning, 2017. ISBN 9781638353966.

[27] Mariano, C. L. *Benchmarking javascript frameworks*. Dublin, Ireland, 2017. Disertační práce. Technological University Dublin.

[28] Mikowski, M. a Powell, J. *Single Page Web Applications: JavaScript end-to-end*. Manning Publications, 2013. ISBN 9781617290756.

[29] Mohammed, Z. *Angular Projects: Build nine real-world applications from scratch using Angular 8 and TypeScript*. Packt Publishing, 2019. ISBN 9781838550387.

[30] Patel, T. *Does Your Web Application Need A Front-End Framework?* [online]. Third Rock Techkno, Květen 2020 [cit. 2021-11-03]. Dostupné z: https://www.thirdrocktechkno.com/blog/does-your-web-application-need-a-front-end-framework.

[31] Petukhova, E. Sitecore JavaScript Services Framework Comparison. 20500 Turku, Finsko: [b.n.]. 2019.

[32] Precht, P. *Understanding Zones* [online]. thoughtram, Leden 2016 [cit. 2021-11-11]. Dostupné z: https://blog.thoughtram.io/angular/2016/01/22/understanding-zones.html.

[33] Rich Harris. *Svelte* [online]. 2021 [cit. 2021-11-08]. Dostupné z: https://svelte.dev/docs.

[34] Richey, B., Yu, R., Vegh, E., Despoudis, T., Punith, A. et al. *The The React Workshop: Get started with building web applications using practical tips and examples from React use cases*. Packt Publishing, 2020. ISBN 9781838821661.

[35] Ryan Carniato. *SolidJS: Getting Started* [online]. 2021 [cit. 2021-11-08]. Dostupné z: https://www.solidjs.com/guide.

[36] Satrom, B. Choosing the Right JavaScript Framework for Your Next Web Application. *Prog./Kendo UI*. 2018, s. 34.

[37] Shrestha, S. *Ember. js front-end framework–SEO challenges and frameworks comparison*. 00520 Helsinki, Finsko, 2015. Bakalářská práce. Haaga-Helia University of Applied Sciences.

[38] Stack Overflow. *2021 Developer Survey* [online]. Červen 2021 [cit. 2021-11-02]. Dostupné z: https://insights.stackoverflow.com/survey/2021.

[39] Storybook community. *Introduction to Storybook for React* [online]. 2021 [cit. 2021-11-04]. Dostupné z: https://storybook.js.org/docs/.

[40] Syamkumar, M., Durairajan, R. a Barford, P. Bigfoot: A geo-based visualization methodology for detecting BGP threats. In: *2016 IEEE Symposium on Visualization for Cyber Security (VizSec)*. 2016, s. 1–8 [cit. 2021-11-29]. DOI: 10.1109/VIZSEC.2016.7739583.

[41] W^{3}Techs. *Usage statistics of client-side programming languages for websites* [online]. [cit. 2021-10-30]. Dostupné z: https://w3techs.com/technologies/overview/client_side_language.

[42] Yan Holtz. *Connection map* [online]. [cit. 2021-11-29]. Dostupné z: https://www.d3-graph-gallery.com/connectionmap.html.

Příloha A

Rozšíření knihovny o nový oficiálně podporovaný modul

Autoři a komunita knihovny Geovisto neustále rozšiřují funkcionalitu knihovny o nové moduly pro konfiguraci tematické mapy. Tato příloha popisuje kroky nutné pro aktualizaci implementované sady komponent za účelem přidání podpory nového nástroje. Pro přidání podpory nového modulu je nezbytné implementaci rozšířit následujícími kroky:

- Vytvoření nové komponenty – je zapotřebí založit nový soubor s příponou `.tsx` v adresáři `src/react/components` a vytvořit kostru zdrojového kódu komponenty. Zavedená konvence pojmenování komponent respektuje názvy jednotlivých modulů, pouze bez prefixu „Geovisto“.
- Do těla nově vytvořené komponenty zasadit následující *React Hooks* implementované knihovnou:
  - `useToolEffect` – se specifikovanými závislostmi na parametry objektu *props* dle implementace konkrétního modulu, vyjma níže uvedených
  - `useDidToolEnabledUpdate` – se závislostí na parametr `enabled` objektu *props*
  - `useDidToolIdUpdate` – se závislostí na parametr `id` objektu *props*
- Přidání typu komponenty do seznamu podporovaných komponent reprezentovaného konstantou `supportedComponentTypes` (soubor `src/react/Constants.ts`).
- Rozšíření konstrukce `switch` v metodě `getToolInstance` o informaci o vytváření nové instance nástroje pro daný modul.
- Export nově vytvořené komponenty v souboru `index.ts` umístěného v adresáři `src/react/components/`.