

Gymnázium, Praha 6, Arabská 14

Programování



MATURITNÍ PRÁCE

Sudoku se vším všudy

Vypracoval:

Vedoucí práce:

Vladimír Vávra, 4.E

Ing. Daniel Kahoun

Březen 2022

Prohlašuji, že jsem jediným autorem tohoto projektu, všechny citace jsou řádně označené a všechna použitá literatura a další zdroje jsou v práci uvedené. Tímto dle zákona 121/2000 Sb. (tzv. Autorský zákon) ve znění pozdějších předpisů uděluji bezúplatně škole Gymnázium, Praha 6, Arabská 14 oprávnění k výkonu práva na rozmnožování díla (§ 13) a práva na sdělování díla veřejnosti (§ 18) na dobu časově neomezenou a bez omezení územního rozsahu.

V dne

Podpis autora

Rád bych zde poděkoval Kryštofu Havránkovi za zapůjčení LaTeX vzoru pro dokumentaci a prezentaci.

Název práce: Sudoku se vším všudy

Autoři: Vladimír Vávra, 4.E

Abstrakt: Vytvořte webovou aplikaci, jejímž primárním účelem je hraní sudoku v klasické podobě i možných variantách lišících se typem hracího pole i počtem hráčů. Aplikace dokáže sudoku vyřešit a vygenerovat zadání, u základní varianty na základě obtížnosti.

Klíčová slova: sudoku, javascript, nodejs, react js

Title: Sudoku with everything in it

Authors: Vladimír Vávra, 4.E

Abstract: Create a web application whose primary purpose is to play sudoku in its classic form and possible variations differing in the type of playing field and the number of players. The application can solve the sudoku and generate the assignment, for the basic variant based on difficulty.

Key words: sudoku, javascript, nodejs, react js

Title: Sudoku mit allem, was dazugehört

Autoren: Vladimír Vávra, 4.E

Abstrakt: Erstellen Sie eine Webanwendung, deren Hauptzweck darin besteht, Sudoku in seiner klassischen Form und möglichen Varianten zu spielen, die sich in der Art des Spielfelds und der Anzahl der Spieler unterscheiden. Die Anwendung kann das Sudoku lösen und die Aufgabe für die Grundvariante nach Schwierigkeitsgrad erstellen.

Schlüsselwörter: sudoku, javascript, nodejs, react js

Obsah

Úvod

1	Sudoku	1
1.1	Typy her	1
1.2	Řešení	4
1.3	Generování	5
1.3.1	Tvorba validní mřížky	5
1.3.2	Tvorba mřížky pro jigsaw variantu	6
1.3.3	Tvorba zadání	7
1.3.4	Tvorba varianty zadání	7
2	Architektura	9
2.1	Autentifikace	9
2.1.1	Local	10
2.1.2	OAuth 2	10
2.1.3	Autorizace	12
3	Backend	13
3.1	Návrh	13
3.2	Databáze	15
3.2.1	Modely	15
3.2.2	Shop	16
3.3	Specifikace REST API	16
3.3.1	CLI pro generátor	18
3.3.2	Testy	19
3.3.3	Unit testy	20
3.3.4	Integrační testy	21
4	Frontend	22
4.1	Struktura souborů	22
4.2	Redux	25

4.3	Přehled funkcí	26
4.3.1	Registrace	26
4.3.2	Přihlášení	27
4.3.3	Herní stránka	27
4.3.4	Hraní různých typů her	28
4.3.5	Obchod	29
4.3.6	Rozehrané hry	30

Závěr

Seznam použité literatury

Seznam obrázků

Úvod

Zadáním projektu bylo vytvořit webovou aplikaci, jejímž primárním účelem je hraní sudoku v klasické podobě i možných variantách. Aplikace dokáže sudoku vyřešit a vygenerovat zadání. Kromě klasického sudoku dokáže také vygenerovat zadání na základě obtížnosti pro všechny typy, u kterých je aktuálně generování podporováno, čímž se práce rozšířila.

Kromě tohoto také aplikace poskytuje základní funkce vnitřní ekonomiky, jako je možnost různých typů přihlášení a nakupování vylepšení za interní měnu.

Co se použitých technologií týče, frontend byl napsán v knihovně ReactJS, backend v NodeJS s frameworkem Express a jako databáze byla použita MongoDB. V každé z těchto částí je využito mnoha knihoven, které jsou specifikovány v souborech `package.json`, jenž nacházejí přímo ve zdrojovém kódu.

1. Sudoku

Tuto hru vymyslel Howard Garns v roce 1979 a publikoval ji v pod názvem „Number Place“. Své velké obliby se dočkala v Japonsku, odkud se později vrátila zpět pod názvem „sudoku“. Ve světě je sudoku vydáváno v mnoha periodikách. U nás jsou to např. Lidové noviny, MF Dnes, Právo, Deník či Metro.

Cílem hry v základní podobě je doplnit chybějící cifry 1 až 9 v zadané, zčásti vyplněné čtvercové tabulce s 9×9 poli. V tabulce jsou zvýrazněny sekce vymezující 9 čtverců (3×3).

K předem vyplněným číslicím je třeba doplnit další číslice tak, aby platilo, že v každém řádku, v každém sloupci a v každém z devíti dílčích čtverců jsou použity vždy všechny číslice jedna až devět, každá právě jednou.

Aritmetická hodnota číslic pro řešení nemá význam, jde pouze o výběr logické řady devíti znaků (v zásadě je možné hrát sudoku i např. s písmeny A–I nebo jakoukoli jinou skupinou devíti symbolů).

Všech různých možností, jak může být hrací pole 9×9 podle těchto kritérií sestaveno, je 6 670 903 752 021 072 936 960, tj. přibližně $6,67 \times 10^{21}$ (6,67 triliard). Pro zvětšující se čtverce je to úloha NP-úplná.¹

1.1 Typy her

Tato aplikace dovoluje uživateli hrát různé varianty sudoku lišící se pravidly pro doplňování čísel do mřížky či samotným tvarem mřížky. Platí přitom, že každá z těchto variant může být spuštěna pro více různých specifikovaných velikostí mřížky a obtížnosti.

Možné velikosti daných variant jsou pevně dané a uživatel si z nich musí vybrat. Nebylo by např. možné sudoku např. o velikosti 36×36 v rozumném čase a kvalitě vygenerovat, natož ji hrát.

Stejně tak je na tom i obtížnost jednotlivých zadání, k dispozici jsou zde obtížnosti: *easy*, *medium*, *hard*. Obtížnost sudoku je v programu reprezentována jako počet políček, které jsou v mřížce zadané. Tento počet je přímo úměrně závislý na velikosti mřížky a

¹WIKIPEDIA. Sudoku. Dostupné z <https://cs.wikipedia.org/wiki/Sudoku>. [cit. 2022 28-2]

je čistě subjektivní. Neexistuje žádný matematický vzorec, který by říkal, že sudoku o velikost N a obtížnosti K má na začátku právě M polí. Tento počet je tedy arbitrárně stanoven na hodnoty ve zdrojovém kódu, které dodávaly nejlepší zadání.

Původně bylo plánováno, že obtížnost bude záviset nikoliv na počtu políček, nýbrž na vztazích mezi těmito políčky, tj. bylo by implementováno řešení lidskými technikami. Takové řešení by však nebylo použitelné i pro jiné, méně časné, typy sudoku. Mimo to, náročnost různých lidských technik je velice subjektivní, počet políček je naproti tomu objektivní ukazatel. Jelikož navíc ukázalo, že toto řešení dává uspokojivé výsledky, bylo od tohoto záměru upuštěno.

Classic

Jedná se o klasické sudoku dle pravidel popsaných na začátku kapitoly. Je možné tuto variantu hrát ve velikostech: 4x4, 6x6, 8x8, 9x9, 10x10, 12x12, 14x14 a 16x16. Číslo zde udává počet políček v jednom řádku.

V případě, že jej nejde odmocnit, aby výsledkem bylo celé číslo, bude menší sekce obdélníkového tvaru. Velikosti boxů jsou pro každou z velikostí pevně specifikovány. Např. pro 12x12 je sice možné vytvořit dva typy sekcí - 2x6 a 3x4, ale aplikace dovoluje hrát pouze 3x4. Například takto vypadá sudoku obdélníkového tvaru pro velikost 8x8.

	3	5	4	1	2	6	
1				4			5
3	8	7	5		4	1	
			1		8	5	
				2		4	
		2		5			7
8	7		6				
	4			8		7	6

Obrázek 1.1: Classic 8x8

V programu je mřížka reprezentovaná pomocí dvourozměrného pole celých čísel o velikosti $N \times N$ specifikované nahoře. Prázdná pole jsou reprezentována pomocí -1. Pro reprezentaci celé

ClassicX

Pravidla u tohoto typu jsou stejná jako u klasického sudoku, nicméně platí zde, že číslo se v hlavní a vedlejší diagonále mřížky musí vyskytovat právě jednou. Například takto vypadá diagonální sudoku pro stejnou velikost 8x8.

		8		4	7	6	
	3						
8	1	6				2	
5		1	4		8	3	
2				8			1
6						7	
7		2		6			
					1	8	7

Obrázek 1.2: ClassicX 8x8

Jigsaw

Tento typ se vyznačuje tím, že sekce zde nemají obdélníkový tvar. V každé sekci se však musí vyskytovat každé číslo právě jednou. Takto vypadá příklad pro jigsaw 9x9:

3	2		9	1				
					8		2	
				7				
	6					7		4
			2		3			
1		9					6	
				2				
	3		5					
				3	7		5	6

Obrázek 1.3: Jigsaw 9x9

V programové reprezentaci nám zde přibývají dvě proměnné:

- **areaPointersGrid** – dvojrozměrné pole o stejné velikosti jako mřížka obsahující prvky 1 - N. Tyto prvky označují číslo sekce, které daný index náleží
- **areasLists** – každá sekce má v tomto listě 1 list obsahující objekty tvaru row, col. Dá se vytvořit z **areaPointersGrid** a slouží k rychlému zkontrolování, zda se v dané sekci nevyskytuje nějaké číslo dvakrát.

1.2 Řešení

Řešení sudoku je v kódu implementováno pomocí kódu v souboru *solvers.js*. Jedná se o klasický backtrackingový algoritmus pracující na principu prohledávání do hloubky. Popis algoritmu:

- potřebné proměnné:
 - N =velikost mřížky
 - mr =mřížka – dvourozměrné pole $N \times N$
- 1. začni na souřadnicích 0, 0,
- 2. pokud $sloupec > N$, $sloupec = 0$, $řádek++$
- 3. pokud $řádek === N$ – ulož řešení a ukonči rekurzi
- 4. číslo – pro všechna čísla od 1 do N :
 - 4.1. zkontroluj, zda dané číslo může být položena na $mr[řádek][sloupec]$
 - 4.2. pokud ano, skoč na krok 2
- 5. vlož -1 na $mr[řádek][sloupec]$ (backtrack)

Tímto způsobem dojde k nalezení všech řešení daného sudoku. Rekurse vždy skončí buď podmínkou v kroku č. 3, popř. doběhnutím metody až za krok 5. Složitost daného algoritmu je $O(M^N)$, kde M je počet možných dosazovaných čísel a N počet políček, za které dosazujeme.

Tato složitost vůbec není dobrá a již pro sudoku 16x16 při určitém čísle N již trvá řešení několik minut. Pro těžké sudoku s malým M dokonce není možné v rozumném čase vygenerovat žádné zadání.

Řešící metoda *solveGeneral* v aplikaci přebírá jako parametr list callbacků, které spouští, aby zkontrolovala, zda může na dané políčko položit dané číslo. Callbackem se myslí metody kontrolující, že v dané části musí být každé číslo právě 1. Jedná se o kontrolu pod-sekce, řádku, sloupce, jigsaw sekci, popř. diagonály. Není tedy třeba psát speciální solver pro každý typ, stačí pouze napsat vstupní metodu, která metodě *solveGeneral* předá příslušné callbacky (např. metoda *startSolvingClassic*)

Všetchna nalezená řešení jsou poté vložena do pole, které bylo metodě předáno jako argument.

1.3 Generování

Program dokáže kromě vyřešení sudoku také zadání vygenerovat. Za validní zadání považujeme takovou mřížku, která má právě jedno řešení.

Algoritmus generování probíhá ve dvou oddělených fázích. V té první dojde k vytvoření validního řešení sudoku pro danou variantu. V té druhé z tohoto řešení odstraníme N čísel, kde N je určeno požadovanou složitostí a kontrolujeme, zda má tato mřížka právě jedno řešení.

1.3.1 Tvorba validní mřížky

Tvorba validní mřížky není až tak jednoduchý úkol, jak by se na první pohled mohlo zdát. Ve skutečnosti je svou složitostí na stejné, ne-li vyšší úrovni než řešení sudoku.

Na tvorbu je použit upravený algoritmus řešení:

1. začni na nultém políčku
2. pokud sloupec $> N$, sloupec = 0, řádek++
3. pokud řádek $=== N$ – ulož řešení a ukonči celý běh metody (zde stačí jen 1 řešení)
4. pro všechna čísla, která se na políčku mohou nacházet v **náhodném pořadí**:
 - 4.1. umístí dané políčko
 - 4.2. vyškrtni zapsané číslo z možností pro políčka v daném řádku, sloupci, boxu / diagonále, na kterých ještě není žádné číslo...
 - 4.3. jdi na krok 2 s dalším prvkem v pořadí (rekurze jako v předchozím případě)
 - 4.4. vlož zpět zapsané číslo z možností pro políčka v daném řádku, sloupci, boxu / diagonále... (rekurze skončila, dané číslo nemůžeme na daném čísle mít – čištění)

V programu je přidána proměnná `possibleNumbersGrid`, což je trojrozměrné pole o velikost $N \times N \times (N+1)$. každý `possibleNumbersGrid[řádek][sloupec]` obsahuje list o $N+1$ prvcích. Tyto prvky reprezentují, kolikrát je dané číslo na dané pozici ohroženo. Např. pokud je `possibleNumbersGrid[řádek][sloupec][2] = 3`, poté to znamená, číslo 2 na pozici řádek, sloupec je invalidní ze 3 důvodů – např. ve stejném řádku je 2, sloupci a boxu. Pokud je zde 0, poté sem číslo položit můžu, pokud něco jiného, tak nesmím.

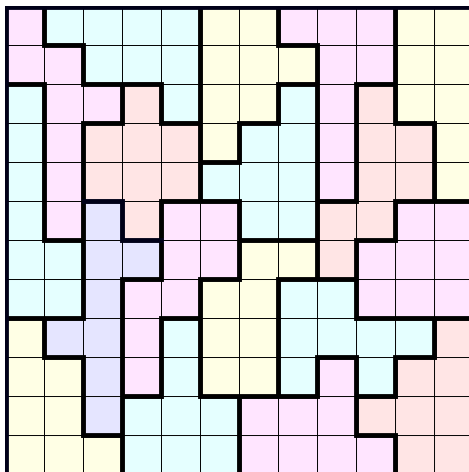
Na první pohled by si člověk mohl říct, že by vlastně šlo pustit sudoku solver, který

pouze vybírá náhodná čísla místo sekvenčního postupu 1 až N na prázdné mřížce, a že není třeba vytvářet nový kód. To by sice šlo, avšak toto řešení je více optimalizované, jelikož škrtnutí možných čísel na daném políčku se provádí pouze pro políčka, která ještě nejsou obsazena. Pro obsazená čísla by to totiž nemělo smysl, jelikož v backtrackingu jsou odebírány až po prvku, kvůli kterému se škrtnutí. Samotný způsob hledání je navíc už z principu více optimalizovaný, jelikož škrtnutí čísel znamená jeden průchod všemi sektory, kterými se má projít, zatímco solver tyto sektory musí projít pro každé číslo, které kontroluje.

1.3.2 Tvorba mřížky pro jigsaw variantu

Jigsaw varianta je specifická v tom, že k fungování potřebuje proměnné `areasLists` a `areaPointersGrid`, které zajistí strukturu mřížky. Předtím, než je tedy možné aplikovat postup z předchozí kapitoly, je tedy nutné rozdělit mřížku do příslušných sekcí a naplnit tak obě proměnné.

Představme si mřížku jako graf, kde jednotlivá políčka jsou uzly a hrany mezi nimi jsou hranami grafu (rohy nejsou hrany grafu). Naším cílem bude rozdělit tento graf do N komponent souvislosti o právě N prvcích, kde N je velikost mřížky



Obrázek 1.4: Ukázka rozdělení jigsaw mřížky do sekcí

Navrhl jsem a implementoval následující algoritmus:

1. rozdělíme mřížku do N validních sekcí, kdy každá sekce bude odpovídat obdélníku, který by byl podsekcí klasického sudoku při dané velikosti (např. pro 8 bychom měli 8 obdélníků velikosti 4×2). Není možné rozdělit sudoku po řádcích

po 8, protože by nebylo možné provést konzistentně další kroky

2. náhodně vybereme dvě sekce
3. jestliže se nedotýkají (aby se dotýkaly, musí alespoň 2 políčka z každé sekce být vedle políčka ze sekce druhé), jdi na krok 2
4. náhodně prohod 1 políčko mezi těmito dvěma sekcemi
5. jestliže vznikla nekonzistence (čísla jedné sekce nejsou v `areaPointersGrid` v jedné komponentě souvislosti), vrať zpět změny z předchozího kroku a znovu ho vykonej
6. vykonávej krok 2 K krát, dokud nebude rozdělení dostatečně dobré (u mě 1000x)

Složitost tohoto algoritmu je poměrně vysoká. Krok 1 bude proveden v čase $O(N^2)$. Krok 2 bude proveden nejhůře v čase $O(N^2)$ a bude proveden K krát. Krok 3 se bude provádět $O(\sqrt{N}\sqrt{N})$, tj. $O(N)$. Krok 4 taktéž. Výsledná složitost bude po vynásobení tedy $O(K * N^3)$

1.3.3 Tvorba zadání

Algoritmus na tvorbu zadání je následující:

1. vygeneruj validní mřížku pro daný typ sudoku
2. zjistí počet čísel, které je potřeba z mřížky odstranit na základě obtížnosti
3. odstraň N náhodných políček ze mřížky
4. vyřeš pro tuto novou mřížku. Pokud má 1 řešení, vrať ho, pokud má více než 1, jdi na krok 3

Ačkoliv se pro těžké sudoku krok 3 několikrát v průměru opakuje, pro 9x9 sudoku se jich stále vygeneruje několik za sekundu.

1.3.4 Tvorba varianty zadání

Vygenerovanou mřížku lze navíc jednoduchými operacemi s kvadratickou časovou složitostí (v závislosti na velikosti mřížky) upravit do podoby, kterou, nedáte-li ji uživateli okamžitě, nebude schopen odlišit od původní mřížky. Konkrétně, z jedné mřížky můžeme vytvořit 5 806 080 takovýchto variant.

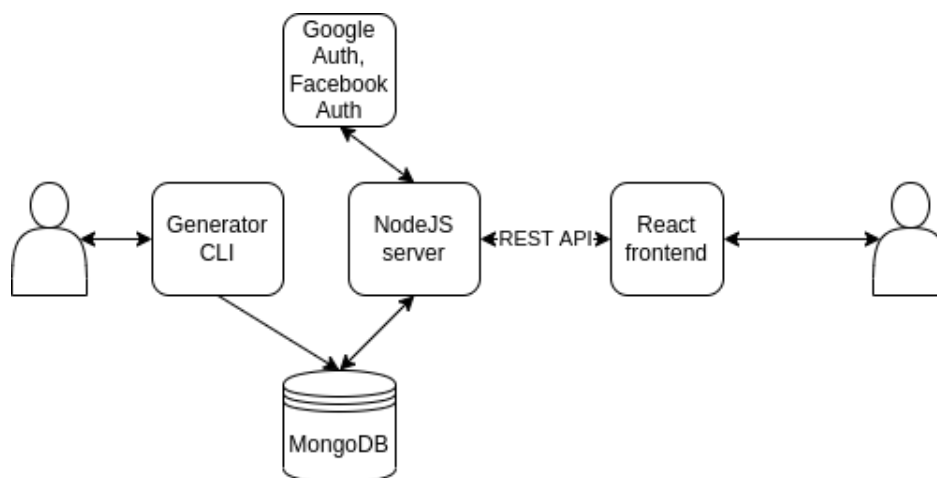
Operace, které můžeme provést jsou:

- permutace čísel – celkem $9!$ možností

- rotace matice – 4 možnosti
- transpozice matice – dle hlavní diagonály, vedlejší diagonály, osy x, osy y – 4 možnosti

Při kombinaci těchto možností vzniká pokaždé odlišné sudoku. Celkem tedy můžeme mít $9! * 4 * 4 = 5806080$ možných variant. O vytváření těchto variant se stará kód v souboru `variationCreator`

2. Architektura



Obrázek 2.1: Architektura

K programu je možné přistoupit ze dvou bodů. Tím prvním je klasický uživatelský přístup z frontendu (viz kapitola 3). Přes REST API komunikuje s backendem (viz kapitola 4). Pomocí tohoto API jsou zpřístupněny funkce jako získání zadání her, autentifikace, změna uživatelských údajů či práce s obchodem.

Tím druhým bodem je CLI interface pro generování her, který slouží k vytvoření her požadovaných uživatelem a jejich uložení do databáze.

Kód pro frontend a backend je pro přehlednost rozdělen do dvou oddělených souborů – client a backend s vlastním package.json souborem. V production módu je poté možnost nechat obě dvě být poskytovány jedním serverem.

2.1 Autentifikace

Tato kapitola pojednává o samotném principu autentifikace v aplikaci. Pro specifikaci autentifikačních API endpointů viz kapitola 3.3 specifikace endpointů.

Autentifikace je proces zjištění totožnosti uživatele. Aplikace dovoluje uživatelům autentifikovat se třemi různými způsoby:

- Local – Přihlášení na stránce pomocí emailu a hesla (vnitřní systém stránky, je nutná předchozí registrace v aplikaci)
- Google Auth (OAuth 2)

- Facebook Auth (OAuth 2)

Pro implementaci tohoto procesu na backendu bylo využito knihovny passport.js, která celý proces usnadňuje. Základ kostry byl použit z našeho dřívějšího projektu AvAvA¹, nicméně prošel velkým refactoringem a předěláním do databáze MongoDB.

Pro každou ze tří strategií je nutné zapsat kód, který se zavolá hned po přihlášení, tedy uložení do databáze - `passport.use(...)` a předat passportu `clientId` a `secretId` daného provideru.

Dále je potřeba implementovat metody `serializeUser` a `deserializeUser`, které se budou volat po popořadě při zapisování ID do sessioncookie a při získávání uživatele z databáze za pomoci ID získané ze sessioncookie.

Pro zjištění, zda je uživatel přihlášen je možné na kterémkoliv z request objektů získat property `req.user`, ve které bude uložen výsledek posledního volání metody `deserializeUser`. V případě, že žádný uživatel není přihlášen, jeho hodnota je `null`.

2.1.1 Local

Aby tuto strategii mohl uživatel použít, musí se nejprve zaregistrovat. Mezi požadované údaje patří první a druhé jméno, email a dostatečně silné heslo. To by mělo obsahovat alespoň 8 znaků, alespoň 1 velké písmeno, alespoň 1 číslici a alespoň 1 speciální znak. Toto je validováno jak na frontendu, tak i backendu.

Tento typ autentifikace se plně odehrává v rámci aplikace a registrovaný uživatel je uložen do MongoDB databáze. Platí přitom, že není možné zaregistrovat uživatele, pokud se v databázi již uživatel s daným emailem nachází.

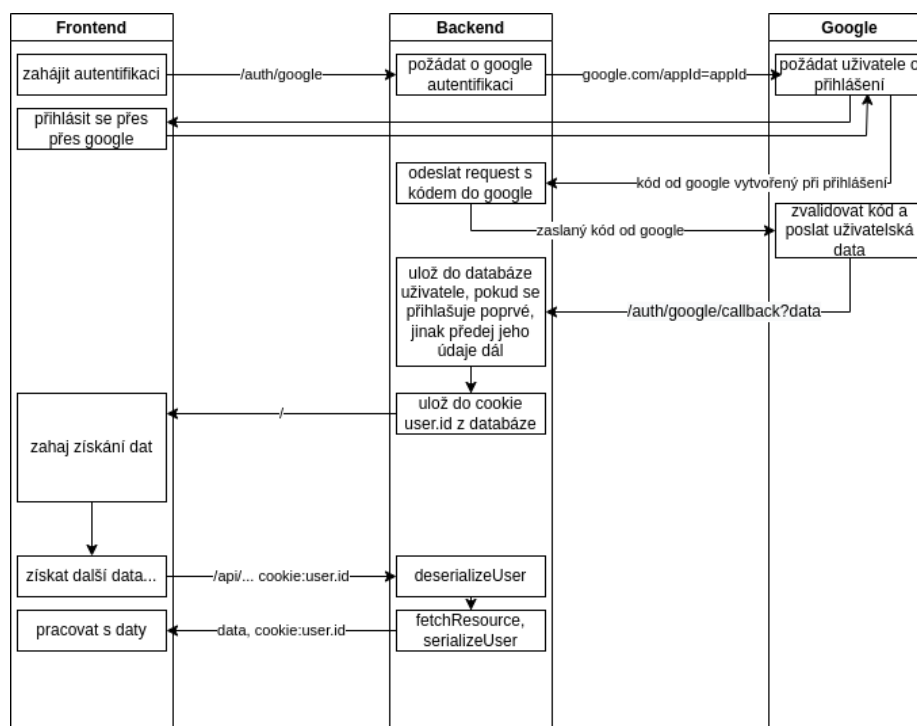
2.1.2 OAuth 2

Díky OAuth 2 protokolu je možné se autentifikovat prostřednictvím poskytovatelů společností Google a Facebook. Pro zprovoznění Google Auth, resp. Facebook Auth je nejprve nutné si v Google Console, resp. Facebook Devtools vytvořit nový projekt. Odtud se zkopíruje `clientId` a `clientSecret`.

Není zde rozdíl mezi registrací a loginem. Jestliže se uživatel přihlašuje poprvé, bude uživatel v databázi vytvořen, jestliže je přihlašuje po několikáté, bude přihlášen.

¹VLADIMÍR, V., JOSEF, L. a KRYŠTOF, H. (2021). Avava. Technical report, Gymnázium, Praha 6, Arabská 14

Jestliže se při prvním přihlášení již v databázi nachází uživatel s daným emailem, je jeho záznam updatován o data z provideru. Daný uživatel se bude moci přihlásit jak svým původním způsobem, tak i novým způsobem. Data jako obrázek či první a druhé jméno zůstanou zachována z minula, aby vše bylo konzistentní. Uživatel se tedy může dostat do situace, kdy se může přihlásit ke stejnému účtu jak přes google, tak přes facebook, tak i pomocí local strategy



Obrázek 2.2: Autentifikace přes Google Auth

Na obrázku výše můžete vidět průběh autentifikačního procesu. Nejprve uživatel musí autentifikaci zahájit, což udělá pokusem o získání nějakého chráněného obsahu. Po odeslání žádosti z klienta na backend přesměruje uživatele na přihlašovací stránku Google. V případě, že se student přihlásí poprvé, musí odsouhlasit, že povoluje programu využívat některé služby. Při dalších návštěvách se už jen přihlásí.

Tím je proces autentifikace hotov a začíná proces autorizace programu u Googlu. Aplikace tedy bude Google žádat, aby jí předal data o uživateli, která potřebuje. To dělá v obdélníku “odeslat request s kódem do google”. Pokud google vyhoví, zavolá callback, který mu byl nastaven při vytváření projektu v Google Console a odešle na něj chtěná data.

Následně je vytvořena session mezi frontendem a backendem. Každá zpráva odted

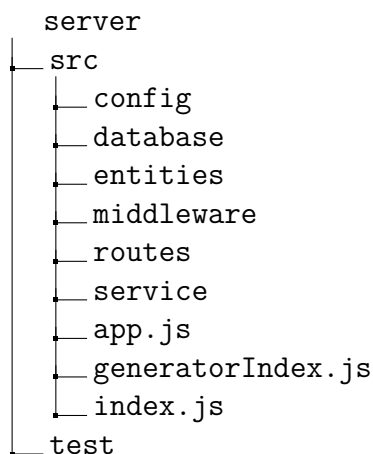
bude v hlavičce obsahovat cookie soubor s ID uživatele.

Po získání dat je uživatel přihlášen a může systém používat. Při odhlášení je session ukončena a už se dále user ID neposílá.

2.1.3 Autorizace

API endpointy je možné pomocí middleware `requireAuth` zajistit proti zavolání bez toho, aby byl uživatel přihlášen. Ostatní kontrolování, zda má uživatel na něco právo jsou subjektivní a jsou popsány u daného logického celku.

3. Backend



Obrázek 3.1: Struktura souborů na backendu

Vstupním bodem aplikace je soubor `index.js` pro skript `npm start`. Na základě dat ze složky `config`, kde se nacházejí `env` a `.env.test` soubory s globálními proměnnými inicializuje server ze složky `app.js` a databázi ze složky `database`. Ve složce `routes` a `middleware` je vše, co zodpovídá za REST API v `entities` jsou poté základní objekty ukládající data, která jsou všude na serveru potřeba a ve složce `service` je poté samotná business logika aplikace. Soubor `generatorIndex.js` je poté vstupním bodem při generaci sudoku příkazem `npm run generate`. Kód `test` sousedící s `src` poté slouží k testování logiky z `src`.

3.1 Návrh

Kód na backendu byl napsán tak za využití některých prvků z čisté architektury od Boba Martina ¹. Kód dělím do několika vrstev od nejvyšší po nejnížší:

1. vnější rozhraní (MongoDB interface, REST API, generator CLI)
2. logické komponenty – solvers, variationCreator, generators
3. entity – objekty reprezentující základní data naší aplikace (user, games). Jejich specifikace nemá smysl uvádět, jsou v kódu dobře popsány.

¹viz. *Bob Martin's Clean Architecture*. Dev Mastery. Dostupné z https://www.youtube.com/watch?v=CnailTcJV_U&ab_channel=DevMastery. [cit. 2022 28-2]

Klíčové je pro ni využití návrhového vzoru **dependency injection (IOC)**. Tento způsob psaní je typický tím, že kód z nižší vrstvy nesmí být silně závislý na kódu z té vyšší. Silnou závislostí mám na mysli závislost vytvořenou importováním. Může na ni být závislý pouze slabě, tj. danou funkcionalitu předá vyšší vrstva té nižší při vytvoření jako parametr. Výhody tohoto přístupu:

- Odolnost vůči změnám. Nižší vrstva není na ničem přímo závislá a tak při změnách v knihovnách či vyšších vrstvách a v případě správného použití adaptérů netknutá.
- Možnost vytvoření velmi spolehlivého jádra aplikace, které má výborné testové pokrytí. Díky tomu v této části prakticky nemohou vznikat chyby.
- Možnost jednoduše měnit závislosti. Např. v případě testování můžeme jednoduše předat našemu REST API serveru pomocí dependency injection testovou databázi.

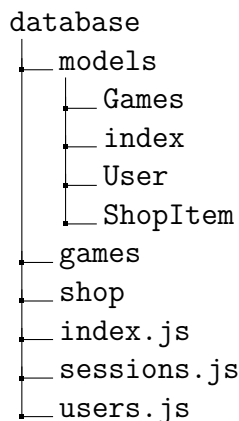
Entity a některé další části kódu – např. databáze jsou vytvářené pomocí **tovární metody** – např. `makeUser()`. Ta vrátí objekt uživatele, k jehož vlastnostem je možné přistupovat pouze pomocí getterů, setterů a několika dalších metod. Zároveň provede kompletní validaci, aby se zabránilo, že objekt bude v invalidním stavu.

Tím, že se s objektem pracuje pomocí metod je však zkomplikovaná práce při posílání dat na klienta přes REST API a posílání dat do databáze. To je však správně, v každém z těchto případů jsou potřeba pouze některé z vlastností, kterých může entita nabývat. Určitě bychom např. nechtěli na klienta posílat hash hesla z databáze. Je tedy před posláním potřeba entitu převést na objekt, který je pro danou situaci potřeba převést tento objekt na klasický javascript objekt bez funkcí. Na to použijeme návrhový vzor **adaptér**. Adaptéry pro API jsou přímo v entitách, zatímco adaptéry pro databáze jsou v databázových souborech.

Aby tomuto bylo předány potřebné závislosti, jsou tovární metody entit obaleny další tovární metodou `buildMakeUser`. Té jsou předány potřebné závislosti a její návratovou hodnotou je callback na tovární metodu vyrábějící dané entity.

Tyto `buildMake` metody jsou volány z příslušného `index.js` souboru, který se nachází v dané složce. Takovýto `index.js` soubor se vyskytuje v každé složce, která odpovídá za nějakou oddělenou činnost a slouží k sjednocení přístupu k daným funkcím.

3.2 Databáze



Obrázek 3.2: Soubory týkající se databáze

Jako databáze byla vybrána MongoDB, jelikož umožňuje pohodlnou reprezentaci polí a v programu se ani v dlouhodobém horizontu nevyskytuje mnoho použití relací.

Ve složce `models` je specifikována struktura modelů pro vytvoření MongoDB kolekcí. V každém z těchto souborů se kromě modelu vyskytuje také metoda vytvářející daný databázový objekt z předané entity. Jedná se o návrhový vzor **adaptér**, jelikož databázové logice jsou předávány entity, nikoliv databázové objekty.

Mimo složku `models` jsou poté soubory obsahující různé metody, které nad těmito modely operují. Jedná se o metody jako `findRandomClassicGame` či `findUserByEmail`. Všechny tyto metody jsou poté exportovány ze souboru `index.js`, ve kterém se zároveň nastavuje celá mongo databáze. Jedná se o návrhový vzor **fasáda**, jelikož vytváříme zjednodušující vrstvu pro práci s databází.

Díky této zjednodušující můžeme kdykoliv databázi vyměnit a zbytek programu nebude nijak ochromen. Stačí nám poté pouze změnit implementaci těchto zmíněných fasádních metod.

3.2.1 Modely

User

Ve schématu se nachází: `first_name`, `last_name`, `email`, `coins_count` (peníze uživatele), `profile_picture_link` (odkaz na obrázek profilovky), `auth` – autentifikační data (`hash` hesla pro local, `id + accessToken` pro facebook a google). Obsahuje také

proměnné pro uchovávání informací o jeho nákupech a rozehraných hrách. Rozehraných her je pouze určitý maximální počet, který je hlídán pomocí kontrolerů.

ClassicGame

Ve schématu se nachází: **seed** – mřížka se zadáním (dvojměrné pole NxN s -1 jako prázdnými místy), **solution** – mřížka s řešením, **difficulty** – obtížnost

Tato reprezentace je výhodná v tom, že sudoku, které pochází z databáze není nutné již řešit, jelikož řešení je zde již uloženo. Když si tedy na frontendu někdo vyžádá vyřešení dané úlohy, jednoduše se jen zobrazí **solution**.

Tento model slouží jako základ pro většinu dalších modelů, protože většinou potřebují podobné typy na uložení.

Každý typ má vlastní tabulku hlavně z důvodů výkonnosti, aby při hledání daného typu hry nebylo nutné procházet i typy, které s ním vůbec nesouvisí, ale také z důvodu, že hry nemusí mít stejné schéma – jigsaw.

ClassicXGame

Stejné schéma jako ClassicGame

JigsawGame

Schéma z classicGame, ale navíc **areaPointersGrid** (již vysvětleno dříve)

3.2.2 Shop

Schéma obsahující informací o předmětech, které lze zakoupit v obchodě. Obsahuje proměnné: **price** – pro cenu, **description** – pro popis předmětu, **name** – jméno a volitelně také **imageUrl** – odkaz na obrázek předmětu

3.3 Specifikace REST API

Tato kapitola se věnuje popisu REST API určené pro komunikaci mezi klientem a serverem.

POST – /api/auth/register

body: email, firstName, lastName, password

V případě, že uživatel předal všechny potřebné parametry, heslo je dost silné a v databázi se daný uživatel ještě nevyskytuje, je uživatel zaregistrován.

POST – /api/auth/login

body: email, password

V případě, že uživatel předal všechny potřebné parametry, uživatel se v databázi vyskytuje a heslo je správné, je uživatel přihlášen pomocí PassportJS.

GET – /api/auth/google

Vstupní bod pro počátek google autentifikace – dále mimo kontrolu aplikace až do vrácení dat od googlu. Ta jsou poté předána do PassportJS middleware

GET – /api/auth/facebook

Vstupní bod pro počátek google autentifikace – dále mimo kontrolu aplikace až do vrácení dat od facebooku. Ta jsou poté předána do PassportJS middleware

POST – /api/auth/logout

Ukončí aktuální session

GET – /api/user

Vrátí objekt aktuálního uživatele, pokud je přihlášen. Jestliže není, vrátí 401

POST – /api/user/password

body: oldPassword, newPassword API endpoint pro změnu hesla uživatele. Pokud je oldPassword stejné jako staré heslo a newPassword je dostatečně silné, dojde ke změně hesla v databázi. Po získání zadání z databáze vytvoří variantu hry a tu poté odešle na klienta.

GET – /api/games/classic

body: size, difficulty API endpoint pro vracení zadání typu classic o dané velikosti a složitosti. V případě, že pro dané zadání není v databázi žádný záznam, vrátí 500. Po získání zadání z databáze vytvoří variantu hry a tu poté odešle na klienta.

GET – /api/games/classicX**GET – /api/games/jigsaw****GET – /api/shop/**

API endpoint určený pro získání všech předmětů, které lze zakoupit v obchodě

GET – /api/shop/buy

API endpoint, pomocí něhož proběhne nákup daného předmětu. Vrací objekt přihlášeného uživatele, jestliže byl updatován state, jinak vrátí 400.

3.3.1 CLI pro generátor

Tato aplikace podporuje i generování her, které trvá vygenerovat i delší dobu než je několik minut. Je proto dobrý nápad nenechat aplikaci generovat hru samotnou, ale nechat administrátora, aby sudoku nechal na požadavek manuálně vygenerovat.

Generátor se zapíná na backendu příkazem `npm run generate`.

Na začátku se uživatelé zeptá na otázku:

```
Which of these game types would you like to generate? Type only the number preceding name. Type only one!
```

- 1) Classic
- 2) ClassicX
- 3) Jigsaw

Odpovědí by mělo být číslo daného typu. Pokud tomu tak není, zeptá se znovu.

Následně se dle předaného typu zeptá na velikost.

```
Select size of Classic
```

- 1) 4x4
- 2) 6x6
- 3) 8x8
- 4) 9x9
- 5) 10x10
- 6) 12x12
- 7) 14x14
- 8) 16x16

Opět se očekává číslo před pravou kulatou závorkou. Cokoliv jiného vyústí v zopakování otázky.

Následně je vyžádána obtížnost úlohy:

```
Select difficulty of Classic
1) easy
2) normal
3) hard
```

Úplně nakonec si CLI vyžádá počet sudoku na vygenerování (limitováno od 1 do 200 – čísla vybrána arbitrárně na základě zkušenosti):

```
How many games of type Classic with size 9x9 and difficulty hard (1-200)
```

Jakmile je vše vybráno, začne se daný počet generovat. Jakmile bude vždy 1 sudoku vygenerováno a uloženo, vypíše se uživateli zpráva:

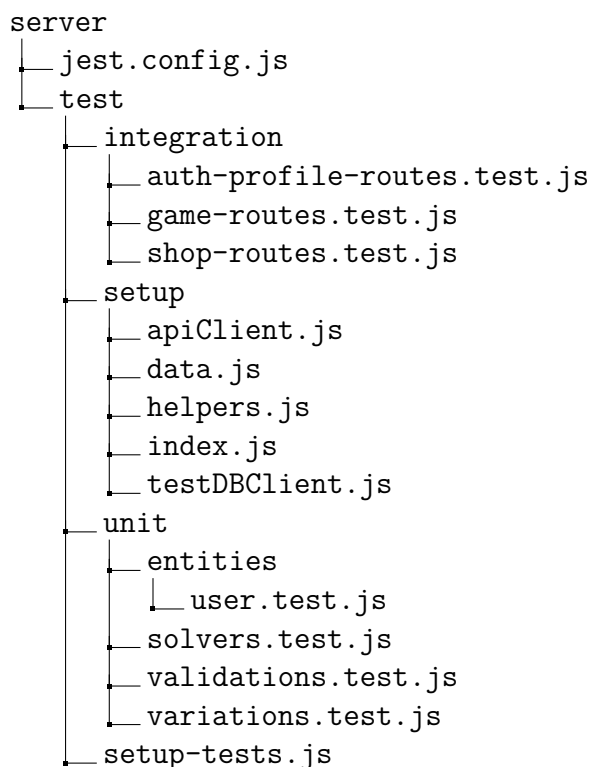
```
49. sudoku generated and stored
50. sudoku generated and stored
```

Během tohoto generování může uživatel psát znova, jelikož se generátor vrací v nekonečné while smyčce na první dotaz. Není tedy třeba znovu skript spouštět. Jelikož se ukládání provádí asynchronně, není třeba na čekat, než se vše dogeneruje.

3.3.2 Testy

Na backendu bylo možné postupovat stylem test-driven development. Jádro tohoto postupu spočívá v tom, že se nejdříve napíše test a jeho výsledek a až následně se píše samotný kód.

Na testování byl použit framework Jest. Pro testování API knihovna supertest-session, což je wrapper pro supertest podporující session. Testy využívají testovací databázi, která má stejnou strukturu jako ta produkční, akorát je k jejímu jménu přidán postfix test.



Obrázek 3.3: Souborová struktura testů

Soubor `jest.config.js` zajišťuje správnou interpretaci absolutních cest na backendu, kterého je docíleno pomocí závislosti na knihovně `sexy-require`. Zároveň specifikuje setup file – `setup-tests.js`, jehož role je připravit proměnné prostředí (`.env.test`)

Ve složce `setup` se nacházejí soubory s často používaným kódem – např. vkládání testovacích prvků do databáze (`testDBClient.js`), časté API requesty (`apiClient.js`), opakující se testovací data (`data.js`) či různé pomocné metody jako kontrola rovnosti mřížek sudoku (`helpers.js`). To vše je exportováno z `index.js`, kde se vše kombinuje a vytváří se zde instance testovacího serveru.

3.3.3 Unit testy

Unit testy slouží k testování jedné, často velmi malé funkcionality. Testují se zde správná funkčnost řešení sudoku (`solvers.test.js`), správná funkčnost variačních technik jako transpozice apod. `variations.test.js` či správná funkčnost validací – silné heslo či nikoliv (`validators.test.js`).

Zároveň se zde také testuje, zda správně funguje vytváření entit a zda je nelze dostat do nekonzistentního stavu. Zatímco u uživatele na toto je speciální soubor

(`user.test.js`), u her je toto prováděno implicitně, jelikož metody na vytváření entit jsou použity v souboru `data.js` kdyby tedy bylo něco v nepořádky, testy by selhaly zde.

3.3.4 Integrační testy

Integrační testy testují několik komponent dohromady. V případech této aplikace se jedná o testování API, v jehož rámci zároveň dochází i k testování databáze.

Pro testování API pro autentifikaci a uživatelské akce je použit soubor (`auth-profile-routes.test.js`) zatímco pro herní API je použit `game-routes.test.js`

4. Frontend

Frontend byl napsán v Javascriptu za pomoci knihovny ReactJS. Vzhled byl zajištěn pomocí preprocesoru SASS. Zároveň bylo masivně využito knihovny Material-UI, která zajišťuje příjemný jednoduchý vzhled.

4.1 Struktura souborů

Následující sekce popisuje strukturu souborů klienta.

```
client
├─ package.json
├─ jsconfig.json
├─ public
└─ src
```

Obrázek 4.1: Struktura souborů frontend

Soubor `package.json` obsahuje informace o použitých knihovnách. Soubor `jsconfig.json` je soubor struktury konfiguračního projektu pro typescript, který ovšem používáme i bez typescriptu, jelikož nám umožňuje používat absolutní adresování v projektu. Složka `public` je vygenerovaná od utility `create-react-app` a slouží pro vygenerování React stromu do HTML souboru. Složka `src` poté obsahuje veškerý náš kód.

Src

```
client
├─ src
│   ├── api
│   ├── assets
│   ├── components
│   ├── config
│   ├── redux
│   ├── games.js
│   ├── history.js
│   ├── index.svg
│   ├── routes.js
│   └── setupProxy.js
```

Obrázek 4.2: Struktura souborů frontend – src

Zdaleka nejdůležitější je ovšem `src` složka, která obsahuje samotný kód. Vstupním bodem je zde `index.js`, který renderuje aplikaci pomocí ReactDOM. Soubor `routes.js` obsahuje text jednotlivých cest na frontendu, aby byly centralizované na jednom místě. Soubor `history.js` obsahuje objekt `history` pro `connected-react-router`.

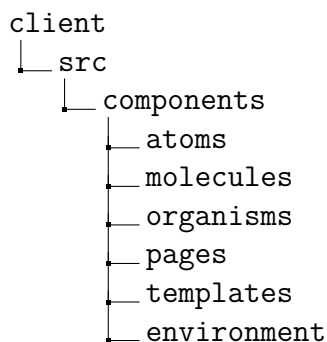
Soubor `games.js` poté obsahuje veškeré informace pro centralizaci informací o různých herních typech. Na frontendu totiž není možná stejná práce s entitami jako na backendu (hlavně kvůli Reduxu), popř. je to silně nevýhodné. K datům je zde lepší přistupovat nikoliv pomocí getterů a setterů, ale přímo (viz Redux). Informace je tedy nutné centralizovat jinde než entitách.

Api

Ve složce `api` se nachází veškerý kód potřebný ke komunikaci s REST API pomocí klientské knihovny `axios`.

Jelikož při developementu nejsou backend a frontend na jednom portu, dochází ke CORS erroru. Na jeho vyřešení byla implementována `http proxy` (soubor `setupProxy.js`), která veškeré specifikované requesty přesouvá na port 5000 se stejnou adresou. Jedná se o návrhový vzor **proxy**.

Components



Obrázek 4.3: Struktura souborů – components

Složka `components` obsahuje veškeré komponenty, ze kterých je aplikace složena. Je zde ctěna architektura **atomic design**. Ten říká, že komponenty je možné rozdělit do 5 skupin:

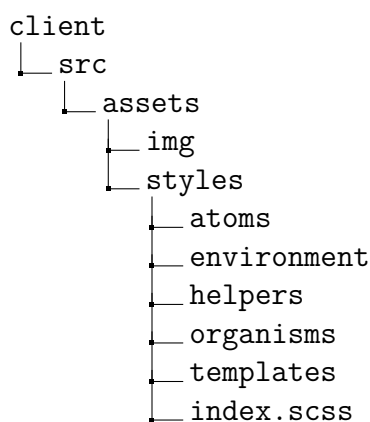
- **atoms** – prvky, které používá více různých komponentů jiné funkce a nelze je dále

dělit – tlačítka, ikonky (zde se kvůli silnému využití material ui téměř nevyskytují)

- **molecules** – skupiny atomů, které používá více komponentů jiné funkce (zde se kvůli silnému využití material ui téměř nevyskytují)
- **organisms** – skupiny molekul a atomů, které používá alespoň 1 stránka – většina komponent aplikace
- **templates** – šablona pro stránku (např. NormalPage pro stránky se Sidebarem a Navbarem)
- **pages** – stránky dostupné na jednotlivých routách

Složka **environment** poté obsahuje pouze **App.js** komponentu, která dává vše dohromady a využívá **connected-react-router**

Assets



Obrázek 4.4: Struktura souborů frontend – assets

Ve složce **assets** se nachází vše, co se týká stylů, obrázků a fontů potřebných pro projekt. Kód napsaný v **assets/scss** se díky preprocesoru zkompile do **css** souborů do složky **assets/css**, které se poté odesílají spolu s **HTML** souborem.

Styly kopírují strukturu **atomic designu**, přičemž navíc je zde složka **helpers**, ve které jsou soubory pro pomocné **SASS** funkce, proměnné a **mixiny**.

Utils

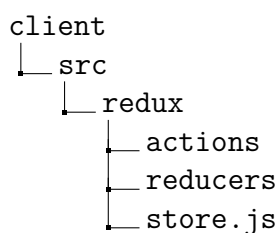
Ve složce service se nacházejí služby, které mohou využívat jakékoliv komponenty. Jedná se o různé druhy validací, notifikace apod.

Ve složce views sídlí jednotlivé stránky. Ty jsou v ní dále rozděleny podle toho, do jaké kategorie spadají. Zde jsou vidět všechny tři kategorie. Ve složce Other se poté nacházejí stránky z templatu pro případ, kdy by bylo možné někdy použít některé z template komponentů.

4.2 Redux

Redux je framework, který dovoluje různým React komponentům sdílet stav pomocí tzv. Redux store. Obdoba návrhového vzoru **kontext** ze Spring frameworku. V případě, že nějaký komponent potřebuje přistoupit k nějaké části storu, tak se napíše metoda mapStateToProps a jednotlivé proměnné jsou poté předány jako properties jednotlivým komponentám. V případě, že dojde ke změně redux storu, dojde k překreslení daného komponentu.

Se storem je možné manipulovat pouze pomocí tzv. akcí. Jedná se o objekty, které obsahují jméno a payload. Tyto akce se nacházejí ve složce actions v odlišných souborech pro uživatelské akce, akce s projekty, akce s uživateli a frontendové akce. Tyto akce mohou být tzv. dispatchnuty, čímž se dostanou k reducerům.



Obrázek 4.5: Struktura souborů frontend – actions

Reducer je metoda, která na základě jména a payloadu předané akce určí, co se má stát s Redux storem. Tento způsob změny stavu se může zdát zvláštní, každopádně zajišťuje vynikající škálovatelnost systému. Aktuálně se zde nacházejí dva reducery – **user** pro zajištění dat o uživateli a **games** pro zajištění dat o hrách.

Na games reduceru se nachází několik důležitých proměnných. První z nich je `currentlyPlayed`. Ta obsahuje řetězec označující typ aktuálně hrané hry.

Poté se zde nachází pro každý typ hry objekt, do něhož se ukládají data pro aktuálně hranou hru. Je tedy možné mít více rozehraných typů her najednou a volně mezi nimi přeskakovat.

Akce mohou být i asynchronní, což se např. hodí pro případ, kdy chceme poslat request na API a jakmile získáme výsledek, tak udělat se získanými daty nějakou akci. Na toto používáme `redux-thunk`.

Jednotlivé komponenty poté přistupují k datům pomocí `useSelector` hooku.

Hlavní účel využití Reduxu v projektu je tedy zpracování dat, které se získají z API do podoby, se kterou mohou komponenty pracovat a poskytnutí možnosti tato data měnit s okamžitými účinky na frontendu.

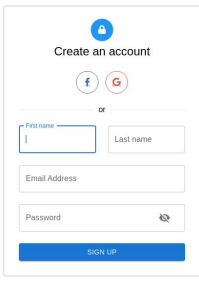
4.3 Přehled funkcí

Tato kapitola ukazuje základní přehled o funkcích frontendu

4.3.1 Registrace

Registrační stránka obsahuje možnost registrovat se pomocí Googlu a Facebooku pomocí dvou kulatých tlačítek, popř. pomocí formuláře. Je zde také odkaz na přihlašovací stránku vpravo dole.

Je zde použito validace, takže se nemůže stát, že by uživatel odeslal invalidní formulář. Toho je docíleno pomocí knihoven `React Formik` a `Yup`.

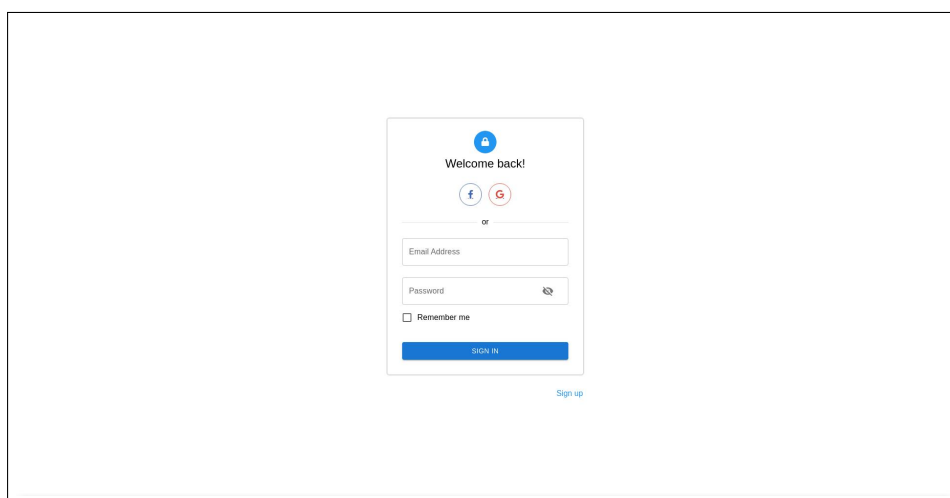


Obrázek 4.6: Registrační stránka

4.3.2 Přihlášení

Přihlašovací stránka obsahuje možnost přihlásit se pomocí Googlu a Facebooku pomocí dvou kulatých tlačítek, popř. pomocí formuláře. Je zde také odkaz na registrační stránku vpravo dole.

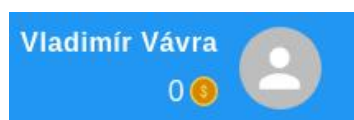
I zde je použito validace, takže se nemůže stát, že by uživatel odeslal invalidní formulář. Toho je docíleno pomocí knihoven React Formik a Yup. Remember-me a forgot password funkcionality však zatím implementovány nejsou, takže jsou spíše na okrasu.



Obrázek 4.7: Přihlašovací stránka

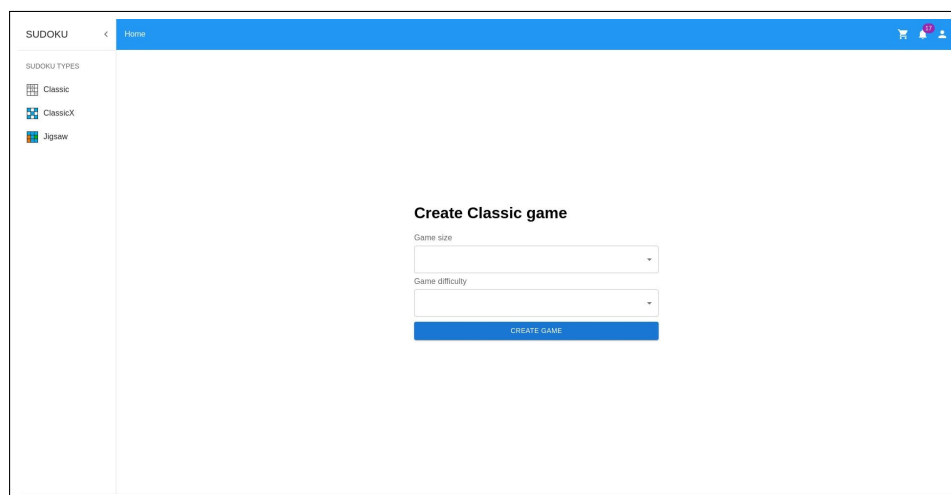
4.3.3 Herní stránka

Normální stránka obsahuje navbar obsahující informaci, jakou hru aktuálně uživatel hraje (pokud žádnou nehraje, nezobrazí nic). a také uživatelské informace. Jestliže uživatel není přihlášený, zobrazí se ikona anonymního uživatele, jelikož se načítá, zobrazí se spinner a jestliže je přihlášený, zobrazí se něco takového:



Obrázek 4.8: Přihlášený uživatel

Stránka obsahuje také zatahnutelný sidebar, pomocí kterého se vybírají hry. Při kliknutí na jednu z možností se zobrazí takovéto menu:



Obrázek 4.9: Stránka s výběrem her

Pomocí select boxů se zde dají nastavit velikost hraného sudoku a obtížnost. Po nastavení a kliknutí na modré tlačítko se dispatchne akce, která se serveru zeptá na daný typ sudoku. Jestliže dostane odpověď s daty, vloží do Redux storu na dané místo data pro hru a je možné hrát. Jestliže dostane negativní odpověď, dispatchne se notifikace, že pro dané nastavení není na serveru žádné sudoku, popř. jinou chybu. Do doby, než přijde odpověď se zobrazuje spinner.

4.3.4 Hraní různých typů her

Aplikace dovoluje uživateli sudoku hrát a zároveň mu je schopna napovídat, zda porušuje pravidla sudoku. Takto např. vypadá, když je poruší:

5	9		2			3		
1		2	6		9	7		
8	6							1
6			4	5		1	3	
	1	5	8			6		
		4	3			5	8	
4	3				8			7
2		4				8	6	3
7			1	6	3		2	

Obrázek 4.10: Porušení pravidel sudoku

Fokus mezi jednotlivými políčky je možné předávat pomocí šipek či myši. Čísla jsou vyplňována klávesnicí a prázdné políčko je vyplněno pomocí delete či backspace. Není

možné přepsat předvyplněné políčko.

Tohoto všeho se dosahuje pomocí dispatchování změny hracích dat do reduxu po každé, co se přidá nové číslo. Musí se tedy vyrenderovat celé sudoku znovu. To sice vypadá jako plýtvání výkonem, nicméně pro každý tah se musí celé sudoku zvalidovat a v případě nekonzistence se musí ukázat nepřesnosti. Tak jako tak se tedy musí celá mřížka přerenderovat.

Zároveň je tento postup dobrý v tom, že je díky němu možné jednoduše přidat tlačítkový interface, kdy při kliknutí na tlačítko by se na fokusované políčko vložilo dané číslo.

4.3.5 Obchod

Tabulka s obchodem obsahuje veškeré předměty, které lze zakoupit. Základ tabulky jsem získal z ¹. Na tom jsem provedl poměrně rozsáhlé úpravy. Aktuální funkcemi obchodu jsou vyhledávání na základě podobnosti s textem pomocí fuzzy finderu, stránkování dat a samozřejmě možnost nakoupit daný předmět.

Name	Description	Price	Buy
Classic12x12Hard	Classic hard sudoku grid of size 12x12	4	🛒
Classic14x14Hard	Classic hard sudoku grid of size 14x14	6	🛒
Classic16x16Medium	Classic medium sudoku grid of size 16x16	8	🛒
Classic18x18Hard	Classic hard sudoku grid of size 18x18	10	🛒
ClassicX12x12Hard	ClassicX hard sudoku grid of size 12x12	4	🛒
ClassicX14x14Medium	ClassicX medium sudoku grid of size 14x14	5	🛒
ClassicX14x14Hard	ClassicX hard sudoku grid of size 14x14	6	🛒
ClassicX16x16Medium	ClassicX medium sudoku grid of size 16x16	8	🛒
ClassicX18x18Hard	ClassicX hard sudoku grid of size 18x18	10	🛒
Jigsaw12x12Hard	Jigsaw hard sudoku grid of size 12x12	4	🛒

Obrázek 4.11: Tabulka obchodu

Jestliže má uživatel dostatek peněz na nákup, zobrazí se mu dialog, ve kterém bude mít uživatel možnost nastavit si počet daných kupovaných předmětů. V opačném případě je vyhozena notifikace, že se akce nezdařila.

¹Material-UI. react-table. Dostupné z <https://react-table.tanstack.com/docs/examples/material-ui-components>. [cit. 2022 28-2]

4.3.6 Rozehrané hry

Tabulka rozehraných her umožňuje uživateli pokračovat v rozehrané hře. Každá hra, která je na frontendu spuštěna, je automaticky uložena a bude v tabulce dostupná. Obsahuje datum vytvoření hry, datum posledního hraní hry, typ hry a obtížnost. Kromě toho je také možné mazat záznamy.

Závěr

Z práce se povedlo splnit vše z povinné části a kromě toho byly přidány některé funkce, které jsem původně nezamýšlel, jako např. možnost pokračovat v uložené hře. Ačkoliv se jedná o projekt, který svým účelem nemá sám o sobě praktické použití, rozhodně mělo jeho vypracování smysl, jelikož mi poskytlo některé nové znalosti o testování, návrhových vzorech a dalších technologiích. Kromě toho jsem zde vytvořil znovu-použitelné komponenty, které je možné kdykoliv použít v programování jiných projektů (autentifikace, komponenty na frontendu, ...)

Seznam použité literatury

Bob Martin's Clean Architecture. Dev Mastery. Dostupné z https://www.youtube.com/watch?v=CnailTcJV_U&ab_channel=DevMastery. [cit. 2022 28-2].

Material-UI. react-table. Dostupné z <https://react-table.tanstack.com/docs/examples/material-ui-components>. [cit. 2022 28-2].

VLADIMÍR, V., JOSEF, L. a KRYŠTOF, H. (2021). Avava. Technical report, Gymnázium, Praha 6, Arabská 14.

WIKIPEDIA. Sudoku. Dostupné z <https://cs.wikipedia.org/wiki/Sudoku>. [cit. 2022 28-2].

Seznam obrázků

1.1	Struktura projektu	2
1.2	ClassicX 8x8	3
1.3	Jigsaw 9x9	3
1.4	Ukázka rozdělení jigsaw mřížky do sekcí	6
2.1	Architektura	9
2.2	Autentifikace přes Google Auth	11
3.1	Struktura souborů v src na backendu	13
3.2	Soubory týkající se databáze	15
3.3	Souborová struktura testů	20
4.1	Struktura souborů frontendu	22
4.2	Struktura souborů frontendu – src	22
4.3	Struktura souborů – components	23
4.4	Struktura souborů frontendu – assets	24
4.5	Struktura souborů frontendu – redux	25
4.6	Registrační stránka	26
4.7	Přihlašovací stránka	27
4.8	Přihlášený uživatel	27
4.9	Stránka s výběrem her	28
4.10	Porušení pravidel sudoku	28
4.11	Tabulka obchodu	29