Univerzita Pardubice

Fakulta elektrotechniky a informatiky

# Webový 3D simulátor těles ve vesmíru

Michal Struna

Bakalářská práce

2019

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne ?. ?. 2019

Michal Struna

## Poděkování

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

## ANOTACE

Bakalářská práce se v praktické části zabývá implementací webové aplikace pro 3D vizualizaci těles ve vesmíru. V rámci aplikace je kladen důraz na dynamický obsah, na kterém se mohou všichni uživatelé po úspěšně autentifikaci podílet. Veškerý obsah je možné pomocí serverového REST API upravovat. Teoretická část je zaměřena na popis technologií pro tvorbu webových aplikací, vysvětlení životního cyklu aplikace a uvedení řešení několika problémů spojených s implementací. Součástí teoretické části je také detailní popis všech stránek nacházejících se v aplikaci.



## KLÍČOVÁ SLOVA

webová aplikace, simulátor, vesmír, astronomie, TypeScript, 3D


## TITLE

Web 3D simulator of bodies in universe


## ANNOTATION

The bachelor thesis in practical parts deals with implementation of web applications for 3D visualization of bodies in space. Within the application, emphasis is placed on the dynamic content on which all users can participate after successful authentication. All content can be edited using the REST API. The theoretical part focuses on the description of technologies for creating web applications, explaining the life cycle of the application and introducing solutions to several problems connected with implementation. Part of the theoretical part is also a detailed description of all the pages that are in the application.



## KEYWORDS

web application, simulator, universe, astronomy, TypeScript, 3D

# OBSAH

# SEZNAM OBRÁZKŮ

# SEZNAM ZDROJOVÝCH KÓDŮ

# SEZNAM ZKRATEK

| | |
|---|---|
| 3D | Three Dimensional |
| API | Application Programming Interface |
| BSON | Binary JSON |
| CSS | Cascading Style Sheets |
| CRUD | Create, Read, Update, Delete |
| DOM | Document Object Model |
| HTML | Hypertext Markup Language |
| HTTP | Hypertext Transfer Protocol |
| JPEG | Joint Photographic Experts Group |
| JS | JavaScript |
| JSON | JavaScript Object Notation |
| PNG | Portable Network Graphics |
| REST | Representational State Transfer |
| SASS | Syntactically awesome style sheets |
| TS | TypeScript |
| UI | User Interface |
| URI | Uniform Resource Identifier |
| XML | Extensible Markup Language |

# ÚVOD

V dnešní době zažívá odvětví astronomie velký rozmach. Lze nalézt mnoho portálů, které nově objevené informace ihned zveřejňují. Často se však jedná o rozsáhlé monografie, které laikovi příliš neřeknou. Navíc ne vždy jsou dostupné v české lokalizaci. Na druhé straně existují 3D simulátory, které je ale nutno stahovat z internetu a poté instalovat. Tyto simulátory však obsahují pouze minimum informací a spíše než informační prostředek a komunitní portál slouží jako pouhá vizuální scéna.

Cílem této bakalářské práce, je vytvořit aplikaci, která poskytne jednoduchý pohled na astronomii lidem, kteří by se o tomto odvětví něco rádi dozvěděli. Díky obsáhlé databázi dat však nabízí i pohodlný a dostupný zdroj informací pro pokročilejší uživatele. Spojuje tak textové zdroje a grafické aplikace. Uživatel má možnost si libovolnou vlastnost libovolného tělesa zobrazit graficky před sebou a tuto vlastnost pak porovnat napříč všemi tělesy v databázi. Vše je dostupné v české lokalizaci a zároveň je celá aplikace online.

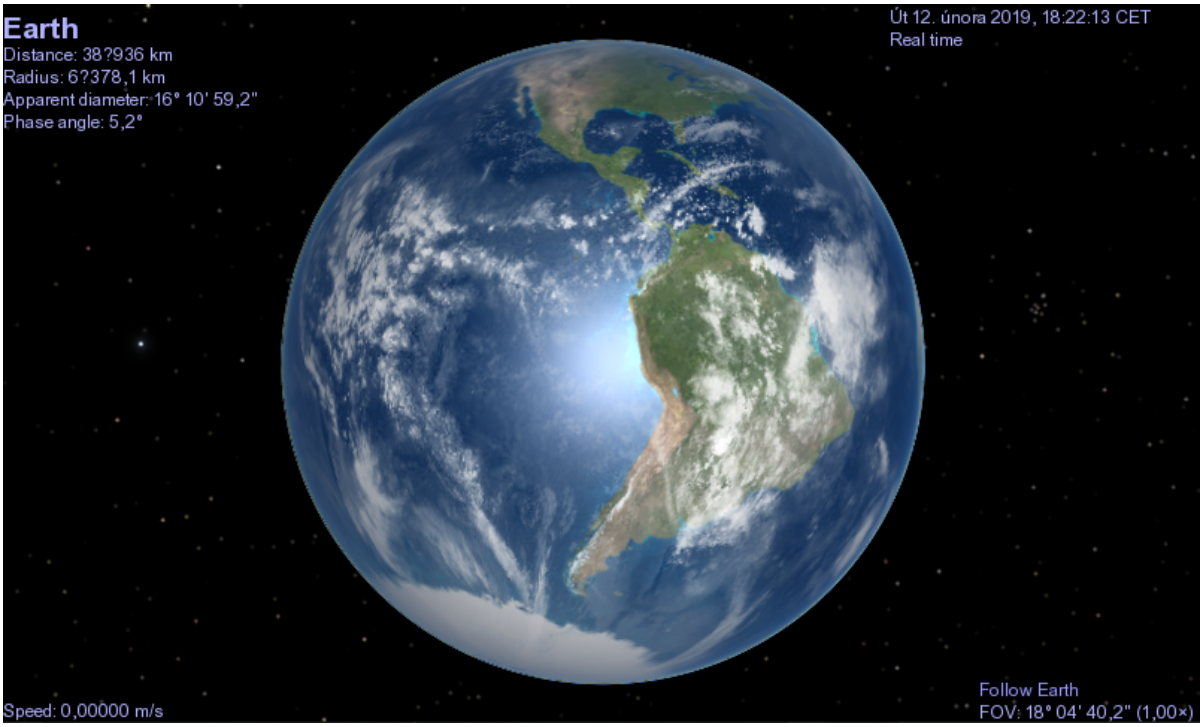
Obsah webové aplikace je plně dynamický a kdokoliv do něj může přispívat svými znalostmi. Všechny takto přidané informace však prochází schvalovacím procesem, kterého se účastní administrátor.

# 1 3D SIMULACE A JEJICH ŘEŠENÍ

## 1.1 Celestia

*Celestia* je *open-source* 3D program zobrazující tělesa ve vesmíru v reálném čase. Podobně jako tato bakalářská práce obsahuje databázi těles od těch nejmenších meteoritů až po celé galaxie. Vzhledem k tomu, že program Celestia vyvíjí komunita a nikoliv jednotlivec je tato databáze mnohem rozsáhlejší a lze si dodatečně stáhnout více jak 10 GB dat.

Program je vytvořen v *C++* a *Lua*, což mu dovoluje výkon počítače využívat podstatně lépe, než webová aplikace. Navzdory tomu je nutno tento program nejdříve nainstalovat, což může některé uživatele odradit. Je však dostupný jak pro *Windows*, tak i pro *Linux* a *macOS*. Neobsahuje ovšem češtinu.




Obrázek 1: Ukázka JPL Solar Simulator [1]

[1] Fotografie pořízena z `https://space.jpl.nasa.gov/`.

## 1.2 Stellarium

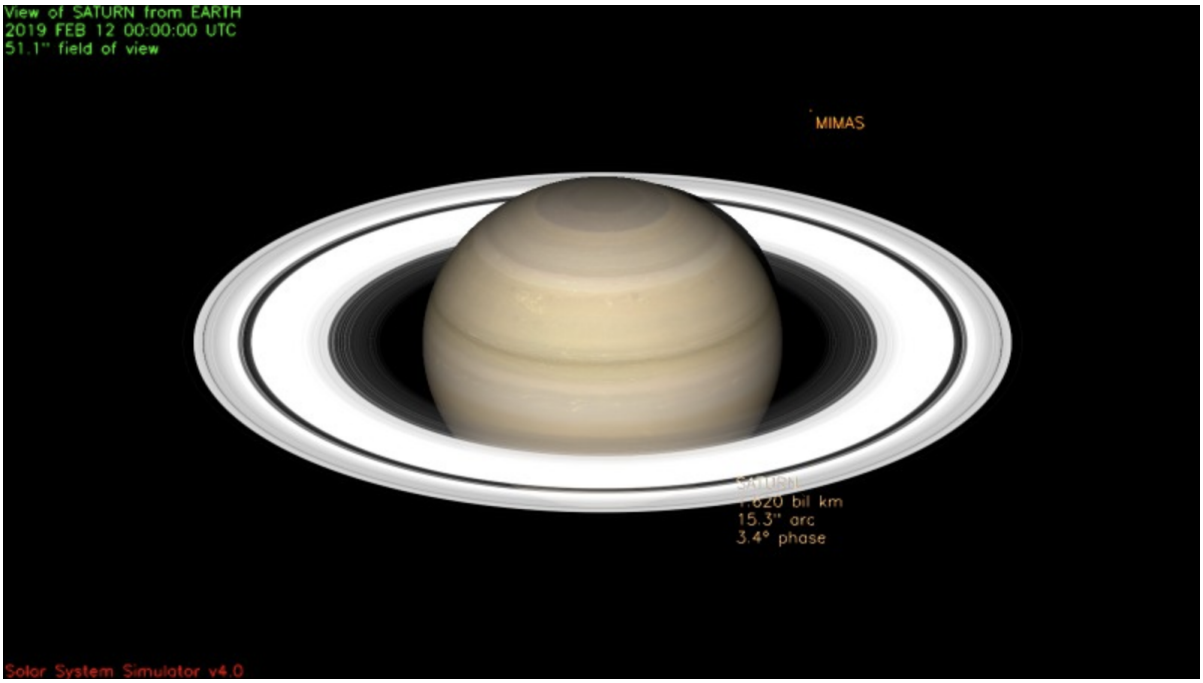
*Stellarium* je dalším z řady *open-source* astronomických programů. Není zde však možné se volně pohybovat prostorem. Program se specializuje na zobrazování oblohy, a to v jakoukoliv dobu jakéhokoliv dne z jakéhokoliv místa na Zemi.

Obsahuje velice rozsáhlou databázi obsahující v základní instalaci necelý milion těles. Je však možné si dodatečně nainstalovat katalogy s více jak 200 miliony tělesy. U každého tělesa je možné si zobrazit stručné informace, především co se jejich polohy a vzdálenosti od Země týče.

Jedná se o program napsaný v *C++*, což stejně jako u programu Celestia vede k vysokému výkonu, avšak nutnosti instalace. Je ovšem dostupný i v české lokalizaci a opět pro *Windows*, *Linux* i *macOS*.

## 1.3 NASA/JPL Solar System Simulator

Jedná se o webovou aplikaci, která umožňuje zobrazit libovolné těleso v databázi libovolnou dobu a to z libovolného jiného tělesa v databázi. Obsahuje pouze tělesa z naší sluneční soustavy. Umožňuje nastavit i velikost zorného pole ve stupních. Vygenerovaný pohled je ale bohužel pouze statický a nelze s ním už dál nijak interagovat.




Obrázek 2: Ukázka JPL Solar Simulator [1]

## 1.4 Space Engine

## 1.5 Universe Sandbox

[1]Fotografie pořízena z `https://space.jpl.nasa.gov/`.

# 2 POUŽITÉ TECHNOLOGIE

## 2.1 Jazyk TypeScript

*TypeScript* je programovací jazyk vyvinutý firmou *Microsoft*. Jedná se o nádstavbu jazyka *JavaScript*, která přidává statické typování a další vlastnosti objektového programování. Kód napsaný v jazyce *JavaScript* je kompatibilní s *typescriptovým* kódem. Pro kompatibilitu v prohlížečích je nutné veškerý *TypeScript* transpilovat do *javascriptového* kódu.

## 2.2 Knihovna React

Javascriptová knihovna *React*, jejíž autorem je *Facebook*, usnadňuje a zefektivňuje tvorbu *UI*. Přináší tzv. *one-way data binding*, které zaručuje okamžitou aktualizaci *UI* při změně stavu aplikace. *React* si vytváří vlastní virtuální *DOM*, který je na rozdíl od toho v *HTML* rychlejší. V tomto modelu pak React provádí všechny své operace. Teprve když je třeba provést změnu v prohlížeči, je třeba aktualizovat *HTML*.

### 2.2.1 Syntaxe JSX

Vytvářet zanořovací strukturu komponent může být nepřehledné. Proto se často s knihovnou *React* používá i syntaxe *JSX*. Ta umožňuje psát *javascriptový* kód v podobě *XML* tagů. *JSX* syntaxi je z důvodů kompatibility prohlížečů nutné transpilovat do nativního *JavaScriptu*.

```
React.createClass('MyComponent', {
    children: 'Hello ' + name + '!',
    className: 'block'
})

<MyComponent className='block'>
    Hello {name}!
</MyComponent>
```

Zdrojový kód 1: Porovnání vykreslení komponenty ve standardním JavaScriptu a v JSX

## 2.3 Knihovna Three.js

*Three.js* je javascriptová knihovna pro vytváření 3D grafiky. Využívá aplikačního rozhraní *WebGL*.

## 2.4 Framework Node.js

Framework *Node.js* umožňuje používání jazyka *JavaScript* na serveru. Narozdíl od *Javy* nebo *PHP* pracuje pouze s jediným vláknem a funguje na asynchronním neblokujícím zpracování požadavků. Jakmile je dokončen požadavek, jeho callback uvedený v argumentu se zařadí do fronty. Tzv. *event loop* pak zjišťuje, zda je zásobník zpracovávaných operací prázdný. Pokud ano, vloží do něj první callback z fronty. Tento cyklus se opakuje po celou dobu běhu serveru.

## 2.5 Databáze MongoDB

*MongoDB* je *NoSQL* multiplatformní dokumentová databáze s otevřeným zdrojovým kódem. Narozdíl od relačních databází používá dokumenty ve formátu *BSON*, který je podobný *JSON*. Při vytváření dokumentů si databáze automaticky vytváří vlastní unikátní ID. Uložené dokumenty lze vedle porovnávání hodnot vyhledávat na základě prvků v poli, podle rozsahu nebo podle regulárních výrazů. V *MongoDB* je možné indexovat libovolné pole dokumentů. Indexy jsou koncepčně stejn, jako v relačních databázích.

## 2.6 Knihovna Mongoose

Open-source knihovna *Mongoose* zjednodušuje práci s MongoDB, zejména pak vytváření schémat a validaci dat.

## 2.7 Framework Swagger

*Swagger* je nástroj pro vytváření *REST API*.

## 2.8 Nástroj Webpack

Prohlížečový *JavaScript* nativně nepodporuje rozdělování kódu do modulů (jako např. *Java* do balíčků nebo *C++* do jmenných prostorů). Jediným způsobem je importovat do

*HTML* větší množství *javascriptových* souborů pomocí tagu `<script>`. Tento postup je ale špatnou praktikou a na web má negativní dopady.

Díky nástroji *Webpack* je možné v *JavaScriptu* využívat modulární systémy jako *CommonJS*, *AMD* nebo *ES modules*. Je tak možné větší množství souborů propojit pomocí klauzulí `require` nebo `import`.

*Webpack* také umožňuje využívat tzv. *loadery* třetích stran. To vede k možnosti podobně načítat, slučovat či parsovat i soubory jiných typů, např. *CSS*, *SVG* nebo *JSON*. Zároveň za využití bohaté nabídky pluginů lze všechny tyto soubory modifikovat, např. převádět nový *JavaScript* verze *ES6* na starší, podporovaný prohlížeči.

```
{
    entry: './src/index.js',
    output: { filename: 'index.min.js' },
    module: {
        rules: [
            { test: /\.tsx?$/, use: ['ts-loader', 'babel-loader'] }
        ]
    }
}
```

Zdrojový kód 2: Konfigurace nástroje Webpack v souboru webpack.config.json

.

## 2.9 Verzovací systém Git

*Git* je systém správy verzí vytvořený *Linusem Torvaldsem*. Při vytváření nové verze dat vytvoří snímky všech souborů tak, jak v daný okamžik vypadají a na tyto snímky pak uloží reference. V případě, že se soubor nijak nemění se nevytváří nový snímek, ale pouze se nastaví reference na ten předchozí, který je identitický.

Pro spravované soubory používá tři stavy. Při změně souboru se nastaví jeho stav na *modified*. Stav *staged* znamená, že soubor byl označen k tomu, aby byl zapsán v další verzi. Zapsaný soubor je ve stavu *commited*. Všechny tyto změny jsou však pouze lokální. Pro umístění změn do vzdáleného repozitáře je třeba všechny zapsané soubory odeslat *(push)*. Ostatní s přístupem do repozitáře si pak mohou tyto změny stáhnout *(pull)*.

## 2.10 Balíčkovací systém NPM

*NPM* je správce balíčků pro serverový i klientský *JavaScript*. Řídícím souborem v projektu je *package.json*, který obsahuje informace o projektu a určuje, které balíčky jsou součástí projektu. Všechny balíčky jsou umístěny do adresáře *node_modules* v projektu. Součástí je také soubor *package-lock.json*, který zachovává verze nainstalovaných balíčků.

```
npm update // Update packages.
npm install // Install existing dependencies.
npm search mongo // Find out name of package.
npm install --save mongodb // Add new dependency.
npm uninstall --save-dev webpack // Remove dev dependency.
npm list // Show dependency tree.
```

Zdrojový kód 3: Ukázka práce s NPM

.

## 2.11 Preprocesor SASS

Preprocesor *SASS* přidává do *CSS* možnost zanořování selektorů, proměnné, funkce, podmínky a další vlastnosti zlepšující čitelnost kódu. Pro kompatibilitu s prohlížeči je nutné ho transpilovat do *CSS*.

# 3 NÁVRH A VÝVOJ APLIKACE

## 3.1 Struktura projektu

React nepřichází s žádnou doporučenou strukturou projektu. Proto došlo k vytvoření vlastní struktury. Vzhledem k velkému množství souborů je celý obsah projektu členěn do podadresářů a modulů.

- **dist**: Produkční sestavení aplikace. Veškerý obsah adresáře se generuje automaticky.
- **node_modules**: Externí knihovny pro serverovou část.
- **src**: Zdrojové soubory aplikace.
  - **Client**: Podprojekt, který obsahuje všechny klientské soubory.
    - **node_modules**: Externí knihovny pro klientskou část.
    - **src**: Zdrojové soubory klientské části.
      - **Controls**: Modul obsahující ovladače a tlačítka využitá v menu.
      - **Forms**: Modul umožňující tvorbu formulářů.
      - **Global**: Hlavičkové soubory dostupné v globálním prostoru.
      - **Panel**: Modul pro panel tělesa.
      - **System**: Hlavní modul obsahující systémové soubory.
      - **Universe**: Modul zprostředkovávající simulátor.
      - **User**: Modul s formuláři a další komponentami pro uživatele.
      - **Utils**: Pomocný modul.
  - **Constants**: Konstanty a konfigurace.
  - **Controllers**: Kontrolery s adresářovou strukturou popisující REST API.
  - **Database**: Databázové soubory.
    - **Plugins**: Databázové pluginy.
    - **Schemas**: Databázová schémata.
  - **Global**: Hlavičkové soubory dostupné v globálním prostoru.
  - **Models**: Modelové třídy obsahující aplikační logiku.
  - **Public**: Vstupní HTML soubor a další statické soubory.
    - **Fonts**: Písma.

        * **Images**: Obrázky rozčleněné do podadresářů.
        * **JavaScript**: Javascriptové soubory.
    - **Utils**: Pomocné třídy.

- **docs**: Teoretická část bakalářské práce včetně zdrojových souborů pro LaTeX.

Moduly obsažené zejména v klientské části projektu jsou adresářové celky, které mohou obsahovat další strukturu:

- **Components**: React komponenty.
- **Constants**: Konstanty a konfigurace modulu.
- **Redux**: Akce a reducer.
- **Styles**: Styly.
- **Views**: Pohledy.

Každý modul obsahuje mapování a export všech souborů, které mají být veřejné. Z vnějšku tedy není nutné znát strukturu daného modulu a adresu hledaného souboru uvnitř modulu.

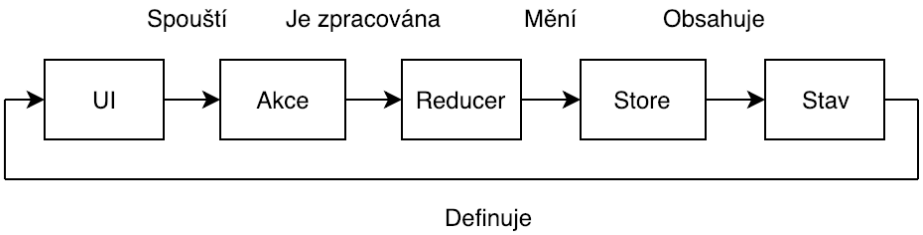
```
import { LoginForm } from '../../User'
import LoginForm from '../../User/Components/LoginForm'
```

Zdrojový kód 4: Správné a nesprávné použití importu souboru z modulu

## 3.2 Uživatelské rozhraní

Klientská část aplikace dodržuje architekturu *Redux*. Všechen stav aplikace je na jednom místě v tzv. *store*, odkud ho mohou číst všechny komponenty, jež jsou na *store* napojené. Kdykoliv se změní stav aplikace, dojde k automatickému překreslení těch částí *UI*, které změněná data obsahují.

Stav aplikace lze změnit zavoláním funkce *dispatch*, jejímž argumentem bude právě prováděná akce. Akce je prostým objektem, který obsahuje povinnou vlastnost `type` s typem akce a libovolné další vlastnosti. Jelikož akce by měly být znovupoužitelné, jsou často umístěny v *action creator*. To je funkce, která akci vrací.




Obrázek 3: Životní cyklus architektury Redux [1]

Samotná změna stavu je pak provedena ve funkci *reducer*. Ten ze starého stavu aplikace a akce vytvoří a vrátí nový stav. Je složen obvykle z přepínače s výčtem všech akcí, kde je u každé akce uvedena její modifikace stavu. Aby změna stavu aktualizovala všechny komponenty, které tomuto stavu naslouchají, je nutné, aby *reducer* vracel vždy nově vytvořený objekt, nikoliv pouze pozměněný ten starý.

```
// Actions.ts
export const increment = () => ({ type: ActionTypes.INCREMENT })

// Reducer.ts
case ActionTypes.INCREMENT:
    return {  ...state, number: state.number + 1 }

// IncrementButton.ts
class IncrementButton extends React.Component<IProps, IState> {

    public render = (): React.ReactNode => (
        <button onClick={this.props.increment}>
            Increment number {this.props.number}
        </button>
    )

}

export default IncrementButton.connect(
    state => ({ number: state.number }), // mapStateToProps
    { increment } // mapDispatchToProps
)
```

Zdrojový kód 5: Ukázka architektury Redux

[1] Vytvořeno v `https://www.draw.io`.

### 3.2.1 Akce

Vytvářet zejména asynchronní akce manuálně je zdlouhavé. Pro správnou funkcionalitu *UI* je totiž při asynchronním požadavku třeba zaznamenat následující stavy:

- **SENT**: Požadavek byl odeslán,
- **SUCCESS**: Požadavek byl úspěšně ukončen,
- **FAIL**: Požadavek byl neúspěšně ukončen.

V projektu byla proto vytvořena vlastní knihovna *Redux* v modulu *Utils*, která zajišťuje vykonávání několika typů akcí:

- **setAction**: Nastavení hodnoty,
- **toggleAction**: Přepnutí hodnoty (generuje *ON/OFF*),
- **asyncAction**: Asynchroní požadavek (generuje *SENT/SUCCESS/FAIL*).

V kódu pak použití asynchronní akce vypadá následovně. Automaticky je zajištěno sledování průběhu požadavku a je tudíž je jednoduše možné zobrazit např. načítací animaci.

```
export const getBody = (bodyId: string) => (
    Redux.asyncAction(
        ActionTypes.GET_BODY,
        { body: Request.get(Paths.GET_BODY(bodyId)) }
    )
)
```

Zdrojový kód 6: Redux akce za využití vlastní knihovny

### 3.2.2 Reducer

Reducer je běžně funkce, která obsahuje *switch* s výčtem všech akcí a jejich modifikací stavu aplikace. Díky vlastní knihovně to ale není nutné. Stačí uvést pole všech akcí a případně i počáteční stav *store*.

```
export default Redux.createReducer(
    Object.values(ActionTypes),
    {
        areLabelsVisible: true,
        areOrbitsVisible: false,
```

```
            bodies: Redux.EMPTY_ASYNC_ENTITY,
            body: Redux.EMPTY_ASYNC_ENTITY
        }
)
```

Zdrojový kód 7: Redux reducer za využití vlastní knihovny

## 3.3 3D grafika

Před vykreslováním těles ve vesmíru je nutné ze serveru nejdříve získat data k těmto tělesům. Následovně je použita třída *BodyFactory*, která z datových objektů těles vytvoří kontejnery obsahující kromě samotných dat z databáze také objekty pro vykreslení.

Celý vesmír je neustále v pohybu. Měsíc obíhá planetu Zemi, která se pohybuje kolem Slunce. Slunce i s celou soustavou obíhá střed naší galaxie. Mléčná dráha se pak volně pohybuje v rámci místní kupy galaxií a ta zas v rámci místní nadkupy galaxií. Počítat absolutní pozici tělesa by proto bylo více než náročné.

Je proto důležité, aby každé těleso bylo umístěno mezi potomky tělesa, kolem kterého zdánlivě obíhá. Tím je dosaženo dědičnosti a souřadnice není nutné uvádět absolutně vzhledem k vesmíru, ale pouze vzhledem k rodičovskému tělesu. Pokud je pozice Slunce [1000, 0, 0], stačí uvést souřadnice Země [5, 0, 0] a knihovna *Three* si už sama dopočítá skutečnou polohu Země [1005, 0, 0].

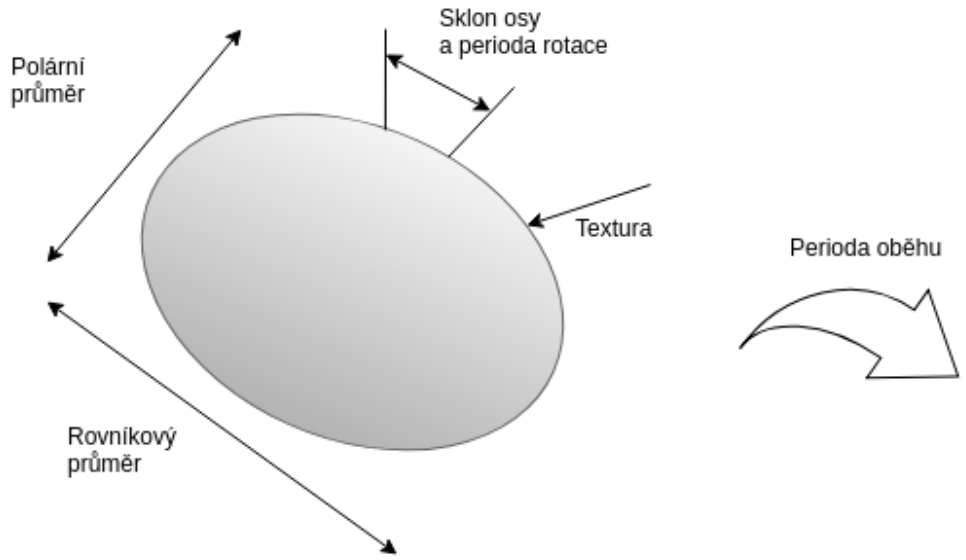
Pro práci s 3D grafikou byla v rámci projektu vytvořena vlastní knihovna *Scene*. Ta v každém průběhu vykreslovací smyčky vypočítá pozice a rotace těles a aktualizuje je.

```
const scene = new Scene({
    controllable: true,
    element: this.container,
    logaritmicDepth: true,
    objects: this.rootBodies.map(this.bodyFactory.create),
    onRender: this.updateBodies,
    target: 'Slunce'
})

scene.setCameraPosition({ x: 10, y: 20, z: 30 })
scene.setCameraTarget('VY Canis Majoris')
```

Zdrojový kód 8: Práce s vlastní knihovnou pro 3D grafiku

### 3.3.1 Těleso

Samotné těleso je reprezentováno koulí, tedy objektem s geometrií *THREE.SphereGeometry*. Pokud má těleso rozdílný rovníkový a polární průměr, je výsledné zploštění řešeno transformaci *new THREE.Matrix4().makeScale()*. Materiálem je *THREE.MeshBasicMaterial* (tělesa emitující světlo) nebo *THREE.MeshPhongMaterial* (tělesa pohlcující světlo). Povrch tělesa tvoří 2D textura *THREE.Texture* ve formátu *JPG* nebo *PNG*.




Obrázek 4: Jednoznačná definice tělesa [1]

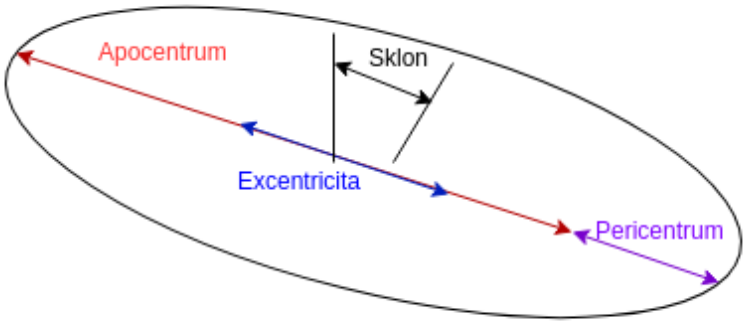
### 3.3.2 Orbita

Orbita je objekt s geometrií *THREE.BufferGeometry*, jejíž tvar je vypočten na základě nejbližší vzdálenosti tělesa od těžiště (pericentrum), nejvzdálenější vzdálenosti tělesa od těžiště (apocentrum) a výstřednosti dráhy (excentricita). Pro materiál je použit *THREE.LineMaterial* s jedinou barvou namísto textury.

```
const a = (apocenter + pericenter) / 2
const b = Math.sqrt(-Math.pow(a, 2) * eccentricity + Math.pow(a, 2))
const path = new THREE.EllipseCurve(0, 0, a, b, ...)
const geometry = new THREE.BufferGeometry()
geometry.setFromPoints(path.getPoints(ORBIT_SEGMENTS))
```

Zdrojový kód 9: Výpočet geometrie orbity tělesa

[1] Vytvořeno v `https://www.draw.io`.



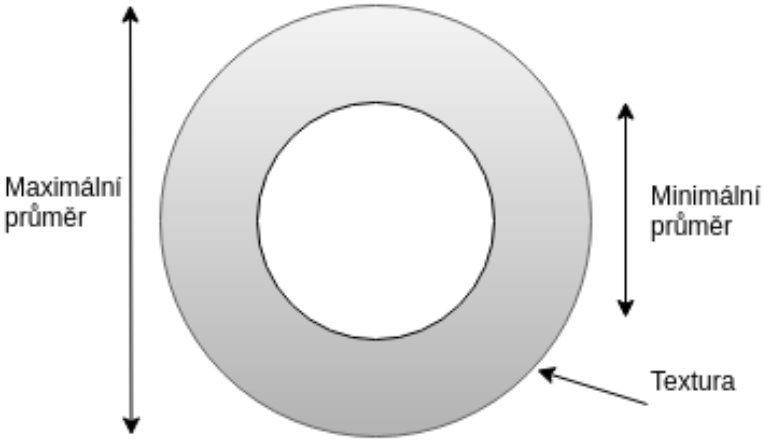

Obrázek 5: Jednoznačná definice orbity tělesa [1]

### 3.3.3 Světlo

Pokud je těleso zdrojem světla, je mezi jeho potomky navíc také bodové světlo *THREE.PointLight*, které emituje světlo o stejné barvě a intenzitě, jakou má reálné těleso.

### 3.3.4 Prstence

Prstenec je objekt s geometrií *THREE.BufferRingGeometry*, materiálem *THREE.MeshLambertMaterial* a 2D texturou *THREE.Texture* ve formátu *JPG* nebo *PNG*. Těleso může mít libovolný počet prstenců, nebo také žádný.




Obrázek 6: Jednoznačná definice prstence tělesa [1]

[1] Vytvořeno v `https://www.draw.io`.

### 3.3.5 Popisek

Popisek je reprezentován *HTML* elementem, který je absolutně pozicován na stejné souřadnice, jako je vykreslované těleso. Kromě názvu tělesa obsahuje i vzdálenost od Země a vzdálenost od kamery.

## 3.4 Serverová část

### 3.4.1 Zachycení HTTP požadavku

Server vystavuje *REST API*, ze kterého si mohou klientské aplikace (v tomto případě pouze webová aplikace) stahovat data. Rozhraní má jasně danou strukturu a jeho implementace a dokumentace je provedena pomocí nástroje *Swagger*. Ten umožňuje sestavit rozhraní z fyzických souborů. Cesta k souboru značí adresu *URI* a obsah zas dostupné metody.

```
// /bodies/{bodyId}.ts
// Routes like [GET] /bodies/10 or [POST] /bodies/20
export default {
    get: (req, res) => res.send('Hello world!')
    put: (req, res) => res.status(401).send('Admin required.')
    post: (req, res) => BodyModel.add(req.body).then(res.send)
    delete: (req, res) => res.send('Body was deleted.')
}
```

Zdrojový kód 10: Definice cesty v REST API

Dokumentace je zpřístupněna na lokální adrese */api-docs*. Je možné se zde informovat o všech možných *HTTP* požadavcích nebo je s nastavenými parametry simulovat.

### 3.4.2 Zpracování HTTP požadavku

Jakmile je přijat požadavek od klienta, musí dojít k následujícím krokům:

- Zjištění identity uživatele z tokenu v hlavičce,
- Zkontrolování vyžadovaných oprávnění,
- Obsluha požadavku (načtení a zpracování dat z DB, ...),
- Nastavení HTTP statusu, popř. chybového kódu,

- Odeslání odpovědi.

V aplikaci byla vyvinuta vlastní knihovna, která všechny ze zmíněných procesů automatizuje. Pro většinu druhů zdrojů stačí dva druhy přístupových bodů:

- Pro všechny zdroje,
  - **GET**: Vrátí pole zdrojů,
  - **POST**: Vytvoří nový zdroj a vrátí zprávu o úspěchu nebo chybě,
  - **DELETE**: Smaže všechny zdroje a vrátí počet smazaných zdrojů,
- Pro jeden zdroj,
  - **GET**: Vrátí jeden zdroj podle ID nebo zprávu o chybě,
  - **PUT**: Upraví jeden zdroj podle ID a vrátí zprávu o úspěchu nebo chybě,
  - **DELETE**: Smaže jeden zdroj podle ID a vrátí zprávu o úspěchu nebo chybě.

Samotná definice přístupových bodů pak může vypadat následovně. U každé přístupové metody je možné určit, jaká oprávnění musí mít uživatel, aby tuto metodu nad daným zdrojem mohl vykonat.

```
// bodies.ts
export default Route.getRouteGroupForAll(
    BodyModel
    { get: Route.all, post: Route.all, delete: Route.onlyAuthenticated }
)

// bodies/{bodyId}.ts
export default Route.getRouteGroupForOne(
    BodyModel,
    { get: Route.all, put: Route.onlyAdmin, delete: Route.onlyAdmin }
)
```

Zdrojový kód 11: Zpracování HTTP požadavku

Modelové třídy většiny entit není nutné psát individuálně. Všechny mají za úkol poskytnout *CRUD operace* nad danou entitou (např. *BodyModel* nad tělesy). Jediné, co se liší, je název databázové kolekce, vybrané sloupce při výpisu detailu entity a při výpisu všech entit, popř. další parametry. Veškerá logika komunikace s databází byla ukryta do třídy *ItemModel*. Většina dalších modelů je pak pouhou instancí této třídy s konkrétními

parametry. *ItemModel* vedle *CRUD* operací automatizuje i vytváření notifikací a schvalovací proces administrátora.

```
// BodyModel.ts
export default new ItemModel<IBody, ISimpleBody, INewBody>(
    dbModel: DatabaseModels.BODY,
    get: {
        selectAll: ['name', 'orbit'],
        joinOne: ['typeId'],
        joinAll: ['typeId']
    },
    add: { approval: true, notification: true }
    update: { approval: true, onAfter: console.log }
    remove: { notification: true, onBefore: onBeforeCallback }
)
```

Zdrojový kód 12: Automatizovaná tvorba modelových tříd

## 3.5 Databáze

Trvalá data se v ukládají do databáze *MongoDB* za využití knihovny *Mongoose*. Serverová část se řídí architekturou MVC a k databázi mají tedy přístup pouze modelové třídy.

### 3.5.1 Připojení do databáze

Připojení do databáze je jednoznačně inicializováno textovým řetězcem obsahujícím jméno a heslo uživatele, cluster a název databáze. Dále je třeba zaregistrovat všechna schémata, která budou v aplikaci využita. Pokud kolekce s nově zaregistrovaným schématem neexistuje, dojde k jejímu automatickmu vytvoření. Je také možné reagovat na úspěšné připojení nebo na chybu.

Veškerou tuto funkcionalitu obstarává třída *Database*, která byla vytvořena v rámci projektu. Instance této třídy je staticky (tedy pouze jednou, při spuštění serveru) vytvořena ve třídě *Model*, který je předkem všech ostatních modelových tříd. Třída *Model* pak tuto instanci databáze poskytuje jako instanční proměnnou s modifikátorem přístupu `protected` všem svým potomkům.

```
Model.db = new Database({
    userName: Config.database.username,
```

```
    password: Config.database.password,
    cluster: Config.database.cluster,
    database: Config.database.name,
    onError: console.error,
    schemas: {
        [DatabaseModels.BODY]: BodySchema,
        [DatabaseModels.BODY_EVENT]: BodyEventSchema,
        [DatabaseModels.BODY_TYPE]: BodyTypeSchema,
        [DatabaseModels.TOKEN]: TokenSchema,
        [DatabaseModels.USER]: UserSchema,
        ...
    }
})
```

Zdrojový kód 13: Připojení k databázi v aplikaci

### 3.5.2 Mongoose schéma

Pomocí schématu lze určit strukturu databázové kolekce a nastavit validační pravidla. Veškerá konfigurace kolekce je umístěna v objektu, který se dosadí jako parametr při vytváření nové instance schématu.

```
const UserSchema = new Mongoose.Schema({
    email: {
        type: String,
        required: [true, 'Email is required.'],
        unique: true,
        validate: { validator: Strings.isEmail }
    },
    ...
})
```

Zdrojový kód 14: Vytvoření databázového schématu

### 3.5.3 Mongoose plugin

Pokud je třeba reagovat na jednotlivé události v kolekci (smazání, vložení, ...), lze toho docílit pomocí pluginů. To jsou obdoby spouští z relačních databází. Jedná se o uživatelem definované funkce, které dostanou v parametru schéma, na kterém jsou zaregistrovány. Na

tomto schématu pak mohou definovat operace `pre` nebo `post` s jednotlivými událostmi a jejich obsloužení.

```
// UserSchema.ts
UserSchema.plugin(HashPlugin, { field: 'password' })

// HashPlugin.ts
const HashPlugin = Plugins.onChange(async (doc, options) => {
    doc[options.field] = await Security.hash(doc[options.field])
}, options.field)
```

Zdrojový kód 15: Vytvoření a použití databázového pluginu

Ve zdrojovém kódu 12 je popsán plugin pro automatické hashování hesla. Ten při každé změně daného pole nový obsah zahashuje a teprve onen hash uloží do databáze namísto původní hodnoty. Protože ale není vyloučené, že bude někdy třeba tento plugin použít také v jiném schématu a na jiném poli, plugin byl vytvořen univerzálně. Jméno pole je proto určeno parametrem v pluginu.

### 3.5.4 Práce s databází

Databáze umožňuje provádět velké množství operací nad daty v ní uloženými. Je možné vyhledávat podle konkrétní hodnoty, pole hodnot, regulárních výrazů nebo číselného rozsahu. Takto vyfiltrovaná data lze číst, smazat či editovat. Je také možné nastavit spoustu parametrů, např. maximální počet dokumentů (*limit*), které vyhovují filtru. Mimo jiné je možné také provádět více dotazů do databáze jako atomické operace pomocí transakcí.

```
const User = this.db.getModel(DatabaseModels.USER)

// User's names, which starts with 'a' and doesn't contain any number.
User.find({ name: /^a[^0-9]*$/ }}).select('name').skip(5).limit(10)

// Create new user.
new User({ email: 'email@domail.cz', password: 'heslo' }).save()

// Update user by ID.
User.findByIdAndUpdate(
    '5ba0bef7df0fca0bf99305c1',
    { email: 'newEmail@domain.cz', name: 'newName' }
```

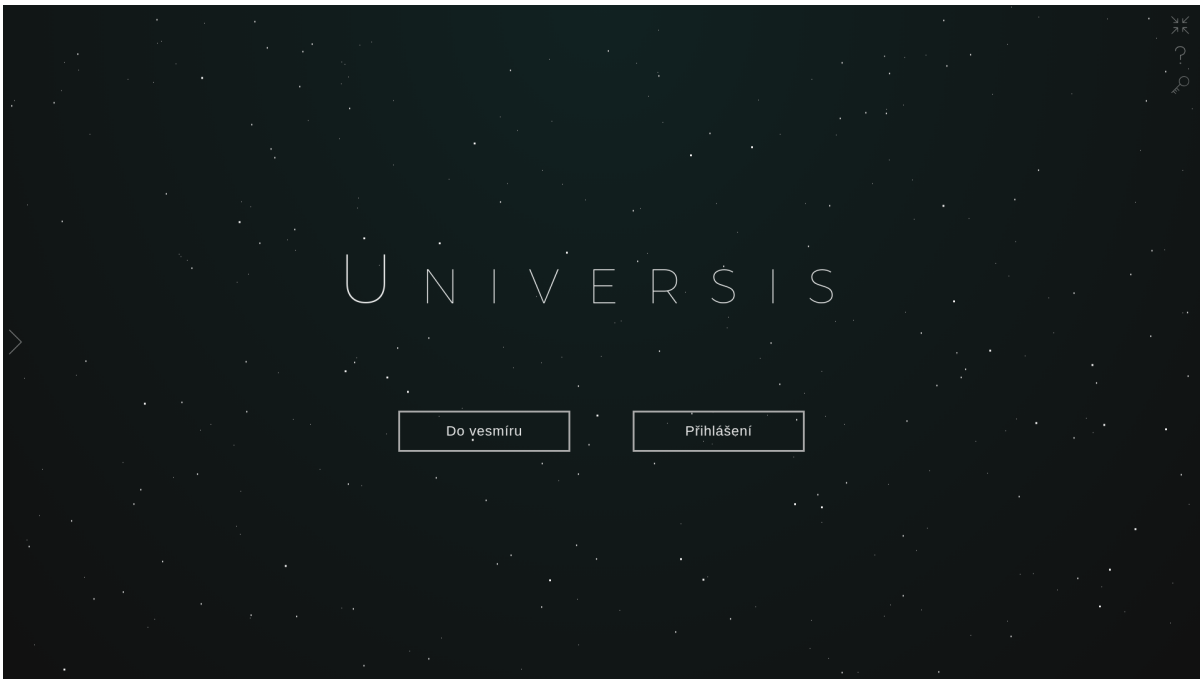
```
)

// Delete all users with names in array, but only if they are adult.
const names = ['michal', 'username', 'admin']
User.deleteMany({ name: { $in: names }, age: { $gte: 18 } })
```

Zdrojový kód 16: CRUD operace nad kolekcí uživatelů

# 4 ROZVRŽENÍ APLIKACE

## 4.1 Hlavní stránka

Hlavní stránka aplikace obsahuje pouze nezbytné odkazy do dalších částí aplikace. Nepřihlášený vidí odkazy do simulátoru a přihlašovací stránku. Přihlášený uživatel vidí taktéž odkaz do simulátoru, ale druhé tlačítko slouží pro odhlášení.




Obrázek 7: Hlavní stránka

## 4.2 Nápověda

V nápovědě je shromážděno několik článků, které mohou pomoci novým uživatelům lépe se orientovat v aplikaci. Mezi jednotlivými sekcemi nápovědy lze přepínat pomocí menu v levé části obrazovky.

## 4.3 Autentifikace uživatele

### 4.3.1 Identita

Každý uživatel, jenž se chce přihlásit nebo se zaregistrovat, se dostane na stránku, kde musí prokázat svou identitu zadáním emailu. V závislosti na tom, zda zadaný email již v databázi existuje bude odkázán na stránku pro přihlášení nebo pro registraci.


Obrázek 8: Formulář pro zjištění identity uživatele

### 4.3.2 Přihlášení

Stránka obsahuje formulář pro zadání hesla. Nachází se zde odkaz zpět a odkaz pro obnovení hesla.


Obrázek 9: Formulář pro přihlášení uživatele

### 4.3.3 Registrace

Stránka pro zadání a potvrzení hesla k nově vytvářenému účtu. V případě, že uživatel chce změnit registrační email, je možné se vrátit na stránku se zadáváním identity.


Obrázek 10: Formulář pro registraci uživatele

### 4.3.4 Zapomenuté heslo

Na této stránce se nachází formulář s polem pro email. Po úspěšném vyplnění se odešle na zadaný email zpráva s odkazem pro obnovení hesla.

### 4.3.5 Reset hesla

Stránka obsahuje formulář pro nastavení a potvrzení nového hesla. Pro úspěch je nutné, aby se v URL adrese nacházel platný token. Ten se vytvoří v okamžiku vytvoření požadavku na obnovu hesla a má platnost 30 minut.

## 4.4 Uživatel

### 4.4.1 Detail uživatele

### 4.4.2 Editace uživatele

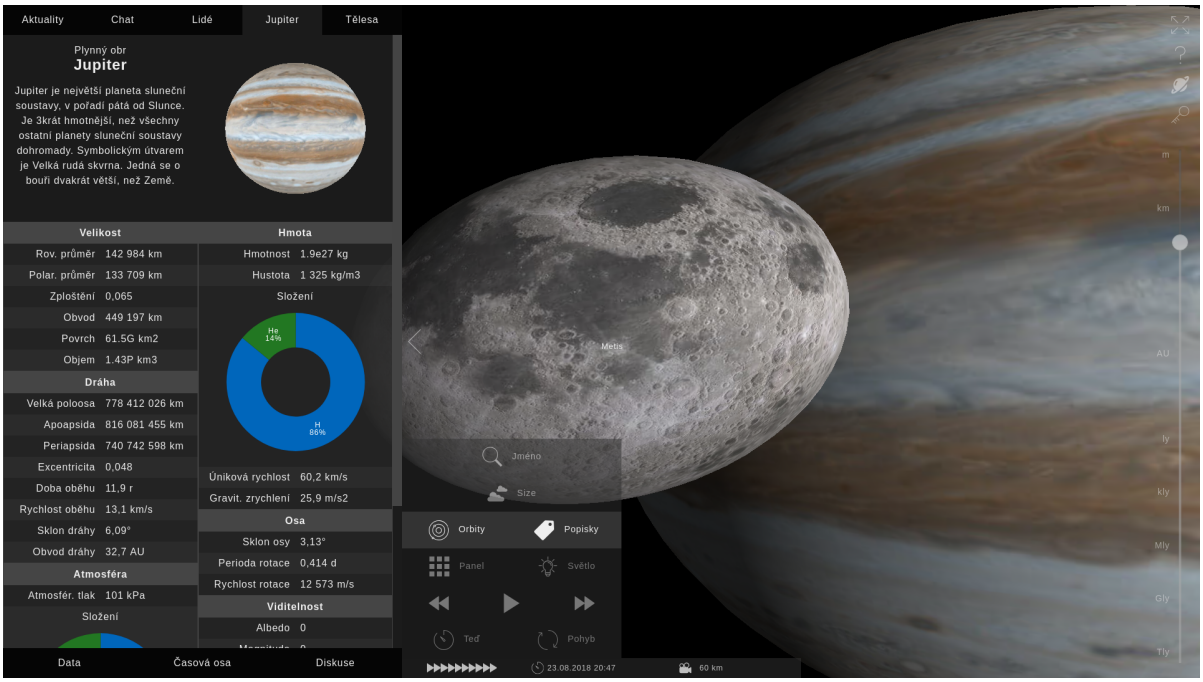
## 4.5 Simulátor

Hlavní náplní aplikace je právě tato stránka. Na 3D scéně zobrazuje tělesa ve vesmíru v reálném čase s realistickými poměry velikostí i vzdáleností. Uživatel může myší nebo touchpadem libovolně otáčet kamerou kolem vycentrovaného tělesa.

V pravém dolním rohu se nachází ovládací panel, jenž umožňuje měnit některá nastavení simulátoru. Všechna z nich jsou dostupná i pod klávesovými zkratkami.

- **Jména** (N): Zobrazí nebo skryje názvy těles.
- **Vzdálenosti od kamery** (C): Zobrazí nebo skryje vzdálenosti těles od kamery.
- **Vzdálenosti od Země** (E): Zobrazí nebo skryje vzdálenosti těles od Země.
- **Zrychlit** (W): Pokud je čas kladný, zrychlí ho 10krát. Jinak ho 10krát zpomalí.
- **Zpomalit** (S): Pokud je čas kladný, zpomalí ho 10krát. Jinak ho 10krát zrychlí.
- **Vrátit čas** (T): Nastaví čas simulátoru na aktuální čas.
- **Vrátit rychlost** (V): Nastaví čas simulátoru na 1.
- **Panel** (P): Zobrazí nebo skryje detail právě vycentrovaného tělesa.
- **Pohyb** (M): Spustí nebo ukončí rotaci kamery kolem vycentrovaného tělesa.
- **Světlo** (L): Zapne nebo vypne světlo.
- **Orbity** (O): Zobrazí nebo skryje orbity těles.

Na dolním okraji obrazovky je lišta zobrazující aktuální nastavení a čas simulátoru. Napravo je možné vidět posuvník s aktuální velikostí pohledu.



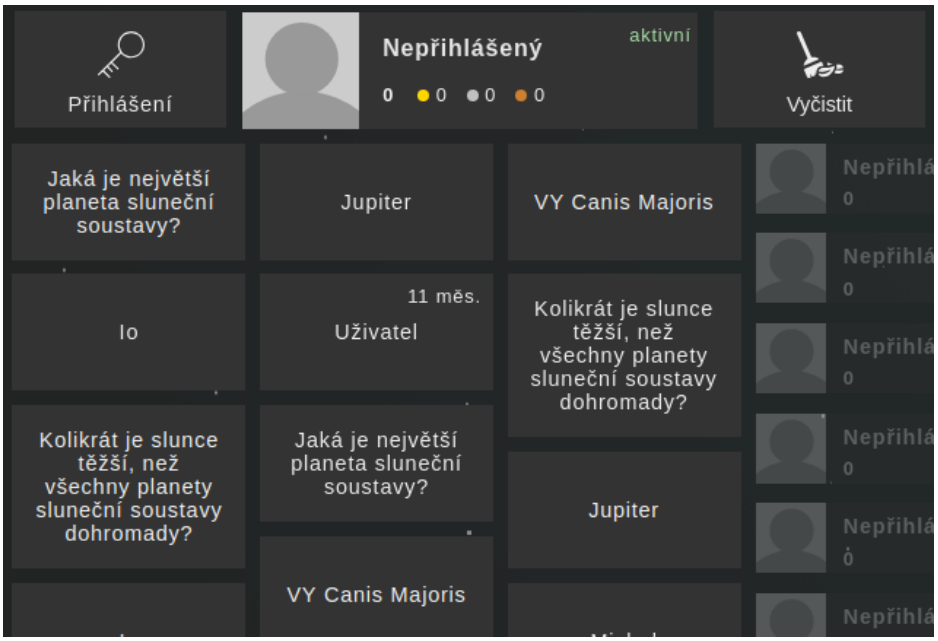

Obrázek 11: Simulátor

## 4.6 Uživatelský panel

Uživatelský panel je sekundární okno, ve kterém si uživatel může zobrazovat části aplikace bez nutnosti opouštět aktuální stránku. Zobrazuje se jako poloprůhledný obdélník v levé části obrazovky a obsahuje 4 nezávislé záložky.

### 4.6.1 Přehled

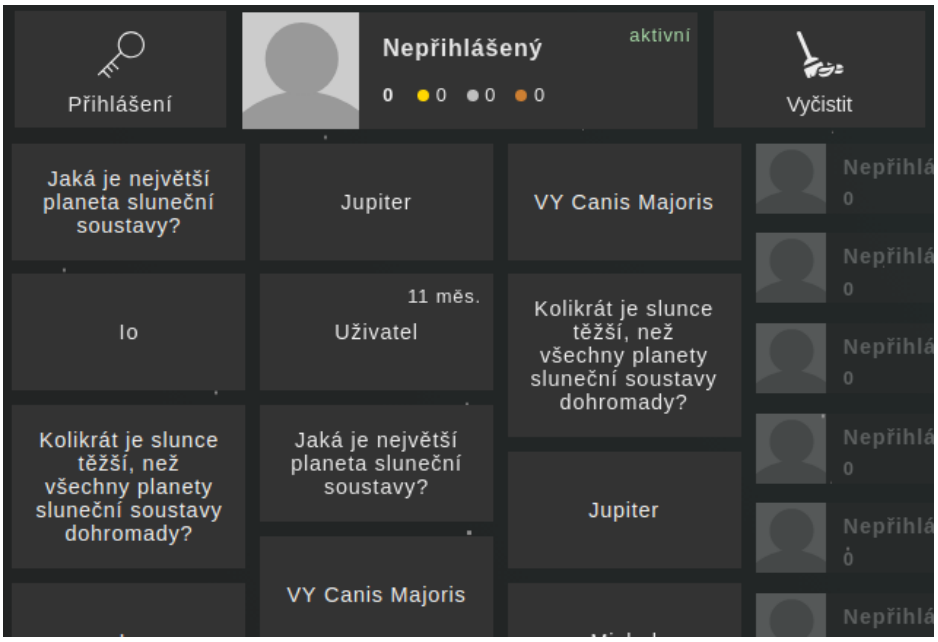
Výchozí položka v uživatelském panelu zobrazuje poslední události na webu a uživatele. Uživatele lze řadit podle času poslední aktivity nebo reputace. Přihlášený uživatel zde také vidí stručné informace o svém účtu.




Obrázek 12: Přehled

### 4.6.2 Chat

Chat slouží pro vzájemnou komunikaci uživatelů. Zprávy jsou zobrazovány okamžitě všem aktivním uživatelům díky technologii *socket.io*.




Obrázek 13: Uživatelský chat

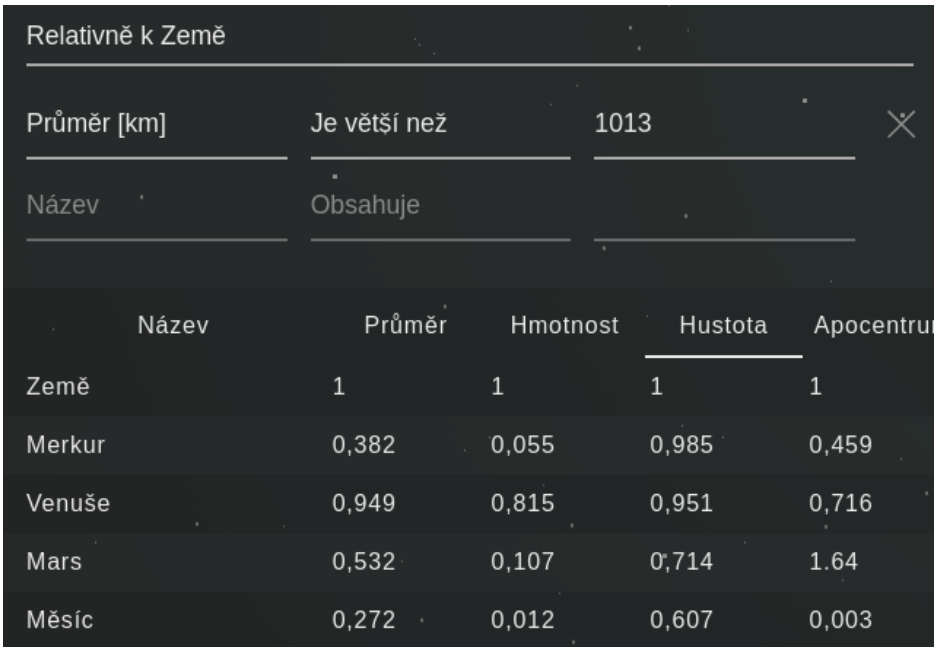
### 4.6.3 Seznam těles

Seznam těles umožňuje zobrazit tělesa v databázi i s jejich údaji. Údaje mohou být:

- **Absolutní**: Zobrazí absolutní hodnotu (např. průměr Slunce je 1 392 684 km).
- **Relativní k libovolnému tělesu**: Zobrazí poměr aktuální hodnoty ku hodnotě u porovnávaného tělesa (např. průměr Slunce je roven 109 průměrům Země).

Tělesa lze řadit vzestupně i sestupně podle libovolného kritéria a taktéž je lze podle libovolných kritérií filtrovat za použití následujících vztahů:

- **Obsahuje**: Vyhoví, pokud hodnota obsahuje hledaný text.
- **Je roven**: Vyhoví, pokud je hodnota rovna hledanému textu.
- **Začíná na**: Vyhoví, pokud hodnota začíná hledaným textem.
- **Končí na**: Vyhoví, pokud hodnota končí hledaným textem.
- **Je větší než**: Vyhoví, pokud je hodnota větší, než hledaný text. V případě nečíselné hodnoty vyhoví ty, které se v abecedě nachází později.
- **Je menší než**: Vyhoví, pokud je hodnota menší, než hledaný text. V případě nečíselné hodnoty vyhoví ty, které se v abecedě nachází dříve.



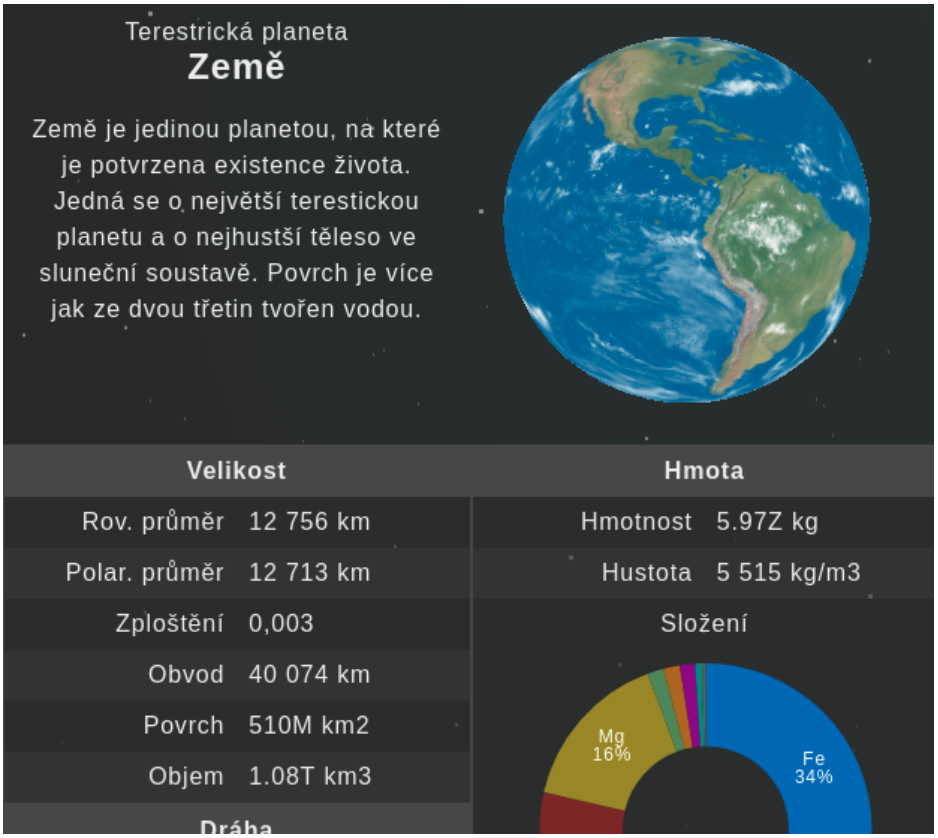

Obrázek 14: Seznam těles

### 4.6.4 Detail tělesa

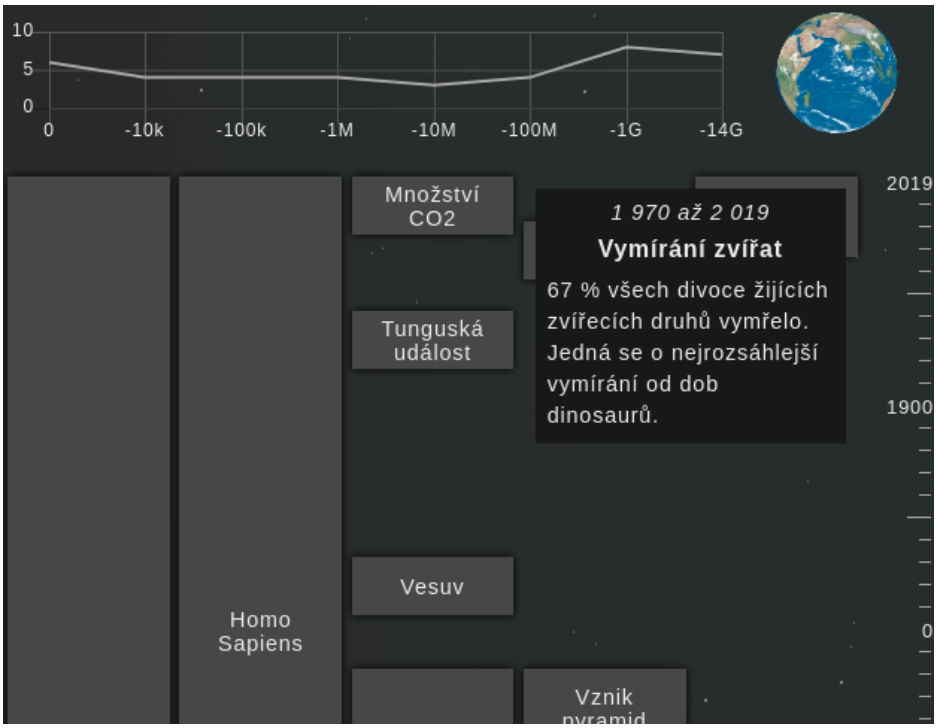
Detail tělesa se skládá ze čtyř záložek. Výchozí z nich zobrazuje výčet všech známých údajů v tabulce o daném tělese. Součástí je i krátká charakteristika tělesa a jeho 3D vizualizace.

Na druhé záložce se nachází časová osa, na které jsou vyneseny historické události spojené s tělesem. Každá událost obsahuje rok, ve kterém nastala, název a po najetí kurzorem i krátký popis. Je zde i stručný graf zobrazující počet výskytů události v jednotlivých časových obdobích.

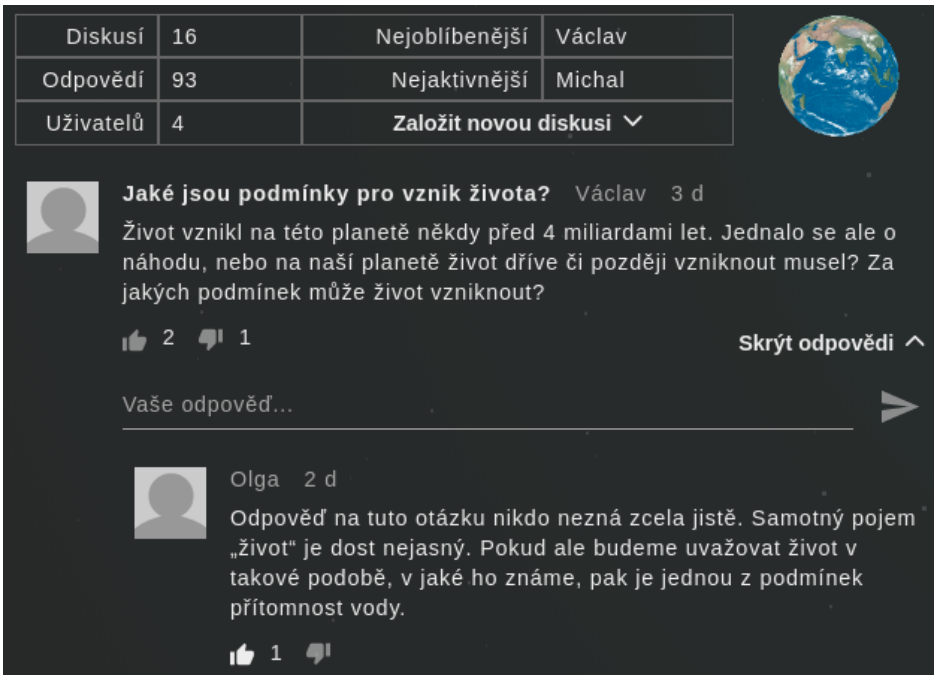
Třetí záložka je určena pro diskuse. Každý zde může reagovat na již existující diskuse, nebo může založit vlastní vlákno. Všechny příspěvky uživatelů lze kladně nebo záporně hodnotit, v důsledku čehož se bude přičítat nebo odečítat reputace autora.




Obrázek 15: Detail tělesa




Obrázek 16: Časová osa tělesa




Obrázek 17: Diskuse o tělese

# 5 PROBLÉMY ŘEŠENÉ PŘI IMPLEMENTACI

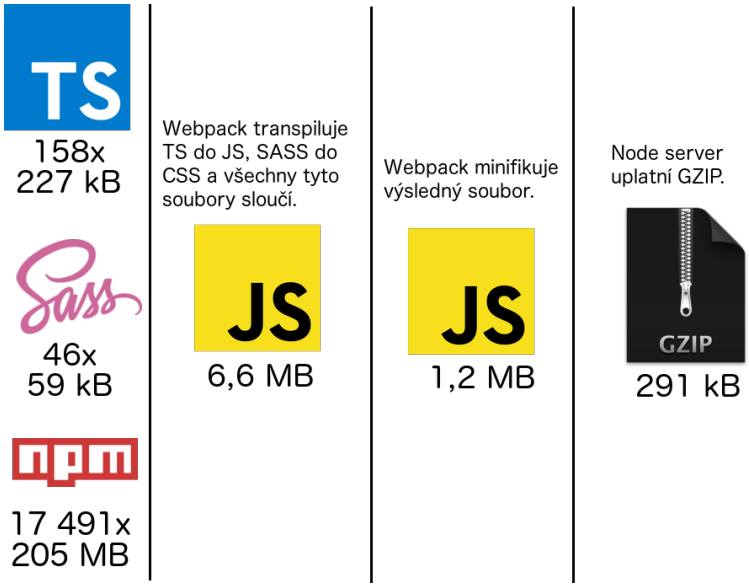
## 5.1 Omezení viditelnosti těles

V databázi je uloženo velké množství těles. Pokud by se popisky a dráhy všech těles zobrazovaly najednou, bylo by těžké se v simulátoru orientovat. Navíc by tento stav měl negativní důsledky na výkon aplikace. Proto se ve vykreslovací smyčce počítá relativní poloha všech těles vzhledem ke kameře. V závislosti na této poloze se určí, jak se bude dané těleso zobrazovat. Může nastat několik případů, které je nutno ošetřit:

- **Dráha je 40krát větší/menší než obrazovka**: Dráha tělesa bude poloprůhledná, popisky se nebudou zobrazovat.
- **Dráha je 1000krát větší/menší než obrazovka**: Dráha ani popisky se nebudou zobrazovat.
- **Vzdálenost od tělesa je 40krát větší než vzdálenost od centrálního tělesa**: Dráha tělesa bude poloprůhledná, popisky se nebudou zobrazovat.
- **Vzdálenost od tělesa je 1000krát větší než vzdálenost od centrálního tělesa**: Dráha ani popisky se nebudou zobrazovat.

První dva body řeší případy, kdy je těleso příliš oddálené nebo příliš přiblížené. Poslední dva body řeší situaci, ve které má uživatel sice přiměřené přiblížení kamery, nicméně se od tělesa nachází příliš daleko. Příkladem může být situace, kdy si uživatel prohlíží ze vzdálenosti 400 tisíc km prstence planety Saturn. Dráha zemského Měsíce je přiměřeně velká (406 tisíc km), ale přesto by se neměla vykreslovat. Od Saturnu je totiž vzdálena 1,2 miliardy km.

## 5.2 Sestavení aplikace a optimalizace




Obrázek 18: Sestavení aplikace a optimalizace [1, 2, 3]

### 5.2.1 Transpilace a sloučení JS a CSS souborů

V projektu se nachází velké množství souborů s typescriptovým kódem a souborů se styly. Zahajovat HTTP požadavek pokaždé, kdy si uživatel chce stáhnout jakýkoliv z těchto souborů vyžaduje hodně režije a je to z hlediska času i přenesených dat náročné. Je proto výhodnější všechny tyto soubory sloučit do jediného a ten stáhnout jako jeden celek.

Ze všeho nejdříve je ale nutné transpilovat zdrojové kódy do nativních jazyků. Prohlížeče totiž TypeScriptu ani SASSu nerozumí. TypeScript se transpiluje do JavaScriptu za využití transpilátoru, jenž je dodáván společně s TypeScriptem. Pro převod SASS na CSS slouží Webpack. Ten následně také všechny takto transpilovaného soubory slouží do jediného javascriptového souboru.

[1] Ikona SASS převzata z `https://sass-lang.com/assets/img/logos/logo-b6e1ef6e.svg`
[2] Ikona NPM převzata z `https://pepa.holla.cz/wp-content/uploads/2016/06/npm.png`
[3] Ikona GZIP převzata z `https://codeopinion.com/wp-content/uploads/2016/02/gzip.png`

### 5.2.2 Minifikace a GZIP komprese

I přesto, že jsou všechny zdrojové soubory sloučené do jednoho, stále se jedná o velký objem dat. Je proto vhodné celý soubor pomocí nástroje Webpack minifikovat – odebrat z něj přebytečné mezery, komentáře a zkrátit názvy lokálních proměnných.

Pro ještě větší redukci přenesených dat je dobré nastavit Node.js server tak, aby na všechny odeslané soubory uplatnil GZIP kompresi.

## 5.3 Bezpečnost a ochrana dat

### 5.3.1 Autentizace pomocí tokenu

Uživatel se autentizuje při přihlášení svým emailem a heslem, čímž mu je umožněn přístup na jeho účet. Nicméně uživatele je třeba znovu autentizovat i během jeho relace po přihlášení, kdykoliv komunikuje se serverem. Tím lze omezit provádění určitých operací (např. editování tělesa, schvalování úprav, ...) pouze na konkrétní uživatele nebo skupiny uživatelů (např. pouze přihlášení, administrátoři, ...). Nabízí se několik možností v tom, co na server posílat vždy, když je třeba autentifikace:

- **Email a heslo**: Nechat uživatele zadávat přihlašovací údaje pokaždé, když je třeba komunikovat se serverem, by bylo obtěžující. A ukládat heslo na straně klienta není zas bezpečné, protože by zde muselo být v čitelné podobě.
- **ID uživatele**: Jedná se o nebezpečný postup. ID uživatele je veřejný údaj, ke kterému mají přístup i všichni ostatní. Nic by tedy nebránilo útočníkovi poslat *HTTP* požadavek s ID libovolného uživatele.
- **Token**: Ideálním způsobem se zdá být vygenerování náhodného textového řetězce při přihlášení. Tímto řetězcem se po dobu přihlášení bude uživatel prokazovat. Protože tento řetězec nezná nikdo jiný, než uživatel sám, nikdo jiný se s ním tedy nemůže autentizovat. Navíc proti zneužití má omezenou platnost, po jejímž vypršení je třeba požádat o nový token.

V aplikaci je využit poslední ze zmíněných způsobů. Token má platnost 30 minut. Je tedy zajištěna bezpečná autoritace. Každý může provádět pouze ty operace, na které má skutečně právo.

### 5.3.2 Hashování hesel

Ukládat hesla v čitelné podobě do databáze není bezpečné. Potenciální útočník, který by prolomil zabezpečení serveru a dostal se k databázi by viděl hesla všech uživatelů. Tento problém řeší *hashování hesel*.

Do databáze se neuloží heslo samotné, ale pouze jeho otisk (*hash*). Když je potřeba porovnat zadané heslo (např. při přihlášování) s heslem, porovnají se jejich otisky. Tím je umožněna autentifikace a zároveň je znemožněno útočníkovi přečíst hesla z databáze.

Pro vyšší bezpečnost se otisk nevypočítává z pouhého hesla, ale hesla spojeného s dalším nijak nesouvisejícím řetězcem. Tím se zabrání možnosti, aby více uživatelů mělo stejné heshe, pokud mají stejná hesla. Tomuto dodatečnému řetězci se říká sůl (*salt*). Často se také otisk vypočítává několikrát za sebou, takže v databázi není uložen otisk hesla, ale otisk, který vznikl na základě jiného otisku.

*Hash* je výsledkem jednosměrné funkce. To znamená, že z již vypočteného hashe nelze získat původní text. Výjimku tvoří zastaralé *hashovací* funkce, které již byly prolomeny. Je vytvořena databáze pro všechny možné otisky a texty, ze kterých byly vypočítány.

V aplikaci je použita *hashovací funkce bcrypt* s 10 iteracemi a to přímo na úrovni databáze. Celá logika je skryta do tzv. *Mongoose pluginu*. Ten může reagovat na různé události a modifikovat dokument.

```
// UserSchema.ts
UserSchema.plugin(HashPlugin, { field: 'password' })

// HashPlugin.ts
const HashPlugin = Plugins.onChange(async (doc, options) => {
    doc[options.field] = await Security.hash(doc[options.field])
}, options.field)
```

Zdrojový kód 17: Hashování hesel v Mongoose schématu

## 5.4 Zobrazování hodnot fyzikálních veličin

Aplikace zobrazuje různě velké hodnoty různých fyzikálních veličin. Z důvodu přehlednosti a omezeného množství prostoru, ve kterém se tyto hodnoty zobrazují, byla vytvořena

pomocná třída `Units` umístěná do modulu `Utils`. Ta umožňuje převádět a formátovat jednotky do několika podob.

```
Units.convert(Units.ANGLE.RADIAN, Units.ANGLE.DEGREE, Math.PI) // 180
Units.convert(Units.MASS.KG, Units.MASS.G, 2.5) // 2500
Units.toFull(1234567890, Units.SIZE.KM) // 1 234 567 890 km
Units.toShort(1234567890, Units.SIZE.KM) // 1.23G km
Units.toExponential(1234567890, Units.SIZE.KM) // 1.23e9 km
Units.toFull(123456780, Units.SIZE.KM, Units.SIZE) // 8.25 AU
```

Zdrojový kód 18: Ukázka formátování jednotek

Všechny číselné hodnoty se automaticky zaokrouhlují s rozumnou přesností:

```
Units.toShort(1234.567890) // 1.23k
Units.toShort(123.4567890) // 123
Units.toShort(12.3456790) // 12.3
Units.toShort(1.234567890) // 1.23
Units.toShort(0.0001234567890) // 0.0001
```

Zdrojový kód 19: Ukázka zaokrouhlování hodnot

Není nutné používat předdefinované jednotky v této třídě a lze si vytvořit vlastní. Vytvoření jednotek pro velikost vypadá takto:

```
public static SIZE = {
    M: { value: 1, shortName: 'm' },
    KM: { value: 1000, shortName: 'km' },
    AU: { value: 149597870700, shortName: 'AU' },
    LY: { value: 9461e12, shortName: 'ly' },
    KLY: { value: 9461e15, shortName: 'kly' },
    MLY: { value: 9461e18, shortName: 'Mly' },
    GLY: { value: 9461e21, shortName: 'Gly' },
    TLY: { value: 9461e24, shortName: 'Tly' }
}
```

Zdrojový kód 20: Tvorba vlastních jednotek

## 5.5 Vykreslování časové osy

Pro vykreslení časové osy představené v bodě *4.5.4 Detail tělesa* byla v rámci projektu vytvořena komponenta *EventsArea*. Ta umožňuje vykreslit pole událostí na *2D* ploše v podobě čtyřúhelníků tak, aby si souřadnice jednotlivých událostí odpovídaly. V případě časové osy jsou těmito souřadnicemi počáteční a koncový rok události.

Situace je o to komplikovanější, že časová osa nemusí být lineární a dokonce ani logaritmická. Komponenta proto na vstupu vedle pole událostí přijme i pole hraničních souřadnic. Díky tomu je možné zobrazit roky 2019 až 0 jako jeden díl na časové ose a o něco níže naprosto stejný díl pro roky -1 až -10 000, stejně jako např. -5 mld. až -10 mld.

Překrývání jednotlivých události je ošetřeno tak, že se vykreslí do různých sloupců. Je také možno nastavit počet mezistupňů mezi hraničními souřadnicemi. Ty lze nastylovat např. jako čáry na pravítku pro zlepšení přehlednosti. Při pohybu s kurzorem myši nad oblastí se zobrazuje horizontální čára s aktuální souřadnicí (rokem), nad kterou se kurzor nachází. Pro implementaci komponenty byla použita CSS vlastnost *grid*.

```
<EventsArea
  columnsCount={5}
  events={[event1, event2, ...]}
  formatTickValue={Units.toShort}
  minorTicksCount={9}
  tickHeight={15}
  ticks={[new Date().getFullYear(), 0, -1e4, -1e6, -1e8, ...]} />
```

Zdrojový kód 21: Příklad použití komponenty EventsArea

# ZÁVĚR

# POUŽITÁ LITERATURA

[1] KLECZEK, Josip. Velká encyklopedie vesmíru. Praha: Academia, 2002s [cit. 25. 10. 2018]. ISBN 80-200-0906-x.

[2] REES, Martin, Vesmír. Přeložil Pavel PŘÍHODA. Praha: Knižní klub, 2006 [cit. 25. 10. 2018]. ISBN 80-242-1668-x.

[3] JPL Solar System Dynamics. *JPL Solar System Dynamics* [online]. [cit. 25. 10. 2018] Dostupné z: `https://ssd.jpl.nasa.gov`.

[4] MARDAN, Azat. Practical Node.js: building real-world scalable web apps. Berkeley, California: Apress, [2014]. Expert's voice in Web development. ISBN 978-1-4302-6595-5.

[5] BANKS, Alex and PORCELLO, Eve. Learning React: functional web development with React and Redux. Sebastopol, CA: O'Reilly Media, 2017 [cit. 25. 10. 2018]. ISBN: 978-1-4919-542-1.

# SEZNAM PŘÍLOH

# SEZNAM SOUBORŮ ZDROJOVÝCH KÓDŮ NA PŘILOŽENÉM NOSIČI