Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Webové šachy pomocí Socket.IO technologie

Bc. Ondřej Synek

Diplomová práce
2017

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2016/2017**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Ondřej Synek**
Osobní číslo:	**I14285**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Webové šachy pomocí Socket.IO technologie**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

Cílem diplomové práce je návrh a implementace webové aplikace pracující v reálném čase, která představuje elektronickou podobu deskové šachové hry.
V teoretické části práce bude provedena rešerše technologií použitých při vývoji aplikací podobného typu s důrazem na komunikaci klient-server.
Aplikace bude využívat vzájemné komunikace dvou klientů pomocí technologie Socket.IO a bude dále postavena na technologiích Node.js, React a NoSQL databázi MongoDB.

Rozsah grafických prací:

Rozsah pracovní zprávy: **cca 65 stran**

Forma zpracování diplomové práce: **tištěná**

Seznam odborné literatury:

MARDAN, Azat. Practical Node.js: Building Real-World Scalable Web Apps. Vyd 1. Apress, 2014, 300 s. ISBN 978-1-4302-6595-5.
EISENMAN, Bonnie. Learning React Native: Building Native Mobile Apps with JavaScript. O'Reilly Media, 2015, 200 s. ISBN 978-1-4919-2900-1.
RAI, Rohit. Socket.IO Real-time Web Application Development. Packt Publishing, 2013, 140 s. ISBN 978-1-78216-078-6.
CHODOROW, Kristina. MongoDB: The Definitive Guide. Vyd 2. O'Reilly Media, 2013, 432 s. ISBN 978-1-4493-4468-9.

Vedoucí diplomové práce: **Ing. Zdeněk Šilar, Ph.D.**
Katedra informačních technologií

Datum zadání diplomové práce: **31. října 2016**

Termín odevzdání diplomové práce: **17. května 2017**

Ing. Zdeněk Němec, Ph.D.
děkan

L.S.

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2016

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.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 18. srpna 2017

Bc. Ondřej Synek

Poděkování
Chtěl bych velmi poděkovat vedoucímu mé diplomové práce Ing. Zdeňkovi Šilarovi, Ph.D. za cenné rady a čas, který mi věnoval. Velký dík patří také mé rodině za veškerou podporu.

ANOTACE

Cílem diplomové práce je návrh a implementace webové aplikace pracující v reálném čase, která představuje elektronickou podobu deskové šachové hry. V teoretické části práce bude provedena rešerše technologií použitých při vývoji aplikací podobného typu s důrazem na komunikaci klient-server. Aplikace bude využívat vzájemné komunikace dvou klientů pomocí technologie *Socket.IO* a bude dále postavena na technologiích *Node.js*, *React* a *NoSQL* databázi *MongoDB*.

KLÍČOVÁ SLOVA

3D šachy, Web socket, babylon.js, Node.js, JavaScript

ANNOTATION

The aim of this thesis is to design and implement realtime 3D web application used for playing the game Chess. The theoretical part does focus on technologies used for development of this kind of applications. Two clients communication via *Socket.IO* is used for transferring application data. The application also uses *Node.js*, *React* and *MongoDB NoSQL* database.

KEYWORDS

3D chess, Web socket, babylon.js, Node.js, JavaScript

Obsah

Seznam obrázků

Seznam zdroj. kódů

Seznam zkratek a značek

API	Application Programming Interface
HTML	HyperText Markup Language
JSON	JavaScript Object Notation
BSON	Binary JSON
FIDE	Fédération Internationale des Échecs
NURBS	Non-Uniform Rational Basis Spline
CAD	Computer Aided Design
CSG	Constructive Solid Geometry
RGB	Red Green Blue
CMYK	Cyan Magenta Yellow Key
W3C	World-Wide Web Consorsium
CSS	Cascading Style Sheets
DOM	Document Object Model
SASS	Syntactically Awesome Style Sheets
BEM	Block Element Modifier
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JS	JavaScript
ECMA	European Computer Manufacturers Association
MVC	Model View Controller
ES6	EcmaScript 6
JSX	JavaScript XML
UI	User Interface
PHP	Hypertext Preprocessor
CEST	Central European Summer Time
REST	Representational State Transfer
CRUD	Create Read Update Delete
SQL	Structured Query Language
NoSQL	Not only SQL
ACID	Atomicity Consistency Isolation Durability
IP	Internet Protocal
WS	Web Socket
WSS	Web Socket with SSL
AJAX	Asynchronous JavaScript and XML
pkg	package
DNS	Domain Name System
URL	Uniform Resource Locator

Úvod

Cílem této diplomové práce je provést návrh a implementaci webové aplikace, napsané v jazyku *JavaScript*. Tento jazyk jsem zvolil proto, že se jedná o moderní jazyk s velkým potenciálem. V poslední době stále roste počet aplikací, napsaných právě v *JavaScriptu* a rozšiřuje se komunita, která se zasluhuje o jeho další vývoj. V ekosystému *npm* vznikají každý den nové moduly, které usnadňují práci.

Za další výhodu jazyku *JavaScript* lze považovat i to, že ho lze použít jak na klientské, tak i na serverové části aplikace. V porovnání s ostatními jazyky není tak výkonově náročný. Proč tomu tak je, bude vysvětleno ve čtvrté kapitole. V neposlední řadě vidím také značné ulehčení procesu vývoje v tom, že s *JavaScriptem* lze použít stejné moduly, jak pro logiku serverovou, tak pro klientskou. V praxi to například znamená, že na *API* je použit pro práci s datumy modul *MomentJS* a v prohlížeči, na straně klientské, tento stejný modul. Stačí tedy jeden zdrojový kód a není nutné učit se dvojí dokumentaci. Pro implementaci serverové části je použita platforma *Node.js*. Na straně klienta jsem použil knihovny *React*, pro tvorbu uživatelského rozhraní a *babylon.js*, implementující 3D scénu samotné hry.

Aplikace implementovaná v rámci této práce je postavena na základech původně 2D aplikace, která byla vytvořena jako semestrální práce. Vizuálně byla primitivní a důraz byl kladen na algoritmy tahů figurek. Byl jsem příjemně překvapen, jak snadné bylo aplikaci rozšířit o 3D scénu. Během tohoto procesu jsem se utvrdil v tom, že *JavaScript*, je pro mě jako programátora správná volba. Stačilo pouze napojit již hotové modely, ovládané předtím obyčejnou 2D *HTML* stránkou, na 3D scénu. Následujícím krokem, který vedl ke zvýšení výsledné kvality aplikace, bylo nahrazení zastaralého *jQuery* moderní *JavaScriptovou* knihovnou *React*.

K dalšímu zlepšení aplikace pomohlo zavedení *web soketů*, čímž se aplikace stala aplikací pracující v reálném čase. Poslední technologickou úpravou pak byla výměna relační *MySQL* databáze za objektovou databázi *MongoDB*. Tato databáze poslouží k ukládání dat v podobě *JSON* objektů. To je výhodné, protože s *JSON* objekty se v *JavaScriptu* dobře pracuje.

1 Hra šachy

Šachy jsou deskovou hrou, která je určena pro dva hráče. Název hry je odvozen od perského slova šáh (panovník). Šachy jsou dnes také považovány za sportovní disciplínu. Při hraní hry nemá žádný vliv náhoda. Rozhodující jsou pouze zkušenosti a schopnosti hráčů. Do šachového světa také výrazně zasahují počítače. Již od začátků digitálních počítačů vytváří šachoví fanoušci šachové programy či stroje s cílem vyhrávat partie díky výpočtům. Na internetu působí silná komunita hráčů, kteří hrají na různých serverech. Existuje rozsáhlá databáze různých šachových partií, uskutečněných již od roku 1475. Lze se tedy podívat na různé historické hry, partie šachových mistrů či známých osobností.

1.1 Historie hry

Není zcela jisté kde přesně šachy vznikly, nicméně nejčastěji se původ šachů zařazuje do Guptovské říše, která se nacházela na území severní Indie. Další zmínka se datuje kolem roku 600 n. l., kdy byla v Persii hra přejmenována na šatrandž. Nejstarší nalezená šachová figurka pochází z Albánie. Byla vyrobena v 6. století ze slonoviny. Hra se dostala do Číny kolem roku 800 pod názvem siang-čchi (čínské šachy). O dvě stě let později je hra již známá i v Evropě. Pravidla hry se do jisté míry lišila a do dnešní podoby se dostala kolem roku 1475. Roku 1851 se konal v Londýně první moderní šachový turnaj [1]. Vyhrál ho Němec Adolf Anderssen, který byl prohlašován za nejlepšího šachistu tehdejší doby. V roce 1924 byla v Paříži založena Mezinárodní šachová federace FIDE a o tři roky později byl pro ženy založen titul mistryně světa. Jako první tohoto titulu dosáhla česko-anglická šachistka Věra Menčíková. Od vzniku systému mistrovství světa v šachu se jeho forma průběžně měnila, nicméně největší změny proběhly mezi lety 1993–2006. Došlo k rozpadu na dvě nezávislá a konkurenční mistrovství. Z toho důvodu celkově opadla prestiž titulu. Změny způsobily, že se k titulu může probojovat sice silný šachista, ale zdaleka ne srovnatelný s většinou mistrů dřívějších dob jako např. Kasparov nebo Steinitz.

1.2 Herní pravidla

Šachy se hrají na šachovnici a zápasí mezi sebou dva hráči - bílý a černý. Šachovnice je čtvercová deska a skládá se z šedesáti čtyř polí, které mají střídavě černou a bílou barvu. Jedná se v podstatě o matici 8x8 kde osa x je značená písmeny A až H a osa y čísly jedna až osm. Každé pole má tedy svoji souřadnici v podobě A1 až H8.

Každý hráč má k dispozici šestnáct figurek. Z toho osm pěšců, krále, dámu a dále pak dvakrát tyto figurky: věž, jezdec, střelec. Figurky jsou bílé a černé - každý hráč má figurky své barvy. Hru začíná bílý hráč tažením figurkou. Další tahy jsou prováděny střídavě.

Cílem hry je mat. Mat je takový tah, který napadá soupeřova krále, který nemůže žádným tahem z tohoto napadnutí uniknout. Hra může také skončit tzv. patem. Pat nastane ve chvíli, kdy hráč, který je na tahu nemůže žádnou figurkou táhnout. Pokud hráč svým tahem napadne soupeřova krále, jedná se o šach. Hráč, který je v šachu, musí svým tahem šach odvrátit. Dále hráč nemůže učinit takový tah, kterým by se sám do šachu dostal. Každý typ figurky má definované povolené tahy.

1.3 Popis figurek

Níže jsou popsány vlastnostni jednotlivých typů figurek, s kterými se šachy hrají.

1.3.1 Král

Král se může pohybovat v libovolném směru, pouze o jedno pole a pouze v případě, že na cílovém poli nebude vystaven šachu. Další způsob jak táhnout králem je tzv. rošáda. Jedná se o posun krále o dvě pole k jedné z věží, která se posune o pole přes krále. Rošádu lze uplatnit v případě, že se od začátku hry ani král ani daná věž nepohnuli. Také mezi věží a králem nesmí být jiná figurka. Poslední podmínkou je, že král před rošádou nebyl v šachu a nesmí přejít přes pole ohrožené soupeřem.

1.3.2 Dáma

Jedná se o nejsilnější figurku. Každý hráč má na začátku hry jednu dámu. Dáma (někdy také královna) se může pohybovat o libovolný počet polí všemi směry. Na začátku hry je umístěna vždy na poli, které barevně odpovídá figurce. Dáma bývá nejvíce účinná ve střední hře či v koncovce.

1.3.3 Věž

Věž je hned po dámě druhá nejsilnější figura a společně patří mezi tzv. těžké figury. Pohybovat se může o libovolné množství polí, ale pouze ve vodorovném či svislém směru, nikoliv šikmo. Na začátku hry má každý hráč věže dvě, obě úplně v rohu šachovnice. Jedna je blíže ke králi, druhá k dámě.

1.3.4 Střelec

Každý hráč má dva střelce a společně s koněm patří mezi lehké figury. Pohybovat se smí diagonálně o libovolné množství polí. Střelec nikdy nezmění barvu pole, jelikož pole v diagonále mají vždy stejnou barvu. Střelci jsou tedy buď bělopolní nebo černopolní.

1.3.5 Jezdec

Jezdec (někdy také kůň) se pohybuje skoky ve tvaru písmene L přičemž má na výběr buď jedno pole rovně a dvě do strany či naopak. Po každém tahu mění jezdec barvu pole. Účinnost jezdce závisí na jeho umístění. V případě, že jezdec stojí uprostřed šachovnice, kontroluje celkem osm polí. Hodnota této figurky je přibližně stejná jako střelce čí tří pěšců. Nicméně vzhledem k zvláštnostem jeho pohybu může být v některých situacích výhodnější využít právě jezdce.

1.3.6 Pěšec

Pěšec se může pohybovat pouze o jedno pole vpřed a pouze pokud je dané pole neobsazené. První pohyb každého pěšce může být o dvě pole. Dále může sebrat soupeřův kámen, který se nachází úhlopříčně na sousedícím poli před pěšcem. V případě, že se pěšec posunul z počáteční pozice o dvě pole, přičemž překročil pole ohrožené protihráčovým pěšcem, pak může být tímto pěšcem sebrán jako by šel pouze o jedno pole. Jedná se o tzv. braní mimochodem (neboli en passant). Pokud pěšec postoupí na poslední pole desky, je nahrazen na tomto poli figurkou vlastního výběru (dáma, věž, střelec nebo jezdec.

1.4 Webové statistiky

Na webu [1] je možné najít záznamy tisíce různých šachových partií. První hra, kterou si lze tah po tahu prohlédnout je z roku 1475, kdy si proti sobě zahráli Francesco di Castellvi a Narciso Vinyoles. Databáze nejen obsahuje odehrané partie, ale nabízí také různé statistiky jednotlivých tahů a je tak výborným pomocníkem pro studium různých strategií.

[1] http://www.chessgames.com

2 3D grafika

Základ 3D generovaného obrazu tvoří matematicky přesně definovaný objekt, složen z polygonů nebo z matematicky definovaných křivek. Poté jsou tomuto objektu nadefinovány fyzikální vlastnosti a to pomocí nanesení povrchu. Povrch má určenou barvu, míru pohlcování světla či jeho odrážení, průsvitnost atd. Tyto vlastnosti lze buď popsat číselně nebo texturou. Textura je matematicky generovaný obraz nebo případně fotografie z reálného světa. Barva a optické vlastnosti objektů ve scéně jsou stejně jak v reálném tak i v tom 3D světě určeny světlem. Pokud není ve scéně světlo není vidět nic. Obsahuje-li scéna světlo lze přikročit k tvz. renderování. To je matematický proces, kdy je vypočítáván výsledný obraz se všemi odlesky a stíny všech objektů, které se ve scéně nachází. Renderování je tedy převodem 3D objektů do 2D zobrazení. 3D grafika se využívá zejména v počítačovém průmyslu pro tvorbu filmů nebo her, ale má také uplatnění ve vědě a průmyslu. Je např. velice důležitá v medicíně, kde se využívá simulace 3D orgánů v lidském těle.

2.1 Historie

Ve větší míře začal výzkum 3D grafiky v šedesátých letech 20. století a to současně na mnoha místech, převážně v USA. Největší podíl na vývoji 3D grafiky má pak Univerzita v Utahu, kde roku 1968 David Evans založil projekt pro rozvoj počítačové grafiky. Projekt byl dobře financován a přitáhl přední experty v oboru. Právě tady došlo k objevům, které umožnily, že se dnešní 3D grafika nachází tam kde je. Právě na univerzitě v Utahu vytvořil Martin Newell slavnou konvici, která dodnes slouží jako jakýsi základní 3D model stejně jako foto Leny Söderberg ve 2D grafice. Později došlo k založení několika grafických firem právě výzkumníky, kteří působili na univerzitě. Jedná se např. o Adobe Systems (John Warnock) nebo Pixar (Edwin Catmull). Roku 1976 spatřil světlo světa film Futureworld, ve kterém bylo možné vidět první 3D snímky generované počítačem. Prvním celovečerním 3D animovaným filmem byl v roce 1995 Příběh hraček z dílny Pixaru ve spolupráci s Walt Disney. Na tomto filmu se podílel i z velké části Steve Jobs, který tak pomohl položit základní kámen 3D filmovému průmyslu [2].

2.2 Modelování

Pro orientaci v rovině nebo prostoru je používána kartézská soustava souřadnic. Jedná se o pravoúhlý a pravotočivý souřadný systém. Z počátku vede kladný směr osy X, kolmo a vlevo na něj je osa Y a kolmo na rovinu definovanou osami X a Y vede osa Z . Souřadnice lze zadávat dvěma způsoby a to buď v kartézské a nebo polární soustavě souřadnic. Kartézská je pravoúhlá a osy se protínají v počátku souřadnic. Jednotky ve všech osách jsou stejné. Naproti tomu v polární soustavě jsou souřadnice definovány nejkratší vzdáleností od počátku souřadnic a úhlem této

spojnice od základní osy. Definice objektu v polární soustavě souřadnic má význam pouze ve 2D prostoru [3]. Převod trojrozměrného objektu do dvojrozměrné reprezentace se nazývá promítání. Při promítání je ztracena prostorová informace a může tedy dojít ke zkreslení představy skutečného tvaru objektu. Z toho důvodu se využívá různých promítání pro různé účely. Existují dva základní druhy rovinné projekce – rovnoběžné (paralelní) a perspektivní. V rovnoběžné projekci je střed promítání nevlastní a promítací přímky jsou určeny směrem promítání. Je zachována relativní velikost modelu. Naopak v případě, že je střed promítání vlastní a promítací přímky procházejí tímto středem promítání, jedná se o perspektivní promítání. Respektuje se zde optický model, který vyjadřuje lidské vnímání reálného světa. Je modelována proporcionální změna předmětů při vzrůstající vzdálenosti od pozorovatele.

2.2.1 Bod

Základním objektem v souřadném systému je bod. Je definovaný souřadnicemi. V pravoúhlém souřadném systému jsou souřadnice průmětem vzdálenosti na jednotlivé osy. Souřadnice bodu jsou zapsané v pořadí [X, Y, Z].

2.2.2 Křivky

Základní křivkou je přímka. Přímka je spojnice dvou bodů v prostoru. Dále existují interpolační křivky. To jsou křivky, které prochází všemi zadanými body. Oproti tomu aproximační křivky všemi zadanými body neprochází, ale řídí se řídícími body. Počet těchto bodů je n+1, přičemž n je stupeň dané křivky. Spojnice mezi těmito body nazýváme řídicím polygonem.

Bézierova křivka

Bézierova křivka je parametrická křivka, která se často využívá pro modelování ve dvou rozměrech ale i pro definici trojrozměrných objektů. Např. se také používají i při definici fontů. Křivka je zadána n řídícími body, přičemž prochází jen prvním a koncovým. Ostatní body tvoří lomenou čáru, která modeluje tvar křivky. V praxi se používá převážně zadání čtyřmi body.

NURBS křivka

NURBS se používá v počítačové grafice pro vytváření křivek a ploch, které nabízejí přesnost a flexibilitu při práci s tvary. Křivka NURBS je definována kontrolními body s různou váhou a uzlovými vektory. Křivku můžeme prezentovat ve dvourozměrné nebo třírozměrné kartézské soustavě souřadnic. Aplikováním různých operací na kontrolní body lze snadno dosáhnout transformací např. rotace. NURBS křivky jsou výhodné z výpočetního hlediska. Funkce jsou vyhodnoceny rychle stabilními a přesnými algoritmy. Dále nejsou náročné na paměť v porovnání s jinými metodami [4].

2.2.3 Plocha

Plocha je trojrozměrně definovaná, má čtyři řídící křivky. Z toho mají dvě sousedící společný vrchol. Dále má plocha směry U a V , definované řídicími křivkami. Tyto směry jsou obdobou os X a Y souřadného systému. Plocha má v těchto směrech určitý stupeň, opět definovaný řídicí křivkou. Řídicí body tvoří řídicí polygon. Změnou umístění kteréhokoliv řídicího bodu se mění celá plocha.

2.2.4 Těleso

Základní geometrická tělesa jsou složena z jednoduchých ploch. Nejobvyklejší reprezentací tvaru těles bývá hraniční reprezentace, kdy jsou tělesa popsány jako hranicemi určené mnohostěny. Hranicemi se rozumí buď stěny, hrany a nebo vrcholy. V CAD systémech se využívá metody CSG (konstruktivní geometrie pevných těles). Modely jsou modelovány z primitivních geometrických těles operacemi sjednocení, průniku a rozdílu. Pro zobrazování se poté model převede do hraniční reprezentace. Dále existuje reprezentace objemová. Zde jsou tělesa definována jako množina bodových vzorků. Vzorky jsou sbírány např. lékařským tomografem nebo 3D scannerem. Pro zobrazování je pak používána metoda sledování paprsku, případně se opět tělesa převádějí do reprezentace hraniční. Výčet základních 3D těles

- koule
- krychle
- anuloid
- kvádr
- válec
- kužel
- komolý kužel
- jehlan
- elipsoid

2.2.5 Voxel

Ve 2D grafice se používá výraz pixel, který vyjadřuje nejmenší možnou jednotku rastrové grafiky. Představuje jeden bod, který je nadefinován svou barvou, např. ve formátu RGB nebo CMYK. Oproti tomu voxel je v podstatě pixel v prostoru a označuje částici objemu. Název vznikl spojením anglických slov volumetric a element (objemový prvek). Voxely se využívají při vizualizaci, 3D modelování případně analýze lékařských nebo vědeckých dat.

2.3 Vizualizace

Ve chvíli kdy je těleso namodelováno, je dalším krokem zajistit, aby mělo v 3D scéně realistický vzhled. Pouhé definování tvaru nestačí, a proto je zapotřebí scénu obohatit o několik dalších faktorů, které budou popsány níže.

2.3.1 Texturování

Aby objekt vypadal realisticky, je potažen texturou. Textura je bitmapa, ve které může být každý pixel jinak barevný. Texturou tedy lze na objektu zobrazit komplexní povrch, kterým může být např. dřevo, lidská kůže atd. Textura je 2D objekt a aby byl správně nanesen na 3D objekt, používá se mapování textur. Textury mají různé kanály

- Diffuse - obrazová informace RGB společně s alfou (průhlednost), při rovnoměrném nasvícení povrchu.
- Bump (normálový kanál) – kanály RGB značí XYZ hodnotu normálového vektoru v tečném prostoru.
- Specular (odraz) – kanály RGB označují barvu a intenzitu odlesku pixelu po nasvícení.
- Refraction (lom světla) – používá se pro definici lámání světla.

Při aplikování textur na objekty je důležité, aby odpovídala jejich pozice. K tomuto účelu slouží nástroje mapování textur. 3D prostor je popsán souřadnicemi X, Y a Z a povrch objektu souřadnicemi U , V , W. V případě plošného modelu má každá plocha své souřadnice UVW a jejich orientace může být vůči sousední ploše jiná Oproti tomu polygonový model je rozvinutelný do jednoho UVW prostoru a poté je možné nanést příslušné textury.

2.3.2 Světlo

Světlo je jakýmsi pomyslným srdcem každé scény. Pokud by scéna nebyla osvětlena, pak by nebyl žádný objekt viditelný. Přidáním osvětlení do scény okamžitě vzroste míra realističnosti jakou scéna vyzařuje. Tím, že se od objektů odráží světlo, působí důvěryhodněji, případně mohou být až k nerozeznání od reálného objektu.

Typy světel

Pro dobrý vzhled scény je nutné vybrat správný typ světla, případně nakombinovat typů více. Na různá místa v různých scénách se hodí různé typy světel s různými nastaveními.

Typy světel

- Ambientní - toto světlo je charakteristické tím, že jeho intenzita je stejná ze všech směrů k objektu. Ambientním světlem lze rozumět v přenesení do reálného světa, světlo, které vzniká odrazy od předmětů např. v místnosti.

- Bodové - v případě bodového světla je světlo, jak už napovídá název, vyzařováno z jednoho bodu do všech směrů. Se zvětšující vzdáleností od zdroje se může zeslabovat jeho intenzita.
- Směrové - světlo, které je definované směrem odkud vyzařuje a intenzitou, která je pro celou scénu stejná. To znamená, že je to světlo, které je natolik vzdálené, že ovlivňuje celou scénu stejně. V reálném světě by se mohlo jednat o slunce.
- Kuželové - světlo je vyzařováno z jednoho bodu směrem definovaném kuželem. Od středu kužele ke kraji dochází k úbytku intenzity. Z reálného světa lze přirovnat např. ke stolní lampě.
- Plošné - světlo se šíří z obdélníkového pole světel. Může jít např. o simulaci světla vnikajícího oknem do místnosti.

2.3.3 Stínování

Pokud se ve 3D scéně nachází světlo pak je položen základ pro realistické zobrazení objektů nacházejících se ve scéně. Nicméně to nejdůležitější, co nejvíce určuje jak realisticky objekt vypadá je stínování. Stínování definuje jak se vykreslí určité místo na povrchu tělesa aby vynikla jeho iluze trojrozměrnosti.

Typy stínování

- Konstantní - jedná se o nejjednodušší a zároveň nejméně výpočtově náročnou metodu stínování. Převážně se hodí pro rovinné plochy nebo pro potřeby rychlého stínování. Algoritmus vychází z toho, že každá plocha má pouze jednu normálu. Na základě normály se vypočítá barva, která se přiřadí všem pixelům dané plochy. Nevýhodou je, že výsledný objekt nevypadá realisticky právě z důvodu, že každá plocha má stejnou barvu.
- Gouraudovo - tato metoda zajišťuje plynulé stínování křivých povrchů tak, že aproximace povrchů ploškami není zřetelná. Nicméně interpolace samotného odstínu barvy nezvyšuje jas na plošce nahrazující oblinu, která je kolmá na dopadající světelný paprsek. Nelze tedy vytvářet realistické odlesky způsobené odraženým světlem. Přesto tato metoda alespoň zahlazuje barevné rozdíly u místních nerovností povrchu.
- Phongovo - v případě tohoto stínování je algoritmus podobný jako u Gouraudova stínování. Obohacen je o výpočet normály ve vnitřních bodech plochy a pomocí nich je ze světelného modelu určen odstín barvy každého pixelu. To má za následek, že na rozdíl od Gouraudova stínování jsou na plochách zřetelné barevné odlesky světel. Tento způsob tedy zajišťuje hladké vystínování ploch s korektně vykreslenými odlesky.

2.3.4 Renderování

Renderováním se rozumí tvorba reálného obrazu na základě počítačového modelu. Může jít o obrázek, video nebo také vykreslování scény v reálném čase. Konečný vzhled scény lze ovlivnit pomocí různých nastavení v závislosti na použitém softwaru. Cílem renderování scén je vytvářet obrazy napodobující reálný svět. Realistické počítačové animace jsou využívány např. při tvorbě filmových efektů, v architektuře nebo různých simulací ve vědě. Renderování je také

důležitou disciplínou, které se věnují herní vývojáři. Objekty nacházející se ve scéně jsou popsány v přesně definovaném jazyce nebo struktuře. Tato struktura obsahuje popis scény jako informace o stínování, o světlech, textury a geometrii. Tato data jsou dále zpracována renderovacím programem ke spočítání výsledku do digitální podoby. K renderování se využívá různých renderovacích algoritmů. Níže jsou popsány nejdůležitější z nich.

Ray casting

Tato metoda je základním renderovacím algoritmem pro zobrazování 3D počítačové grafiky. Využívá geometrického algoritmu sledování paprsku. Renderovací algoritmy, které jsou založené na sledování paprsku, umožňují transformaci virtuální trojrozměrné scény do dvourozměrného obrázku. Geometrie paprsků je sledována do zdroje světla z oka pozorovatele. Tato metoda je rychlá a jednoduchá a to z toho důvodu, že dochází k výpočtu pouze barvy světla bez opakovaného počítání dalších paprsků, které se odráží do stejného místa od jiných ploch. Proto není přesné zobrazení odrazů, průhledností nebo přirozeného úbytku stínu. Tato renderovací metoda byla využívána převážně v počátcích 3D her.

Ray tracing

V překladu metoda sledování paprsků je metodou globálního osvětlení. V reálném světě se paprsky pohybují od zdroje, odráží se a lámou a nakonec dorazí až do oka pozorovatele. V případě ray tracingu paprsky naopak vycházejí z kamery. To z toho důvodu, že ze zdrojů světla vychází nekonečně mnoho paprsků a je nemožné rychle spočítat, které dopadnou na pixely plátna. Dále se metoda používá k simulaci optických systémů. Výhodou metody je její univerzálnost. Dá se použít i k simulaci systémů, jejichž popis je pouze přibližný nebo není možný. Např. prvky pro rentgenovou optiku. Nevýhodou je vysoká výpočetní náročnost [4].

3 Klientské webové technologie

V této kapitole budou popsány technologie, které se využívají při implementaci klientské části webových aplikací.

3.1 HTML

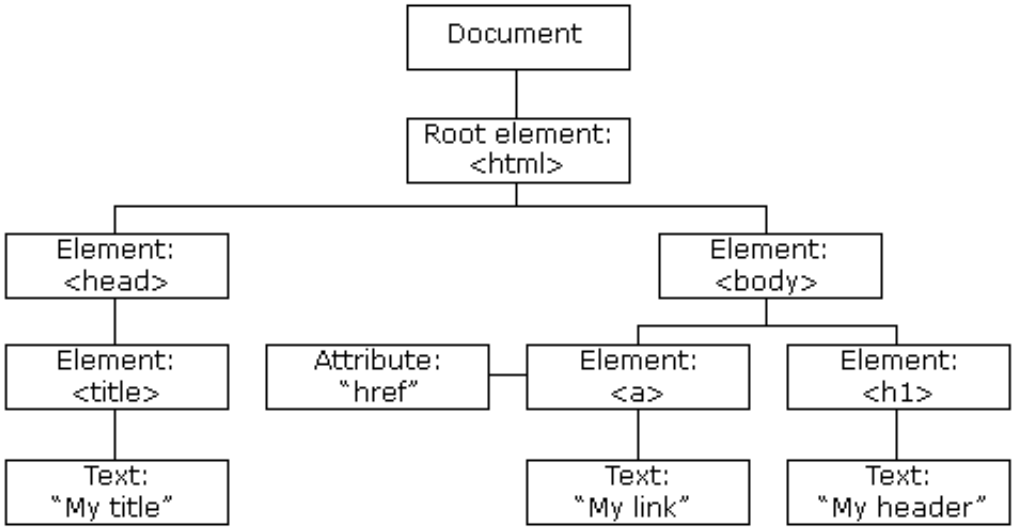
HTML je již celkem dlouho standardem pro tvorbu webu. Vytvořen byl již v roce 1991 Timem Berners-Leem. Projekt převzalo konsorcium W3C a dále se věnuje jeho vývoji. Nyní je aktuální *HTML*5. *HTML* je značkovací jazyk. Je složen z jednotlivých značek, které definují strukturu dokumentu. Každá značka má své specifické chování. Např pokud vložíme do dokumentu několik značek typu `div`, tyto elementy se vykreslí pod sebou. Naopak v případě značky typu `span` se vykreslí vedle sebe. Toto chování lze libovolně měnit pomocí *CSS* stylů. V aplikaci, které se věnuje tato diplomové práce, je jazyk *HTML* použit jako vstupní bod do reactové aplikace. Při requestu na URL aplikace vrátí *Node.js* server *HTML* stránku, která importuje skript - build reactové aplikace.

3.1.1 DOM

DOM (Document Object Model) je abstrakcí strukturovaného textu, v případě webu - *HTML* kódu. Elementy kódu se stávají uzly a *DOM* je reprezentace *HTML* kódu v paměti. Je to stejný princip jako u procesu, který je instancí nějakého programu. Stejně jako může stejný program běžet ve více procesech tak může mít *HTML* několik *DOMů* (např. stejný *HTML* obsah v několika tabech) *HTML DOM* nabízí rozhraní pro práci s jednotlivými uzly. Obsahuje metody jako `getElementById` nebo `removeChild`. Vždy když se má změnit obsah webové stránky, je nutné změnit *DOM*. Na obr. 3.1 je možno vidět stromovou strukturu *DOMu*.

3.2 SASS

První *CSS* specifikace spatřila světlo světa v roce 1996, když byla zveřejněna konsorciem W3C. Slouží ke stylování *HTML* dokumentů. *HTML* říká, jaké elementy dokument obsahuje a také jaké je jejich pořadí. Ale samo o sobě nic nevypovídá o tom, jak má výsledná stránka vlastně vypadat. Právě proto jsou tu styly. Ty mají za úkol každému elementu nadefinovat jeho vlastnosti, které mají dopad na jeho grafickou reprezentaci. V případě textu to může být, barva, velikost, typ fontu, efekty jako podtržení, kurzíva atd. U ostatních objektů pak např. odsazení (vnější - margin, vnitřní - padding). s *CSS3* přišli animace a *HTML* elementy je tak možné i rozpohybovat.



Obrázek 3.1: Stromová struktura DOMu [1]

Nicméně po čase se psaní samotných *CSS* stylů stalo těžkopádné. Z jednoduchých webových stránek se začaly rozrůstat obsáhle webové aplikace a s příchodem různě velkých monitorů a zařízení s menšími obrazovkami (tablety, mobily) se objevila nutnost začít psát styly responzivně. To znamená použitelné zobrazení na všech možných rozlišeních. Zdrojové kódy *CSS* stylů tak začaly bobtnat a brzy se staly nepřehlednými. Proto se vyvinuly preprocesory, které všechny fungují na stejném principu. To co se napíše čistě a přehledně, se následně překompiluje do klasické *CSS* podoby. Pro potřeby aplikace byl vybrán preprocesor *SASS* a v další kapitole budou vysvětleny jeho nejdůležitější výhody. Ale nejprve bude představen model BEM, aby mohla být demonstrace *SASSu* kompletní.

3.2.1 BEM a objektové CSS

BEM (Block, Element, Modifier) je metodikou pro pojmenovávání tříd v objektově orientovaném *CSS*. Celá myšlenka objektového *CSS* je založena na tom, aby se styly psaly bez duplicit a aby bylo možné různé části přepoužít pro další komponenty. Představme si modelovou situaci - webová stránka s pravým a levým postranním oknem. Obě mají shodné rysy (barvu, margin), ale jsou mezi nimi rozdíly (velikost textu, padding). Je nesmysl, to co mají společné, definovat u obou explicitně. V případě, že by takových elementů, které mají shodné vlastnosti bylo víc, by bylo nutné vše ručně přepsat ve chvíli, kdy by se design stránky změnil a místo modré barvy by měla být použita zelená. Proto je důležité držet stejné styly na jednom místě a na dalších se na ně pouze odkazovat. To lze jednoduše uskutečnit pomocí *SASS* funkcí. Nicméně stále je potřeba myslet na to, aby byly styly co nejpřehlednější. Pokud se *SASS* funkce využívají bez rozmyslu může výsledný kód být ještě hůře čitelný než klasický *CSS*. V této fázi na pomoc přichází BEM.

[1] https://www.w3schools.com/js/js_htmldom.asp

Zdroj. kód 3.1: Ukázka HTML struktury s BEM metodikou

```
<div class='login'>
  <button class='login__buttonlogin__button--signin'>
    SignIn
  </button>
  <button class='login__buttonlogin__button--signup'>
    SignUp
  </button>
</div>
```

Blok

Celá stránka je pomyslně rozložena do jednotlivých bloků. Pod blokem si lze představit jeden či více elementů, které je možné přesunout jinam do stránky aniž by ztratili smysl. To může být například *login* formulář. Je jedno je-li velký, uprostřed stránky a nebo malý v pravém horním rohu. Stále plní svoji funkci. Nicméně třeba samotný input pro uživatelské heslo, které je součástí loginu, nikam mimo *login* přesunout nelze, jelikož samo o sobě, bez ostatních částí loginu význam nemá. A jako takové prezentuje druhou část BEM metodiky a to element.

Element

Element je tedy něco co musí být uvnitř bloku, protože bez něj nemá smysl. Element se zapisuje jako název bloku, dvě podtržítka a název elementu (*login-form__button*).

Modifikátor

Poslední skupinou jsou modifikátory. Ty mají za úkol popsat, že element kromě toho, že má své základní definované vlastnosti, je nějakým způsobem upraven. Přičemž elementy stejného typu mouhou mít na sobě různé modifikátory. Např. blok přihlašovací formulář obsahuje dvě tlačítka (elementy) - příhlásit a zaregistrovat. Oba elementy jsou tlačítkem formuláře a mají shodné některé styly - velikost, odsazení, text... Každý má ale jinou barvu aby bylo na první pohled jasné, že představují odlišnou funkcionalitu. Modifikátory se zapisují jako element, který má na konci ještě dvě pomlčky a název modifikátoru (login-form__button–signin / login-form__button–signup). Modifikátory by se neměly pojmenovávat podle barvy, nýbrž podle funkce, kterou představují. To pro případ, že by se v budoucnu změnila grafika. Popsáním modifikátoru podle funkčnosti místo barvy se vyhneme případnému přejmenování. Na první pohled může BEM notace vypadat poněkud zmatečně, nicméně je na první pohled jasně poznat při čtení *CSS* stylů a *HTML* kódu jaká je struktura. Navíc pomocí *SASSu* lze BEM psát velmi jednoduše a přehledně díky využití zanořování. Ukázka použití BEM v *HTML* je na zdroj. kód 3.1, zápis BEM pomocí *SASS* pak na zdroj. kód 3.2

3.2.2 Klíčové vlastnosti SASS

Níže jsou popsány nejdůležitější vlastnosti, kterých lze využít při stylování preprocesorem *SASS*.

Zdroj. kód 3.2: Ukázka zápisu BEM pomocí SASS preprocesoru

```
.login {
  padding: 20px;

  &__button {
    padding: 10px;
    color: white;

    &--signin {
      background-color: red;
    }
    &--signup {
      background-color: blue;
    }
  }
}
```

Zdroj. kód 3.3: Ukázka použití proměnných v SASS

```
$primary-font: Helvetica, sans-serif;
$primary-color: #CDCDCD;

body {
  font: 100% $primary-font;
  color: $primary-color;
}
```

Zanořování

Jednou z klíčových vlastností preprocesoru *SASS* je možnost libovolného zanořování. Na rozdíl od *HTML*, kde je jasně určená struktura dokumentu, v *CSS* žádná struktura není. Každý styl je dán pouze selektorem, který popisuje danou *HTML* strukturu. Nicméně *SASS* právě umožňuje zanořování ve struktuře úplně stejně jako *HTML*. Ve stylu selektoru lze tedy mít další selektor, který obsahuje definici stylů a libovolný počet dalších selektorů [5].

Proměnné

SASS umožňuje práci s proměnnými. Lze si tedy uložit hodnoty, které jsou využity na více částech kódu (např. barvy, velikosti odsazení atd.). Odpadá tedy nutnost, při změně nějaké takové hodnoty, přepisování všech jejich výskytů, tak jak je to nutné dělat v *CSS*. Ukázka viz. zdroj. kód 3.3

Zdroj. kód 3.4: Ukázka použití mixin v SASS

```
@mixin border-radius($radius) {
  -webkit-border-radius: $radius;
  -moz-border-radius: $radius;
  -ms-border-radius: $radius;
  border-radius: $radius;
}

.radius-element {
  @include border-radius(10px);
}
```

Modularizace

Aby byl kód co nejčistší využívá se rozdělování do více souborů, které jsou následně importovány. Je tedy možné vytvářet různé moduly, které lze dále libovolně přepoužívat. Import je sice podporován i v *CSS*, nicméně pro každý import je nutné vytvářet HTTP request což není velmi efektivní. V případě SASSu vzniká stále pouze jeden výsledný překompilovaný soubor.

Mixiny

Mixiny se hodí pro případy, že nějaký styl má různé prefixy pro různé prohlížeče. V samotném selektoru je pak zavolán mixin s nadefinovaným parametrem a v překompilovaném výsledném souboru se objeví všechny styly obsažené v mixině. Ukázka viz. zdroj. kód 3.4

Dědičnost

Pomocí dědičnosti, lze nadefinovat rodičovské prvky (většinou tzv. placeholdery - píší se s obsahují obecné styly, které podědí všechny selektory, které na rodiče využijí extend funkci. Využití dědičnosti lze demonstrovat na výše prezentovaném příkladu zanoření a využití BEM metodiky. Výhodou oproti předchozímu řešení je fakt, že tlačítko je definované jako abstraktní element (placeholder - sám o sobě není nikde použit) a je přepoužitelný i mimo *login* tedy kdekoliv v celé aplikaci. Ukázka viz. zdroj. kód 3.5

Operátory

Při stylování lze použít matematických operátorů. Není tedy nutné počítat výslednou hodnotu ale pouze ji matematicky zapsat. Ve výsledném *CSS* už je použita vypočítaná hodnota. Ukázka viz. zdroj. kód 3.6

Zdroj. kód 3.5: Ukázka použití dědičnosti v SASS

```
%button {
  padding: 10px;
  color: white;
}

.login {
  padding: 20px;

  &__button--signin {
    @extend %button;
    background-color: red;
  }

  &__button--signup {
    @extend %button;
    background-color: blue;
  }
}
```

Zdroj. kód 3.6: Ukázka použití operátorů v SASS

```
$width: 1000px;
$z-index: 10;

.wrapper {
  width: $width/2;
  z-index: $z-index+10;

  .content {
    width: $width/4;
    z-index: $z-index+15;
  }
}
```

Zdroj. kód 3.7: Ukázka použití příkazů v SASS

```
$type: debug;

header {
  @if $type == debug {
    color: blue;
    @for $i from 1 through 3 {
      .debug-info#{$i} {
        display: block;
      }
    }
  } @else {
    color: black;
  }
}
```

Zdroj. kód 3.8: Ukázka použití funkce v SASS

```
@function calc-percent($target, $container) {
  @return ($target / $container) * 100%;
}

$percentage: calc-percent(100px, 200px); // 50%
```

Příkazy

V SASSu lze použít některé základní příkazy známé i z ostatních jazyků. Na základě podmínky nějakému stylu přiřadit hodnotu, případně projet cyklem několik selektorů, které jsou očíslovány. Toho lze například využít pro debugování, kdy styly obsahují debugovací větve, které se při nasazení do produkce ignorují. Ukázka viz. zdroj. kód 3.7

Funkce

Při použití preprocesoru *SASS* lze definovat a volat funkce. Stejně jako v jiných jazycích do funkce je možné poslat jeden či více parametrů a získat z funkce výsledek. Ukázka viz. zdroj. kód 3.8

3.3 JavaScript

JavaScript (dále *JS*) spatřil světlo světa v roce 1995, kdy byl představen společností Netcape. Navzdory, že je v názvu obsaženo slovo Java, s tímto jazykem nemá *JS* nic společného. Maximálně mají tyto jazyky podobnou syntaxi. Roku 1997 byl jazyk asociací ECMA standardizován pod názvem ECMASript. [6] Poté se dostal *JS* do webových prohlížečů a je tedy umožněno spouštět kód na straně klienta. Díky tomu mohlo dojít ke zlepšení uživatelské atraktivity webových stránek.

V současné době je *JS* stále jediným jazykem, umožňujícím v prohlížeči skriptování. Vytvořit dynamické *HTML* stránky je tak možné buď přímo v *JS* nebo jiném jazyku, který je nutné do *JS* zkompilovat. Na základě ECMA standardizace je *JS* verzován s tím, že každý egine podporuje rozdílné části specifikace. To je důvod, proč není možné vždy použít nejnovější funkce. Proto je nutné používat tzv. transpiléry. Transpiléry umožňují transpilaci do nižší, všemi podporované, verze *JS*. Lze tak tedy psát nově specifikované funkce a využívat nejnovější podporovanou syntaxi a kód je následně převeden do plně kompatibilního *JS*.

Vzhledem k tomu, že je možné ve webovém prohlížeči využít jenom *JS*, je k dispozici velké množství různých knihoven a frameworků. Nejvíce známá je pravděpodobně knihovna *jQuery* na které bylo postavené velké množství aplikací. Nicméně pokrok jde v před a *jQuery* nyní využívá zastaralý přístup, který postupně ztrácí v konkurenci moderních technologií na atraktivitě. V *jQuery* chybí koncept pro vývoj komplexnějších aplikací (např MVC). V další kapitole bude hlouběji vysvětlena jedna z moderních technologií - *React*. Níže budou představeny základní vlastnosti JS s občasnými odkazy na novou verzi *JS* - ECMAScript 6 (dále jen ES6). Vlastnosti nové verze ES6 budou poté dále dopodrobna rozebrány.

3.3.1 Metody a funkce

V programovacích jazycích je běžná praxe rozlišovat slova metoda a funkce. V *JS* mají uplatnění obě slova a mají jiný význam. Výraz funkce označuje definici jež není odpovědná žádné třídě či objektu. Podle konvence se zapisuje camelCase. Naopak funkce, která náleží nějakému objektu či třídě je nazývána metodou. Z *JS* pojmů je pak dále zajímavý Konstruktor neboli také konstrukční funkce. Je používána pro vytvoření proměnných všech typů. Při zápisu se používá PascalCase. Další pojem je Třída a má stejný význam jako konstrukční funkce. V *JS* nejsou klasické třídy jako v dalších jazycích a mají svá specifika. Třídám bude věnována větší pozornost v dalších kapitolách.

3.3.2 Reference

Mezi klasické *JS* vlastnosti patří práce s referencemi. Reference je ukazatel na místo, kde se aktuálně objekt nachází. Pracujeme-li tedy s referencí, pracujeme přímo s objektem na který ukazuje. V případě, že několik proměnných ukazuje na ten samý objekt tak úpravou kteréhokoliv z nich dojde ke změně všech.

3.3.3 Typy

V *JS* neexistují klasické typy proměnných jako v typových jazycích. Alespoň ne přímo. Když se vytváří proměnná tak pomocí klíčového slova var (ES6 umožňuje ještě `const` a `let`). Takové proměnné lze přiřadit libovolnou hodnotu jakéhokoliv typu, který *JS* obsahuje.

Seznam typů v *JS*

- `Object`
- `Array`

- `Function`
- `String`
- `Number`
- `Boolean`

3.3.4 Kontext

Užívání proměnných se řídí jistými pravidly a jedním z nich je jejich působnost v kódu (tzv. *scope*). Ten je závislý na kontextu ve kterém je scope udržován.

Druhy kontextu

- globální
- lokální
- blokový (pouze ES6)

Globální kontext existuje pouze jeden a v prohlížeči je to objekt *window*. Globální proměnná je proměnná, jejíž definice se nachází na samostatném místě v kódu, kde nenáleží žádnému lokálnímu kontextu. Nesmí tak být uvnitř těla funkce. Pokud se globální proměnná nadefinuje v bloku `<script>` přímo v *HTML* dokumentu (nebo je tento kód, přímo do *HTML* naimportován), je tato proměnná dostupná napříč celou aplikací a to např. i ve zdrojových souborech *Reactu*. Pokud je globální proměnná definována v nějakém samostatném souboru, který není do *HTML* přímo importován, ale jedná se například o předpis třídy, pak je proměnná dostupná z celého souboru, kde se nachází ale už ne v celé aplikaci. Lokální kontext pak vzniká vždy v rámci funkce s tím, že je možné do sebe funkce zanořovat. V případě takového zanoření vzniká tzv. *scope chain*. Pokud se tedy definice proměnné nachází v těle funkce jedná se o lokální proměnnou. Taková proměnná je přístupná pouze v rámci kontextu, tedy např. pouze v těle dané funkce, kde se nachází její definice. Pokud se v těle funkce nachází další funkce, ve které je nadefinována proměnná se stejným názvem, dojde k překrytí vnější proměnné tou vnitřní - dojde tím k výše popsanému *scope chainu*.

3.3.5 Callback

JS je jazyk asynchronní, to znamená, že se různé části kódu mohou provádět současně i když jsou napsané pod sebou. Příkladem může být volání *API*. Pokud je například potřeba poslat dva různé requesty na *API* a na základe jejich odpovědi složit request třetí, nestačí pouze kód napsat pod sebe a očekávat, že se vše provede jak má. Je nutné počítat s tím, že první dva requesty se budou vykonávat současně a teprve poté co přijdou ze serveru odpovědi pro oba tyto requesty, lze poslat request třetí. Pro práci s asynchronním kódem slouží tzv. *callbacky*. Princip je takový, že se *callback*, což je reference na funkci, předá do parametru volané funkce, kde se vykonává asynchronní kód. Kód se vykoná a poté se tento *callback* zavolá, přičemž se začne vykonávat kód ve funkci odkud byl *callback* předán a byla volána asynchronní funkce. *Callback* může být v některých případech velice užitečný a jeho výhodou je jednoduchost. Nicméně zmíněný příklad, kde je potřeba zareagovat na dokončení dvou funkcí běžících paralelně, není pro využití *callbacku* úplně vhodný. Celkově se nedoporučuje používat *callbacky* tam, kde je funkcí více

a kvůli *callbackům* dochází k zanořování. Kód se poté stává nepřehledný. Tento stav bývá označován jako *„callback hell“* tedy *callbackové peklo*. Proto se doporučuje používat spíše *promisy*, které budou vysvětleny v další podkapitole.

3.3.6 Promise

Promise vyjadřuje budoucí hodnotu asynchronní operace. Operace může buď uspět (*resolve*) nebo selhat (*reject*). Pokud je např. volána asynchronní funkce, např. request na *API* pak přijde response, která je buď úspěšná a obsahuje data, která měla být získána a nebo request z nějakého důvodu selže (např. nevalidní autentifikace, případně timeout) a response vrátí pouze error. Ať už request uspěje nebo selže, je nutné na tento výsledek zareagovat. To *promise* umožňuje velice elegantně a to tak, že za volání funkce se přidávají dva handlery a to právě pro úspěch - *then*, či v případě neúspěchu - *fail*. Do těchto handlerů se píše kód, který se provolá poté co se promise dokončí. Uvnitř volané funkce je pak nutné na základě výsledku zavolat jeden z parametrů, které jsou do funkce implementující *promise* předány. Jedná se o parametr *resolve*, který se provolá v případě, že je funkce úspěšná a na základě provolání dojde právě ke skoku do *then* větve. Naopak *reject* slouží pro případ, že vykonání funkce není úspěšné a jeho provolání vyústí ve vykonání *fail* větve. Pokud je volán *promise*, tak v těle může být buď zmíněný mechanismus nebo lze vracet další volání jiné *promisy*. *Promisy* lze tedy neomezeně řetězit (*chainovat*) na sebe. *Promise* také umožňuje výše zmíněné vykonání několika asynchronních funkcí najednou a další vykonání kódu se provolá až ve chvíli kdy jsou všechny dané funkce hotové.

3.3.7 ES6

Nová verze *JavaScriptu* ES6 přináší velké množství novinek, které zpříjemňují a zlehčují práci a produkují čistší kód. První novinkou je obohacení definice proměnných pomocí var o dvě nová klíčová slova - `const` a `let`. V případě `const` se deklaruje proměnná, které se okamžitě přiřazuje hodnota s tím, že už nelze změnit. Jedná se tedy o klasickou konstantu známou i v jiných jazycích. Klíčové slovo `let` pak definuje tzv. blokovou proměnnou. Tato proměnná je platná pouze v rámci bloku ve kterém je definována. Další nová věc, kterou ES6 do *JS* přináší, je možnost nastavit výchozí hodnotu parametrům funkce. Pokud se tedy tato funkce zavolá bez parametrů, všechny parametry, které mají přiřazenou výchozí hodnotu se na ni nastaví. Dále je nově k dispozici tzv. spread operátor, který slouží ke spojení polí. Použití je jednoduché, stačí v definici nového pole, za poslední prvek přidat tři tečky a referenci na pole, které má být připojeno. Výsledné pole pak obsahuje definované elementy a všechny elementy z přidaného pole. Další vychytávkou je zkrácený zápis při předávání parametrů do objektu. V klasickém *JS* bylo nutné při předání parametru do objektu vždy zapsat *{ promenna: promenna }*. V ES6 pokud se má jmenovat klíč v objektu stejně jako se jmenuje předávaná hodnota, stačí jednoduše zapsat pouze *{ promenna }* a hodnota se předá automaticky [7].

Arrow funkce

Velké změny přináší možnost užití arrow funkce (=>). S arrow funkcí lze psát zjednodušený zápis funkcí. Příklad lze vidět na zdroj. kód 3.9

Zdroj. kód 3.9: Porovnání klasického javascriptu a ES6

```
// plain JS
function loginToLobby(name) {
    console.log('Hello', name);
}

setTimeout(function () {
    console.log('Loaded');
}, 2000);

players.forEach(function (player) {
    console.log(player);
});

function getColor() {
    return this.color;
}

// ES6
loginToLobby = name => console.log('Hello', name)

setTimeout(() => console.log('Loaded'), 2000)

players.forEach(player => console.log(player))

getColor = () => (
    this.color
)
```

Zdroj. kód 3.10: Použití importu v ES6

```
import {
    BrowserRouter as Router,
    Route,
    Redirect
} from 'react-router-dom'
```

Zdroj. kód 3.11: Použití funkce find v ES6

```
// plain JS
const players = [
    {color: 'black', name: 'John'},
    {color: 'white', name: 'Joe'}
]

function findBlackPlayer(name) {
    for (let i = 0; i < players.length; ++i) {
        if (players[i].color === 'black' && players[i].name === name) {
            return players[i];
        }
    }
}

// ES6
player = players.find(player => player.color ==='black' && player.name
   === 'John');
console.log(player); // {color: 'black', name: 'John'}
```

Import

Dále je možné využít zjednodušeného zápisu při importování modulů, u kterých se importuje objekt a je nutné importovat několik částí tohoto objektu. V klasickém *JS* se používal na import modulů require, ES6 přináší klíčové slovo import. Ukázkový import může vypadat viz. zdroj. kód 3.10

Find

Díky ES6 je nyní jednodušší práce s poli, pokud je nutné vyhledat nějaký objekt. Dříve bylo nutné celé pole projet cyklem. S ES6 je možné využít funkci `find`. Ukázka kódu viz. zdroj. kód 3.11

Babel

ES6 přináší spoustu výhod, nicméně zatím není podporován ve všech prohlížečích. Proto je nutné kód napsaný v ES6 transpilovat do starší verze *JS*, která je všemi prohlížeči podporována.

Nejvíce používaným transpilerem je *Babel*. *Babel* zvládá překlad ES6 kódu a navíc dokáže také zpracovat JSX tedy *Reactové* soubory. Modul *Babel* se dříve jmenoval 6to5 a dnes ho používá řada známých firem jako Facebook, Mozilla, Yahoo atd. *Babel* lze také nasadit do *Node.js* aplikace a lze tak tedy plně využívat ES6 i na serveru (*Node.js* sice ES6 již z velké části podporuje ale stále ne plně).

3.4 React

React byl vyvinut společností Facebook, která ho vyslala do světa jako open source v květnu 2013. Nyní má repositář na githubu přes osm a půl tisíc commitů a tisíc přispěvatelů. Je to knihovna, která má na starost vytváření uživatelského rozhraní. Prezentuje pouze V (View) vrstvu z modelu MVC. Soustředí se tak pouze na jednu věc a ta je v tomto případě maximálně propracovaná. Tím se liší od dalších knihoven jako např. Angular a Ember. Bez použití *Reactu* (např. *jQuery*, či obyčejný javascript) se postupuje tak, že se vykreslí výchozí stav aplikace a každou interakcí uživatele se aplikace nějakým způsobem mění. Tedy píše se kód, který má za cíl vyvolávat změny. S *Reactem* je to přesně obráceně, tedy píše se kód, popisující jakým způsobem má vypadat výsledek. Rozdíl je v tom, že není přistupováno k *DOMu* [8].

3.4.1 Komponenta

Reactová stránka se skládá z jednotlivých komponent. Každá komponenta implementuje metodu render, která popisuje pomocí jakých *DOMů* se komponenta vykreslí. Existují dva způsoby jak do komponenty uložit data. Buď je lze uchovat ve *State*, přičemž po každé změně *State* (metodou `setState`) se volá render aby se změny vykreslily. Nebo lze komponentě předávat data pomocí props. Props jsou v podstatě parametry dané komponenty nastavované v rodiči. Props jsou v rámci komponenty pouze pro čtení. Měnit se dají pouze v rodiči. Princip předávání props lze vidět na obrázku 3.12.

3.4.2 Životní cyklus komponenty

Komponenta prochází několika fázemi životního cyklu. Porozumění vlastnostem těchto fází je nutné pro správnou práci s uloženými daty. Pokud se například nevhodně zavolá `setState`, může dojít k zacyklení aplikace a to v případě, kdy se `setState` volá ve fázi, která se provádí vždy po přerenderování. Dojde tady k situaci, že po vykreslení se opět volá vykreslení a tak stále dokola. Metody fází životního cyklu se rozdělují na mounting (inicializační - jsou volány při vytváření komponenty), updating (volané při změně komponenty) a unmounting (mazání komponenty).

constructor

Jako první v celém životním cyklu se volá konstruktor komponenty. Zde je nutné volat *super(props)* aby bylo možné přistupovat k props a dále se zde inicializuje *State* pomocí přiřazení *this.state = objekt definující state*. Konstruktor je jediné místo kde lze nastavovat *State* pomocí

přiřazení. Na ostatních místech je nutné použít metodu `setState`. Toto je přístup využívaný při psaní *Reactu* syntaxí ES6. Při psaní starší ES5 se *State* neinicializuje v konstruktoru, ale je k tomu určená metoda `getInitialState` a obdobně `getDefaultProps` pro inicializaci props.

getInitialState

Tato metoda slouží v případě, kdy je psán *React* pomocí ES5, k nastavení výchozích hodnot ve statu, které jsou pak přístupné přes *this.state*.

getDefaultProps

Stejně jako metoda výše slouží k nastavení výchozích hodnot ale v tomto případě propsu. Tyto hodnoty jsou poté přístupné přes *this.props*.

componentWillMount

Jak název napovídá tato metoda se volá ještě před tím, než se komponenta vytvoří a tím pádem před renderem. Proto lze v této metodě bezpečně používat `setState` aniž by došlo k zacyklení.

render

Metodu `render` musí mít každá *Reactová* komponenta. Metoda popisuje jak se daná komponenta vykreslí - jaká je *DOM* struktura a jaká jsou data. Render se volá vždy poté co je změněn stav komponenty metodou `setState`.

componentDidMount

Po vykonání metody render je dále na řadě `componentDidMount`. Zde už jsou k dispozici DOMy a lze s nimi manipulovat. Touto metodou končí cyklus inicializační fáze.

shouldComponentUpdate

Jedná se o první metodu aktualizační části životního cyklu a volá se před renderem. Lze zde definovat zda je nutné, aby se komponenta překreslila či nikoliv. Stejně jako u všech ostatních metod v aktualizační části životního cyklu je v parametrech předán nový *State* a nové propsy. Právě na základě porovnání původního a nového stavu dat komponenty lze rozhodnout o překreslení. Metoda není volána při inicializačním vykreslení.

Zdroj. kód 3.12: Ukázka použití props v Reactu

```
class Lobby extends Component {
    render() {
        return (
            <Text>Hello {this.props.name}!</Text>
        );
    }
}

class Page extends Component {
    render() {
        return <Lobby name='Chessmaster' />
    }
}
```

componentWillUpdate

Metoda je volána poté co `shouldComponentUpdate` vrátí true a stále ještě před renderem. Stejně jako u metody `shouldComponentUpdate` je v parametrech předán nový *State* a nové propsy. V metodě nelze měnit *State* z důvodu zacyklení.

componentDidUpdate

Tato metoda je zavolána po renderu a v parametru se nachází předchozí *State* a předchozí propsy. Je možné zde provádět operace s *DOMy*.

componentWillReceiveProps

Další metoda, která stojí za zmínku je `componentWillReceiveProps` a slouží pro přijetí nových props.

componentWillUnmount

Tato metoda je z poslední fáze a je volána před odebráním komponenty. Slouží k vyčištění kódu.

3.4.3 Flux

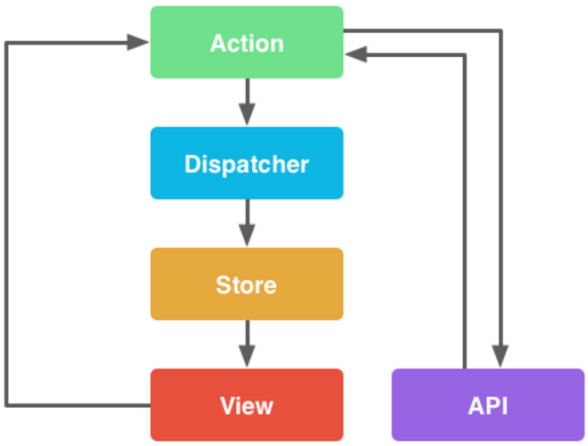
Jak již bylo zmíněno výše, *React* sám o sobě prezentuje pouze View vrstvu. Zbývající části má na starosti už konkrétně zvolená architektura. Flux architektura aplikuje návrh jednosměrného toku dat a pro práci s *Reactem* ho využívá např. Facebook. Výsledné View je reprezentovatelné pomocí dat, které se volají z úložiště. Data v úložišti se mění podle toho, jaké volá uživatel svými interakcemi akce. Schéma struktury Flux je naznačeno na obr. 3.2.

Zdroj. kód 3.13: Ukázka akce v architektuře Flux

```
$.ajax({
    method: "GET"
    //...
}).done(players => {

    dispatcher.dispatch({
        type: "RECEIVE_PLAYERS", players: players
    });

});
```



Obrázek 3.2: Schéma Flux architektury [2]

Action

Nejprve před sestavením komponenty je potřeba získat data. Proto se jako první věc zašle request na *API*. Data, která přijdou v odpovědi se dále předají *Dispatcheru* a to pomocí akce. Akce obsahuje svůj unikátní název a samotná data. Podle názvu (či typu) je akce odchycena ve *Store* . Při použití v aplikaci by se posílaly např. akce informující o změně seznamu připojených hráčů. Ukázka viz. zdroj. kód 3.13.

Store

Store slouží pro uchování dat. Aplikace jich může mít několik, každý pro jednu dílčí část aplikace. V případě šachů by např. *lobby Store* uchovával seznam hráčů. *Store* má implementovány *gettery* a *settery*. Ve chvíli, kdy pomocí *Dispatcheru* obdrží data tak si je přes *setter* nastaví a v aplikaci je pak možné tyto data získat zavoláním *getteru*. Pro odchytávání dat slouží funkce

[2] https://scotch.io/tutorials/getting-to-know-flux-the-react-js-architecture

Zdroj. kód 3.14: Ukázka storu v architektuře Flux

```
class DeviceStore extends EventEmitter {
    constructor() {
        super();
        this.players = [];
    }

    setPlayers(players) {
        this.players = players;
    }

    getPlayers() {
        return this.players;
    }

    handleActions(action) {
        switch (action.type) {
            case "RECEIVE_PLAYERS":
                {
                    if (action.players) {
                        this.players = action.players;
                        this.emit("changePlayers");
                    }
                    break;
                }
        }
    }
}
```

`handleActions` ve které se nachází jako parametr objekt akce. Podle typu akce se pak nastaví příslušná data. *Store* je poděděn od třídy EventEmitter a to z toho důvodu aby mohl využívat emit. Emit slouží k informování komponent, které naslouchají na změny daného *Store*, že je nějaká změna v datech. Vždy když se data na základě nějaké akce změní tak se změní ve *Store* a komponentám je odeslána informace o změně. Tyto komponenty si pak zavolají *getter* a dostanou aktuální data. Ukázka viz. zdroj. kód 3.14.

Dispatcher

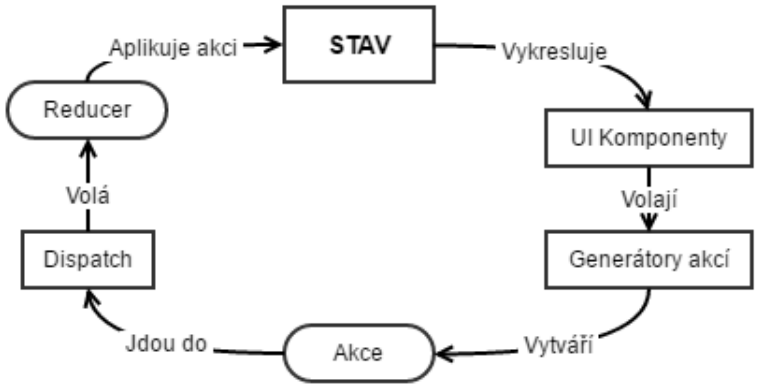
Dispatcher je v aplikaci implementován jako Singleton. Existuje tedy jediná jeho instance. Jeho úkolem je vždy když nastane nějaká akce, přeposlat data do všech *Store*, které naslouchají na danou akci. To funguje tak, že *Store* si na *Dispatcher* zaregistrují callback, který je provolán ve chvíli, kdy nastane událost. *Dispatcher* umožňuje řídit, kterému *Store* se předají data dříve. To z důvodu, že by v aplikaci existovaly dva *Store* naslouchající na stejnou akci a bylo by nutné aby se nejdříve dostaly do prvního a až poté do druhého. K tomuto účelu lze využít funkci `waitFor`.

View

View je prezentováno *React* komponentami. Ty vykreslují definovanou strukturu a data, která získávají ze *Store*. Aby komponenta dostávala pravidelně upozornění na změnu dat ve *Store* je nutné se každý *Store* zaregistrovat jako posluchač. To se většinou realizuje ve funkci `componentWillMount`, kde se na třídě *Store* volá funkce on a v parametru se posílá název funkce, kterou *Store* emituje a handler, který na změnu dat reaguje. Tento handler přijme nová data a uloží je do *State* komponenty pomocí funkce `setState`. Tím, že se zavolá `setState` se komponenta přerenderuje již s novými daty.

3.4.4 Redux

Alternativou k Fluxu je knihovna Redux, která je vhodná pro velké projekty díky dobré udržovatelnosti kódu. Je postavena na myšlence udržovat související kód na několika málo místech. Při potřebě změny je pak zřejmé, kde je nutné změnu udělat. Tím se zamezí chybám. Redux má za úkol spravovat stav aplikace a na rozdíl od Fluxu udržuje data na jednom jediném místě. To řeší problém ke kterému může ve Fluxu dojít, kdy je více instancí *Store*, které uchovávají duplicitní data. Pokud se mají stejná data ukázat na více místech ale jsou prezentována jinými komponentami, přičemž každá má svůj *Store* a tím pádem vlastní data, může se stát, že budou zobrazena rozdílná data. Tento stav se dá přirovnat k člověku, který má hodinky jak na pravé tak i na levé ruce. Pokud budou oboje ukazovat jiný čas, tak kolik je vlastně hodin? Redux má všechna data v jednom velkém *Store* a všechny komponenty tak mají jediný „zdroj pravdy“ v podobě stavu aplikace. Tento stav je navíc možné ukládat a opětovně z něj složit vzhled stránky přesně tak jak vypadala v době uložení stavu. Dále lze ze sledování podoby stavu určit co uživatel v aplikaci dělá. Na jaká kliká tlačítka, jaké volá interakce. Díky využití stavu je možné jednoduše implementovat vracení zpět a přesun vpřed. Architektura knihovny Redux je znázorněna na obrázku 3.3. UI komponenty se vykreslují na základě stavu aplikace a při každé změně tohoto stavu se překreslí. Uživatel svou interakcí s komponentami vytváří akce, které jsou metodou `dispatch` předány do *Store*. Dále se pomocí *Reduceru* aplikují na stav, který opět slouží pro vykreslení komponent.



Obrázek 3.3: Redux - tok informací [3]

[3] http://vsadnajavu.cz/2016-08/odborne/javascript/jak-redux-definuje-tolik-potrebna-omezeni-na-velkem-projektu

Stav

Stav (*State*) je popsán tak, že se k výchozí podobě stránky připočtou veškeré změny, které v aplikaci nastaly až do doby, kdy stav posuzujeme. Stav v sobě zahrnuje nejen javascriptové proměnné, ale také *DOM* elementy, jejich atributy, obsah atd. Díky stavu je mimo jiné možné použít renderování na serveru jelikož je ze stavu zřejmé jak má přesně stránka vypadat. Tím pádem pokud je na serveru k dispozici stav, je možné na klienta odeslat již zkompletovanou stránku. Ukázka stavu, který znázorňuje seznam hráčů v *lobby* je možné vidět na obrázku 3.4.

Obrázek 3.4: Redux - ukázka stavu

Akce

Akce v reduxu mají stejný význam jako v architektuře flux. Parametry v akci jsou stejné, tedy typ akce, což je řetězcový klíč, a samotná data. K zavolání akce může dojít např. kliknutím tlačítka v komponentě. Nicméně akce můžou také vzniknout nezávisle na jakékoliv komponentě. Jako příklad může posloužit logika Authentifikace, využita v praktické části práce. Při reloadu dojde ke ztrátě stavu aplikace, a tedy i přihlašovacích údajů. Proto jsou tyto údaje uloženy v cookie a v Authentifikační třídě se v metodě, která vrací zda je uživatel přihlášen, daná cookie načte. Pokud jsou tedy uloženy přihlašovací údaje je potřeba je dostat do stavu aplikace. To zařídí zmiňované zavolání akce, které není vázáno na komponentu. Takové volání akce je zařízeno metodou `dispatch`, která se volá na *Store*, jež je singleton a jeho instance je dostupna napříč celou aplikací. Dále se pak také pomocí akce dostanou do stavu data, která přichází přes soket. Tedy např. nově příchozí hráč do *lobby*. Ukázka jak vypadá akce, příchozího hráče je znázorněna na zdroj. kód 3.15.

Reducer

Ve chvíli kdy nastane nějaká akce, je jasné, že nastane nějaká změna ale zatím není jasné jak ovlivní aplikaci. Úkolem *Reduceru* je na tuto změnu zareagovat a zajistit, že se daná změna promítne ve stavu aplikace. V reduxu je využit pouze jeden *Reducer* a jeden *Store*. Nicméně v aplikaci se objevuje několik různých komponent, které mají svá různá data a logiku. Proto není

Zdroj. kód 3.15: Ukázka akce v architektuře Redux

```
export const playerJoined = (player) => {
    return dispatch => {
        dispatch({type: ActionTypes.playerJoined, player})
    }
}
```

dobré spravovat vše v jednom souboru, protože by se kód stal nepřehledným. Proto je využívaná funkce `combineReducers`, která má za úkol z více *Reducerů* vytvořit jeden společný.

Komponenta

Reduxová komponenta funguje stejně jako komponenta fluxová s jedním rozdílem. Komponenta v architektuře redux nepřistupuje k datům ve stavu přímo, ale používá se mapování ze stavu do props. V tomto mapování je nutné nadefinovat všechny parametry, které bude props obsahovat a přiřadit k nim hodnotu ze stavu aplikace.

3.4.5 React Native

React Native je framework, vyvinutý Facebookem, pro vývoj nativních mobilních aplikací. Výhodou implementace aplikací v React Native je, že stačí jeden kód a kompilace pak probíhá zvlášť do platforem Android a iOS. Tato kompilace je postavena na následujícím principu. Programátor v kódu pracuje s Reactovými komponentami, které vzhledově a funkcionalitou odpovídají komponentám nativních platforem. Při kompilaci se pak Reactové komponenty mapují na nativní komponenty obou platforem [9].

Webová aplikace, která je napsána v Reactu, lze tedy relativně jednoduše přepsat do React Native a následně přeložit do nativních mobilních aplikací Android či iOS. Jediné co je potřeba změnit je View vrstva. Místo HTML komponent použít React Native komponenty. Veškerá logika, implementována buď architekturou Flux či Redux, zůstává.

3.5 Webpack

Pro snazší práci s *Reactem* a *SASSem* je výhodné použít knihovnu *Webpack*. Jedná se o tzv. bundlovací nástroj a je určen pro práci s javascriptovými moduly a vytváření balíčků pro prohlížeč. Dále umožňuje práci s různými assety. Assetem se rozumí *SASS*/*CSS* styly, obrázky, fonty atd. Zkrátka soubory, které je nutné načíst v prohlížeči. *Webpack* má za cíl tyto assety zpracovat a vytvořit balíček (modul), který se jednoduše nahraje na webserver. *Webpack* modul je cokoliv, co lze v kódu naimportovat. Může to být např. `@import` v *SASS* kódu. Ten *Webpack* neumí nativně ale pomocí rozšíření. Assety, které nejsou javascript je nutné na javascript převést nebo vyextrahovat do souboru. K tomuto účelu slouží loadery a pluginy, které *Webpack*u Na tuto práci používá *Webpack* loadery a pluginy, které mu umožňují nad moduly provádět preprocessing před tím, než je výstup uložen do filesystému [10].

Webpack dovoluje importovat např. tyto soubory:

- *jsx, coffee*
- *css, SASS, less, stylus, postcss*
- *png, jpeg, svg*
- *json, yaml, xml a další*

3.5.1 Vývoj vs produkce

S *Webpackem* lze pracovat ve dvou módech - vývojovém (develop) a produkčním (prod). Při vývoji lze využít buď *Webpack* s parametrem watch, kdy automaticky při každé změně dojde ke kompilaci tzv. bundle. Bundle je soubor obsahující zkompilovaný kód všech zdrojových souborů. Bunde je importován do *HTML* souboru, který slouží jako vstupní bod do aplikace. Alternativou k parametru watch je vývojový server - *Webpack dev server*. Myšlenka je taková, že se vytvoří jednoduchý server, obalující bundle soubor. Dev server umí tzv. hot reload, což je automatické přenačtení obsahu v prohlížeči při změně zdrojových souborů. Ve vývojovém režimu je možné debugování pomocí tzv. source map. neboli map zkompilovaného kódu do původního zdrojového kódu ve kterém lze krokovat. Oproti vývojovému režimu, produkční slouží k finálnímu nasazení na web. Využívá se minifikace, díky které dochází k dramatickému snížení velikosti bundlu. Minifikace redukuje kód různými metodami, např. vytknutí částí kódu, odstranění mezer atd. Výhodou toho, že z několika původních souborů (*JS*, *SASS*, obrázky atd.) vznikne jeden zkompilovaný bundle je ten, že ke stažení všeho stačí pouze jeden request. To ušetří čas, protože v případě více requestů na více souborů je spotřebován čas na samotnou režii requestů.

4 Serverové webové technologie

Tato kapitola se věnuje technologiím, využívaným při implementaci serverové části webových aplikací.

4.1 Node.js

Node.js vznikl v roce 2009 a je tak relativně mladý např. oproti PHP, které spatřilo světlo světa již v roce 1995 a je tedy téměř o 15 let starší. Na poli informačních technologií je to opravdu dlouhá doba a již několikrát se ukázalo, že za tuto dobu může být všechno jinak. Na internetu se vedou diskuse zda PHP ztrácí na atraktivitě či nikoliv každopádně na rozdíl od PHP, které už si za ta léta vybudovalo stálou komunitu, *Node.js* a potažmo celkově *JavaScript* komunita značně sílí každým okamžikem. Každý den přibývají na repozitáři npm nové moduly, rodí se různé návody či dotazy. V roce 2012 tedy pouhé 3 roky od vydání první verze byl *Node.js* webem InfoWorld, který se zabývá internetovými technologiemi oceněn jako nejlepší technologie roku.

Node.js slouží pro implementaci serverové části aplikace v jazyku *JavaScript* a byl vybrán pro to, že je díky němu možné implementovat jak klientskou část tak serverou část ve stejném jazyce. To je obrovská výhoda oproti ostatním serverovým technologiím v tom, že jak na klientské tak na serverové části lze používat stejné moduly. Např. pro formátování data je použit modul Moment.JS, který umožňuje ze standardního textového vstupu (např. Sun May 07 2017 15:50:59 GMT+0200 (CEST)) vytvořit objekt, který má k dispozici užitečné funkce jako výpis data v jiném formátu nebo přičítání/odečítání dní či měsíců, zkrátka vše co se může při práci s datem hodit. A jelikož se s daty pracuje jak na serveru tak na klientovi je nutné tento modul na obou místech importovat. Na klientské části - v prohlížeči se používá pouze *JavaScript* a proto v případě implementace aplikace v *Node.js* je možné použít pouze Moment.JS. U aplikací kde se serverová část implementuje pomocí jiného než *JavaScriptového* jazyka (*.NET, Java, PHP...*) je nutné použít na serveru modul, který je napsán ve stejném jazyku na kterém server běží a zároveň *JavaScriptový* na klientovi. Stejný problém je tedy třeba řešit dvěma odlišnými způsoby a je vyžadováno naučit se používat oba moduly.

Node.js je založen na enginu V8, který vyvinul *Google*. Nad *JavaScriptovým* interpretem je vrstva *C++* kódu. Je navržený pro vysoce škálovatelné aplikace zejména webové servery. Využívá model řízený událostmi a asynchronní I /O operace. To zaručuje maximální výkon [11].

4.1.1 Event loop

Node.js je postaven na myšlence pracovat efektivně. Typická věc, bez které se žádná aplikace neobejde je přístup do databáze. A to může být zdlouhavý úkon. Pokud je databáze špatně navržena a přeplněna daty může každý dotaz trvat klidně i jednotky až desítky sekund. Proto by

Zdroj. kód 4.1: Zacyklení v Node.js

```
var continueInTheLoop = true

// one second timeout
setTimeout(() => {
    console.log('this␣code␣will␣never␣be␣called')
continueInTheLoop = false;
}, 1000);

while (continueInTheLoop) {
    // this blocking code will freeze application
}

console.log('this␣will␣never␣be␣called␣either');
```

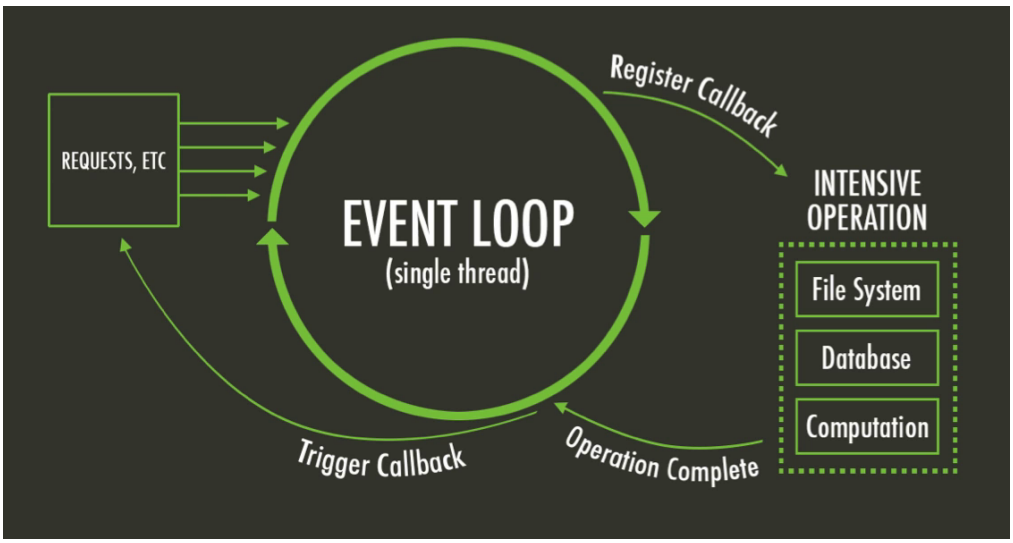
bylo velmi neefektivní poslat požadavek na databázi a dokud nepřijde odpověď tak pouze čekat. *Node.js* funguje tak, že místo čekání přidá informaci, co se má po dokončení operace provést dál. K tomu je použit tzv. callback. *Node.js* tedy pošle dotaz na databázi, zpracovává další operace a po dokončení databázové operace vykoná její callback, což může být například přeposlání dat z databáze klientovi. Přesto, že se nečeká na odpověď ze souborového systému, případně databáze, nemůže nikdy nastat, že by jeden proces zpracovával dvě instrukce paralelně. To je velká výhoda, protože není třeba řešit konkurenční přístup. Postupné zpracování instrukcí nicméně přináší i jistá úskalí. Aplikace může být zablokována náročnější prací s daty.

Na příkladu kódu viz. zdroj. kód 4.1, lze vidět blok kódu, který by se měl vykonat po jedné sekundě od spuštění skriptu. Takto napsaný skript ale zamrzne. Nicméně z *Node.js* aplikace lze spouštět další procesy a také je možné mezi nimi komunikovat. Každý *Node.js* proces běží vždy v jednom vlákně a je tedy možné kontrolovat jeho vytížení procesoru. Každý proces pak může mít svůj specifický účel. Myšlenka *Node.js* tkví v tom, aby se co nejvíce vytížilo vlastní vlákno a zároveň byl poskytnut dostatek prostoru pro ostatní vlákna.

Základním kamenem *Node.js* je tzv. „Event loop“ - smyčka událostí. Smyčka stále prochází frontu a vykonává dané události a jejich callbacky. Jelikož se pracuje pouze s jedním vláknem, odpadá režie obsluhy vláken. Stačí méně operační paměti (v případě více vláken vyžaduje paměť každé vlákno) a ušetří se na nepotřebném přepínání mezi vlákny, které by mnohdy mohlo být náročnější než vykonání nějakého nenáročného úkolu. Princip event loopu je naznačen na obr. 4.1

4.1.2 Express

Express je jedním z nejpoužívanějších *Node.js* frameworků určený pro tvorbu webových aplikací a *API*. Použití Expressu se neřídí žádnými striktními pravidly. Jeho úkolem je vyřešit několik nejdůležitějších věcí bez kterých se základní aplikace neobejde a programátorovi dává v podstatě volnou ruku v tom jak bude aplikaci stavět. Není nijak pevně určeno jakou by aplikace měla mít strukturu. Tu si může zvolit programátor tak aby se mu s ní dobře pracovalo. První z věcí, kterou Express implementuje jsou handlery HTTP requestů, které jsou na aplikaci



Obrázek 4.1: Princip event loopu [1]

zasílány. Těmto handlerům se říká routy a jsou spravovány v jednotlivých routerech, což jsou soubory obsahující routy oddělné podle logiky k jaké se requesty vážou. Requesty jsou odchytávány na základě url na kterou jsou posílány a HTTP metody requestu. Co se týče metod tak lze buď zpracovat request na danou url pro všechny HTTP metody definováním routy all nebo definovat zvlášť jednotlivé metody GET, POST, PUT, DELETE [12].

4.2 REST API

V posledních letech stále roste zájem o to, jak co nejlépe propojit mezi sebou různé webové aplikace či servery. Z tohoto důvodu se ustálil výraz *API*. Jedná se o rozhraní, které je veřejně dostupné na nějaké URL adrese a slouží pro komunikaci s aplikací. Může jít např. o server, který poskytuje nějaké informace. Na toto *API* se lze připojit pomocí aplikace, která z tohoto *API* dané informace stahuje. Zkratka REST potom představuje *API*, s kterým lze komunikovat pomocí HTTP requestů, přičemž na každou URL lze použít všechny HTTP metody. Vždy když chce klient komunikovat s *API*, tak pošle request a čeká na odpověď (response) [13].

4.2.1 Metoda GET

Tato metoda slouží k získání dat. Nastavení dat, která se mají vrátit, je možné prostřednictím tzv. query. Query jsou parametry, které se zapisují na konec URL za znak otazníku, ve tvaru ?parametr=hodnota. Je-li potřeba poslat parametrů více, oddělují se znakem &. Typický příklad pro praktickou část aplikace může být získání seznamu online hráčů v *lobby*.

[1] https://www.sencha.com/blog/asynchronous-javascript-promises

4.2.2 Metoda POST

POST metoda slouží k posílání dat na *API*. Posílaná data se nachází v těle requestu. Většinou se na základě POST requestu na *API* uloží nový záznam do databáze. V případě *lobby* by byl POST request použit pro přihlášení nového hráče.

4.2.3 Metoda PUT

Metoda PUT je určena pro zaslání změněných dat. Data se opět nachází v těle requestu. Většinou mají data stejnou strukturu jako pří prvním zaslání metodou POST, ale liší se hodnoty. V případě *lobby* by se jednalo o request, který by odpovídal změně jména hráče v *lobby*.

4.2.4 Metoda DELETE

Jak už z názvu vyplývá tato metoda slouží ke smazání záznamu. Většinou se v URL requestu zašle ID záznamu, který má být smazán. Pro příklad s *lobby*, by DELETE request odpovídal odhlášení hráče z *lobby*.

4.3 Socket API

Pokud pro potřeby *API* nedostačuje technologie REST, pak lze využít technologii soketů. Výhoda soket *API* je v tom, že se na daný endpoint stačí pouze jednou připojit a v případě, že jsou k dispozici nová data, dojde automaticky k jejich přijetí. To s REST *API* není možné, tam si veškerou komunikaci řídí klient, *API* pouze odpovídá. Proto je socket *API* vhodné pro aplikace, které pracují v reálném čase. Na druhou stranu využití socket *API* má své nevýhody. Poslání zprávy na server pomocí soketu je pouze jednosměrné. Klient tedy zašle zprávu a už nedostane žádnou odpověď, jak jeho zprávu server přijal. To lze vyřešit posláním nové zprávy ze serveru na klienta, nicméně to není úplně nejčistší způsob jelikož zpráva musí mít nějaké označení.

4.4 Kombinace REST a socket API

Vzhledem k výhodám a nevýhodám obou *API*, může být nejlepší cesta tyto *API* zkombinovat. Pro requesty kde je nutné obdržet informaci o tom, zda byla zpráva úspěšně zpracována, se může použít REST *API*. Typickým příkladem z praktické části, kdy je vhodné využít REST *API* a nikoliv soket, je přihlášení do *lobby*. Tím, že se pošle klasický request, lze jednoduše a programově čistě zareagovat na odpověď. Pokud přihlášení proběhlo úspěšně, lze pokračovat. V opačném případě, kdy např. uživatel se stejným jménem je již v *lobby* přihlášen, je přijata v response chybová hláška a v UI je zobrazena. Další podstatná věc je bezpečnost. Klient si může ručně složit jakoukoliv zprávu, kterou může zaslat na *API*. Např. pokud je hráč na tahu, může poslat fingovaný tah figurkou, který vůbec není možný. Taková zpráva o tahu je jednoduchý JSON obsahující souřadnice výchozího a cílového pole tažené figurky. Aby tedy hráč nemohl zahrát

neplatný tah je třeba na serveru zkontrolovat zda je tah validní a teprve poté informovat soupeře. Pro provedení tahu je tedy opět lepší REST *API*, protože před odesláním odpovědi může být daný tah zkontrolován a následně je klient informován zda proběhl v pořádku. Závěrem může být tedy tvrzení, že pro CRUD operace, které případně potřebují na serveru kontrolu, se hodí REST *API* a naopak pro jednoduché zprávy, které je potřeba doručit rychle, socket *API*.

4.5 Databáze MongoDB

MongoDB je implementací objektové *NoSQL* databáze. Název mongo vychází z anglického slova humongous (obrovský). Slovo *NoSQL* neznamená „bez *SQL*“, jak by se mohlo na první pohled zdát, ale not-only *SQL*, tedy nejen *SQL*. To v podstatě znamená, že se nevyužívá klasického *SQL* jazyka jako v případě relačních databází, ale jiného přístupu, který nicméně splňuje stejné požadavky.

Princip objektové databáze je postaven na tom, že místo tabulek se data ukládají do objektů. Tyto objekty jsou v terminologii *MongoDB* nazývány dokumenty. Dokumenty se nachází v jednotlivých kolekcích, ze kterých se celá databáze skládá. Dokument je uložen ve formátu BSON, což je formát určen pro binární ukládání serializovaných dat, strukturovaných jako JSON. BSON oproti JSONu obsahuje také datové typy uložených dat. Databáze *MongoDB* nemá definované databázové schéma. To znamená, že struktura dat není před samotným uložením dat známá a každý dokument uložený v kolekci může mít strukturu zcela odlišnou.

Výchozí jazyk pro práci s databází je *JavaScript*, nicméně jsou k dispozici ovladače, umožňující využívat databázi v různých jazycích. Ovladač má pak za úkol typy z formátu BSON překonvertovat do typů daného jazyka [14].

4.5.1 Databázová transakce

Databázová transakce představuje zpracování souboru příkazů, po jejichž dokončení dojde k převodu databáze z jednoho konzistentního stavu do druhého. Konzistentím stavem se rozumí takový stav, jež splňuje všechna integritní omezení definovaná databázovým schématem. Ve spojitosti s transakcí se používá výraz ACID, který definuje vlastnosti transakce, které musí být dodrženy, aby byl stav databáze stále konzistentní. Konzistenci lze vysvětlit na příkladu s bankovnímy účty. Ve chvíli, kdy má dojít k převodu peněz z jednoho účtu na jiný účet, skládá se transakce ze dvou operací. Nejprve je nutné částku z původního účtu odečíst a poté na cílový účet přičíst. Konzistence je zachována v případě, že jsou úspěšně vykonány oba kroky. Nelze se dostat do stavu, kdy jsou peníze odečteny ale z důvodu nějaké chyby se již nepřičtou k cílovému účtu. Níže jsou vysvětleny jednotlivé požadavky na konzistenci.

ACID:

- A - Atomicity (atomicita)
- C - Consistency (konzistence)
- I - Isolation (nezávislost)
- D - Durability (trvanlivost)

Atomicita

Transakce musí proběhnout úspěšně celá nebo se nesmí provést ani z části. Selže-li její část, pak musí selhat jako celek.

Konzistence

Po provedení transakce je databáze vždy v konzistentním stavu - zapsána jsou vždy pouze platná data.

Nezávislost

Částečné změny, které se provedou před dokončením transakce nejsou viditelné jiným transakcím. Transakce tedy mohou probíhat současně a vzájemně se neovlivňují

Trvanlivost

Úspěšně zapsané změny jsou již v databázi uloženy natrvalo.

V případě *MongoDB* je to s transakcemi komplikovanější. *MongoDB* sice splňuje ACID požadavky ale pouze na úrovni dokumentu. Lze se tedy spolehnout, že při úpravě dokumentu se buď úprava provede nebo ne a výsledkem jsou konzistentní data. Nicméně *MongoDB* transakce jako takové, známé z relačních databází, nepodporuje. *MongoDB* neumí zajistit konzistenci při práci s více dokumenty na jednou. Příklad transakce s bakovním převodem, zmíněný výše, by pak musel proběhnout v rámci jednoho dokumentu. Z hlediska trvanlivosti má *MongoDB* další zvláštnost. V případě relačních databází je trvanlivost zajištěna tím, že každá zapisovací operace je zapsána do tzv. žurnálu. Tím je bezpečně uložena a nelze o ni přijít např. následkem výpadku proudu. *MongoDB* pro změnu zapisuje do žurnálu v nastavených časových intervalech, nezávisle na zapisovacích operacích. Je tedy možné, i když celkem nepravděpodobné, že by data ukládaná po posledním zápisu do žurnálu mohla být ztracena. Tento problém lze nicméně vyřešit tzv. škálováním, tedy rozdělením odpovědnosti na více serverů. [2]

4.5.2 Objektová databáze vs relační databáze

Databáze *MongoDB* se primárně zaměřuje na výkon a dostupnost dat. Zajištění konzistence je pak složitější a s tím je nutno při výběru databáze počítat. *MongoDB* je vhodné využít ve specifických případech. S jistotou lze říci, že nenajde uplatnění vždy a všude. Při analýze aplikace nelze ihned na začátku říci, zda je rozumné použít objektovou, relační či kombinaci obou databází. Toto rozhodnutí musí být důkladně promyšleno na základě požadavků aplikace a je nutné zvážit pro a proti různých přístupů.

[2] https://dzone.com/articles/how-acid-mongodb

Zdroj. kód 4.2: Ukázka vložení dokumentu do kolekce

```
db.players.insertOne(
    {
        color: "white",
        name: "John Doe"
    }
)
```

Zdroj. kód 4.3: Ukázka získání dokumentu z kolekce

```
// Returns all white players
db.players.find(
    {
        color: "white"
    }
)
```

4.5.3 Ukázka příkazů

Pro práci s daty jsou nadefinované *JavaScriptové* příkazy, které se volají na jednotlivých objektech, které databáze obsahuje. Nejčastěji se jedná přímo o referenci databáze samotné, kolekce databáze a jejich dokumentech.

Insert

Vkládání dokumentů se provádí např. příkazem insertOne, který je volán na kolekci a vloží právě jeden dokument. Ukázka použití insertu viz. zdroj. kód 4.2.

Find

Získání dat z databáze se docílí zavoláním funkce `find` na kolekci, ze které jsou data získána. Buď se nadefinuje podmínka, která filtruje vrácená data, případně bez podmínky je vrácena celá kolekce. Ukázka použití funkce find viz. zdroj. kód 4.3.

5 Web Socket technologie

Jelikož vyvíjená aplikace je hrou dvou hráčů v reálném čase, je nezbytné aby veškerá interakce uživatele s aplikací byla co nejrychlejší. V případě, že by se po každém kliknutí a vyvolání nějaké akce zobrazil indikátor nahrávání (tzv. loading bar) a byl by zobrazen několik sekund, nepochybně by to hráče natolik otrávilo, že by hru po krátké době opustil. Proto aby byl uživatelský zážitek co nejpříjemnější, byla pro implementaci komunikace obou hráčů použita technologie Web Socket.

5.1 Historie komunikace v reálném čase

Dříve veškerá komunikace probíhala podle HTTP protokolu. Ten ale umožňuje pouze jednosměrnou komunikaci. To znamená, že je nejprve zaslán na server požadavek, který je serverem zpracován a zpět je odeslána odpověď. Spojení je následně ukončeno. Pokud tedy klient očekává nějaká data, vždy musí server o tyto data požádat. Obrácený způsob, kdy by server zaslal data svojí vlastní iniciativou, ve chvíli kdy nějaké má k dispozici, není přes HTTP možný. Výměnu dat mezi serverem a klientem pomocí HTTP lze tedy řešit tak, že je v pravidelných intervalech zasílán požadavek na server, který buď vrátí data, jsou-li k dispozici nebo nikoliv. To je tzv. princip polling či hearthbeat. To si lze představit na modelové situaci kdy běží na serveru nějaká aplikace, která má za úkol prezentovat stavy zařízení, které se aplikaci hlásí. Je nutné stanovit interval ve kterém budou posílat informaci, že jsou aktivní.

Nevýhodou tohoto přístupu je, že není posílána přímo informace o změně, nýbrž se pravidelně musí posílat informace, že stav je stále stejný a ve chvíli kdy očekávaná informace nedorazí, vyvodí se z toho změna stavu. To je nejen zbytečné vytěžování sítě ale navíc platí, že čím delší je interval, tím déle trvá vyhodnocení změny od doby kdy nastala. Pokud zařízení selže a interval je nastaven na několik minut, protože kratší interval není možný pak aplikace promítne stav až během těchto několika minut. Další možnost komunikace přes HTTP je odeslat na server požadavek a nechat ho otevřen až do doby, kdy server má data, která chce poslat v odpovědi - tzv. long-polling či streaming.

5.2 Web Sockety v JavaScriptu

Pojmem soket je rozuměno komunikační spojení dvou koncových uzlů definovaných IP adresou. Vytváří stálý kanál, kterého využívají server a klient pro posílání zpráv. To probíhá v reálném čase oběma směry pomocí jediného soketu. Tato komunikace je založena na protokolu WS, respektive na zabezpečené alternativě WSS. Protokol HTTP je zde použit pro tzv. handshake - úvodní výměnu hlaviček [15].

Při práci s Web soketem v *JavaScriptu* je nejprve potřeba vytvořit instanci objektu WebSocket. Souhrn metod a vlastností objektu webového soketu lze vidět na obr. 5.1

Metody a vlastnosti objektu `WebSocket`

`url`	URL serveru; pouze ke čtení
`protocol`	vrací prázdný řetězec, nebo použitý subprotokol; pouze ke čtení
`readyState`	stav připojení k serveru (hodnoty: 0 – probíhá navázání spojení, 1 – spojení otevřeno, 2 – spojení uzavíráno, 3 – spojení uzavřeno); pouze ke čtení
`bufferedAmount`	počet bytů dat čekajících na odeslání; pouze ke čtení
`binaryType`	určuje, jak má skript interpretovat binární data; výchozí hodnotu `blob`, lze změnit na `arraybuffer`
`send()`	odešle data (řetězec, pole nebo soubor)
`close()`	uzavře spojení

Obrázek 5.1: Přehled metod a vlastností webového soketu [1]

Po vytvoření instance následuje implementace metod, které jsou používány pro soketovou komunikaci. Ukázka kódu této implementace viz. zdroj. kód 5.1

Komunikace funguje na základě výměny zpráv. Zprávy jsou pouhé řetězce, nikoliv objekty. Web soketový objekt implementuje tři události

- onopen - volá se při zahájení spojení - signalizuje, že je možné zasílat zprávy
- onmessage - handler přijetí zprávy
- onclose - volá se při ukončení spojení

Princip soketů v praxi bude vysvětlen na součásti aplikace - *lobby*. Do místnosti se připojí první hráč - vytvoří spojení a čeká se na zprávy. Ihned jakmile se připojí do *lobby* druhý hráč dostane první hráč zprávu o tom, že je dostupný soupeř. Zavolá se handler onmessage a klient ví, že je změna a nechá přerenderovat element vykreslující *lobby*. Jakmile se první hráč rozhodne druhého vyzvat ke hře, přijde druhému hráči pomocí soketu zpráva, že byl vyzván. První hráč poté čeká na zprávu zda protihráč výzvu přijal či nikoliv. V praxi vypadá použití soketů tak, že se nejprve vytvoří spojení a čeká se na zprávy. Konkrétní použití v aplikaci je popsáno v kapitole implementace.

[1] https://www.interval.cz/clanky/komunikace-v-realnem-case-diky-server-sent-events-a-web-sockets/

V současné době jsou web sokety funkční v aktuálních verzích všech webových prohlížečů, kromě jediného a tím je opera mini [2].

Zdroj. kód 5.1: Vytvoření Web Socket objektu

```
let parameters
let mode

const socket = new WebSocket("ws://www.some-server.com/socket-url")
socket.onopen = function() {
    this.send(parameters)
}

socket.onmessage = function(message) {
    parameters = JSON.parse(message.data)
}

socket.onclose() {
    mode = 1;
    console.log("terminated");
}
```

[2] http://www.caniuse.com/#search=web%20socket

6 Analýza a návrh aplikace

Cílem diplomové práce je navrhout a implementovat hru šachy ve 3D. Aplikace je určena pro dva a více hráčů, kteří hrají přes internet ve webovém prohlížeči. Jedná se tedy o webovou aplikaci pracující v reálném čase. Aplikace není vázána na žádnou danou platformu. Serverová část může být vysazena na jakýkoliv server, kde je nainstalován *Node.js*. Ten je multiplatformní a lze ho zprovoznit téměř na každém serveru. Klientská část aplikace pak běží v internetovém přohlížeči. Ten je též multiplatformní. Jediné omezení spočívá v tom, že aplikace potřebuje ke svému chodu WebGL.

6.1 Požadavky

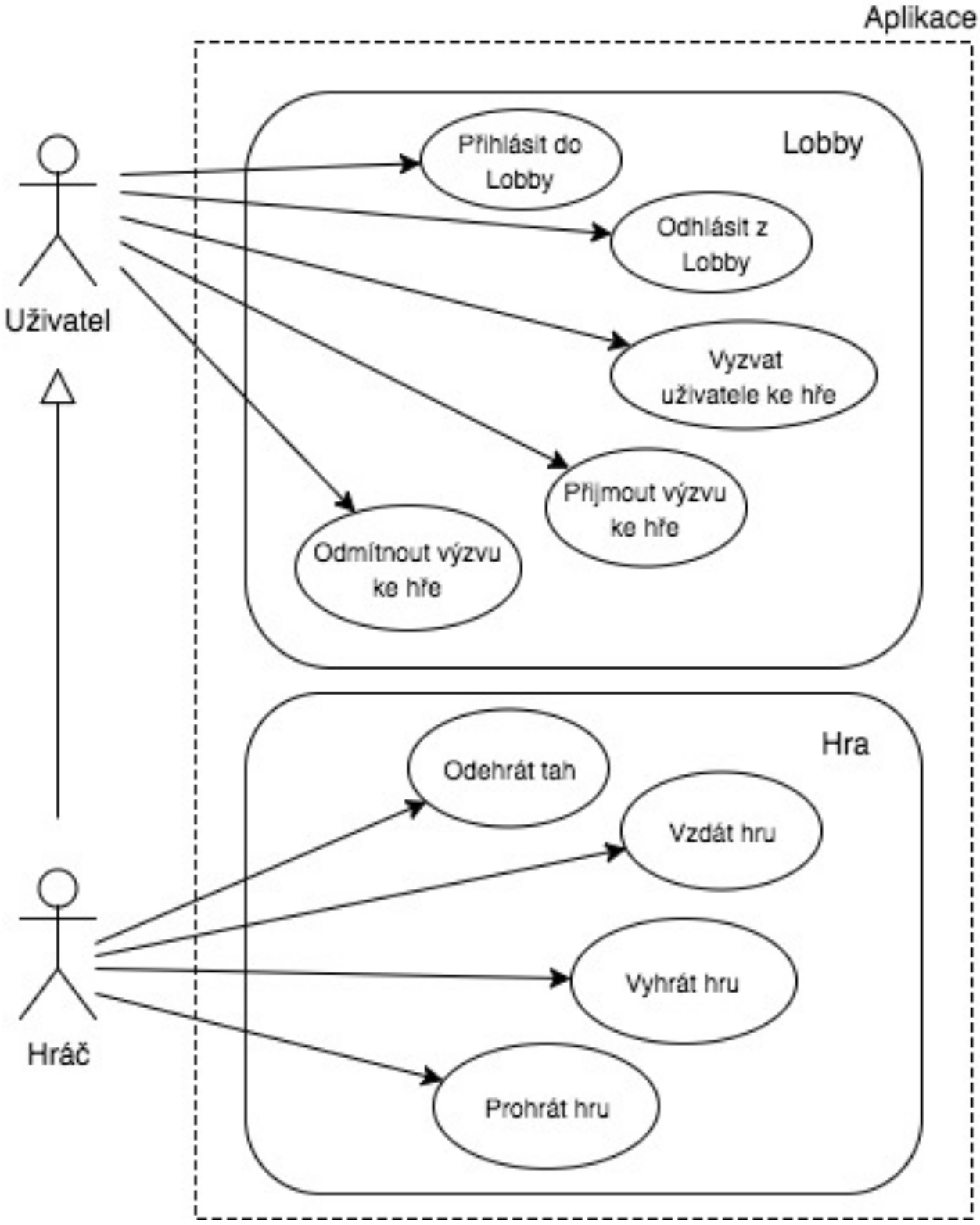
Níže je vypsán seznam požadavků na aplikaci a dále na obr. 6.1 je možné vidět tzv. *use case diagram*.

6.1.1 Funkční požadavky

- zobrazit seznam připojených hráčů
- vyzvat hráče ke hře
- přijmout/odmítnout výzvu od hráče
- zobrazit šahovnici
- provést tah
- zobrazit informaci kdo je na tahu
- zobrazit hráčův i protihráčův zbývající čas
- upozornit v případě šachu

6.1.2 Nefunkční požadavky

- webová aplikace musí být neomezeně dostupná
- webová aplikace je kompatibilní s prohlížeči *Google Chrome, Mozilla Firefox* a *Safari*
- pro chod aplikace je nutné aby byl v prohlížeči zapnut *WebGL*
- aplikace pracuje v reálném čase



Obrázek 6.1: Use case diagram

6.2 Struktura aplikace

Vstupním bodem do aplikace je přihlášení do tzv. *lobby*. *Lobby* je místnost, kde uživatel po přihlášení vidí seznam všech ostatních připojených uživatelů, které může vyzvat ke hře. Pokud některého vyzve a ten výzvu přijme, začíná samotná hra. Hra se skládá z jednotlivých kol, přičemž každé kolo je ukončeno tahem hráče, který je na tahu. Hra končí ve chvíli, kdy jeden z hráčů dá buď protihráči šach mat nebo zahraje takový tah, který vyústí v remízu (pat). Po tomto posledním tahu je zobrazeno informační okno ukazující výsledek hry se souhrnými statistikami. Po prohlédnutí statistik uživatel přejde kliknutím tlačítka zpět na *lobby*.

6.2.1 Lobby

Pro přihlášení do *lobby* je nutné zadat unikátní uživatelské jméno. Pod tímto jménem je uživatel zobrazen pro ostatní uživatele v seznamu hráčů. Přihlášení bude implementováno požadavkem na *API*. V případě úspěšného přihlášení je navrácena odpověď s kódem 200. Je-li přihlášení neúspěšné, je naopak navrácen kód 401 a uživateli se zobrazí informace o nezdařeném příhlášení a žádost o opakování s jiným jménem. Po přihlášení probíhá další komunikace pomocí socketů. Do budoucna lze implementovat registraci a do *lobby* by pak mohl být uživatel přihlášen automaticky pod svým přihlašovacím jménem účtu.

Seznam hráčů je klikatelný seznam, který po vybrání hráče nabízí možnost vyzvat vybraného hráče ke hře. Při odeslání výzvy se zobrazí dialogové okno informující o probíhající výzvě s možností tuto výzvu zrušit. Protihráč, který obdrží výzvu má pak viditelný dialog informující o tom, že byl vyzván s možnostmi buď výzvu přijmout nebo odmítnout. Při odmítnutí se dialog s výzvou schová a uživatel má tak zobrazenou klasickou *lobby*. Naopak pokud výzvu přijme, spouští se samotná hra.

6.2.2 Hra

Hra začíná přijmutím výzvy v *Lobby*. Nejprve se na serveru náhodně vybere hráč, kterému bude přidělena bílá barva a bude tedy začínat. Po načtení hry se zobrazí v hlavičce informační prvky - který hráč je na tahu, zda je hráč v šachu, kolik času oběma hráčům zbývá do konce hry atd. Dále je zde tlačítko „vzdát se“, kterým lze hru ukončit.

Dalším prvkem stránky je samotná 3D scéna. Stavebním kamenem celé scény je šachovnice. Ta se skládá z šedesáti čtyř šachových polí, rozdělených do osmi řad po 8 sloupcích. Šachovnice je postavena na podložce a zbytek scény obklopuje černé pozadí. Na šachovnici se nachází modely jednotlivých šachových figurek. Po najetí myší na objekty figurek či polí dojde ke změně barvy aby bylo dobře viditelné, že je objekt vybrán. Po kliknutí na figurku, či pole, na kterém je přítomna figurka, dojde k vybrání této figurky jako aktivní a zároveň jsou barevně vyznačena pole, na které je možné danou figurkou táhnout. Dalším kliknutím na některé z vyznačených polí dojde k přesunu figurky. V případě, že se na vybraném poli nachází soupeřova figurka, je sebrána. Na pole obsahující figurku hráče, který táhne, se nelze přesunout.

Ve hře jsou implementované algoritmy, jejichž implementace vychází z herních pravidel:

- nelze táhnout na pole, na kterém se nachází vlastní figurka

- je-li vybrán pěšec na startovní řadě pak má možnost posunout se o jedno či dvě pole
- má-li tah protihráče za následek šach, je v hlavičce zobrazena informace o šachu
- nikdy nelze táhnout tak, aby následkem tahu vznikl šach protihráčem
- v případě, že je hráč v šachu, může táhnout pouze tak, aby šachu zabránil
- pokud nelze zabránit šachu, je konec hry (šach mat). Vítězí protihráč
- pokud nelze odehrát žádný tah, který by nevedl k šachu, jedná se o tzv. pat a hra končí remízou
- dojde-li pěšec až na poslední pole šachovnice stává se z něj dáma
- hra končí v případě, že některému z hráčů dojde čas
- čas je hráči přidělen na začátku podle nastavení hry
- čas se hráči odečítá v případě, že je na tahu

Zvláštním případem tahu je tzv. Rošáda. Jedná se o jediný tah, během kterého lze táhnout dvěmi figurami najednou - králem a věží. Rošáda se provádí tak, že se král posune o dvě pole směrem k věži a věž se přemístí na pole mezi původním a aktuálním polem krále. Aby bylo možné provést rošádu je nutné splnit všechny tyto podmínky:

- králem nebylo ještě táhnuto
- s příslušnou věží nebylo ještě táhnuto
- mezi králem a věží nestojí žádná jiná figura
- král nesmí přejít přes žádné pole, které ohrožuje nepřátelská figura (byl by v šachu)
- král před provedením a po provedení rošády nestojí v pozici šach
- král a vybraná věž musí stát na stejné řadě

6.2.3 Konečné statistiky

Po ukončení hry se zobrazí statistiky celé hry. Ve statistikách jsou zobrazeny zbývající časy obou hráčů. Do budoucna se zde mohou objevit další informace jako např. počet tahů, počet ztracených figurek, průměrný čas na tah, případně možnost proklikat si zpětně celou hru po jednotlivých tazích.

6.3 Návrh aplikace

Návrh aplikace začal nakreslením tzv. wireframes, tedy drátěných modelů budoucí webové stránky. Tím bylo nadefinováno, jak by mělo principielně vypadat uživatelské rozhraní. Dále byl vytvořen logický model aplikace, kde došlo k návrhu jednotlivých modelů, které v aplikaci figurují.

6.3.1 Drátěné modely

V příloze A této práce jsou k dispozici drátěné modely. Jak je načrtnuto na modelu *lobby*, při spuštění aplikace se zobrazí stránka přihlášení. Ta se skládá z jednoduchého textového pole a tlačítka. Po přihlášení se objeví *lobby*, které se skládá z horní části, kde jsou zobrazeny informace o výzvách. Výzvy se skládájí z informačního textu, které vysvětlují o jakou výzvu se jedná. V případě odchozí výzvy je zobrazeno jméno hráče, kterému byla výzva odeslána. V opačném případě jméno hráče, který výzvu zaslal a dvě tlačítka - přijmout a odmítnout. Dále se v dolní části nachází seznam hráčů, skládající se ze jmen přihlášených hráčů, řazených pod sebou. Na jméno hráče lze kliknout, čímž se danému hráči pošle výzva.

6.3.2 Modely

Celá aplikace je složena z jednotlivých modelů. Modely v aplikaci jsou implementovány pomocí ES6 javascriptových tříd. Model má nadefinované své atributy, metody a vazby na ostatní modely. K popsání vlastností modelů aplikace byl vytvořen logický model viz. příloha B. Níže budou tyto modely obecně vysvětleny. V sedmé kapitole bude pak podrobně vyložena jejich implementace.

Uživatel (`User`)

Tento model popisuje uživatele, který se přihlásí do `Lobby`. Jeho jediným parametrem je jméno.

Hráč (`Player`)

Hráč je uživatel, který se již dostal do hry a obsahuje navíc další parametry. Každý hráč má svoji barvu (černá/bílá), která je náhodně určena na serveru v době zahájení hry. Bílý hráč zahajuje hru. Dále má hráč na sobě zbývající čas do konce hry a nakonec Id vygenerované databází.

Místnost (`Lobby`)

`Lobby` obsahuje pole přihlášených uživatelů. V tuto chvíli je počítáno pouze s jednou aktivní výzvou. `Lobby` tedy dále obsahuje jméno vyzývajícího či vyzvaného hráče a příznak zda je výzva příchozí či odchozí. Do budoucna lze místo jedné výzvy implementovat podporu pro více aktivních výzev.

Hra (`Game`)

Každá hra musí mít dva hráče. Dalšími parametry jsou pole odehraných kol, časové údaje hry (začátek, konec), herní nastavení a vygenerované Id.

Herní kolo (`Round`)

Herní kolo má opět vygenerované Id, dále odehraný tah a časové údaje o kole.

Tah (`Move`)

Herní tah se skládá z hráče, který táhl, počátečního pole, ze kterého byl tah proveden a cílového pole, na které bylo taženo. V případě, že daný tah byla rošáda, pak navíc obsahuje počáteční a cílové pole věže se kterou byla rošáda provedena. I tah má vygenerované Id.

Pole (`Field`)

Herní pole má své id podle umístění na šachovnici. Očíslování polí odpovídá klasické matici. První pole v pravo u bílého hráče má index 00. Poslední pole, tedy v pravo u černého, hráče pak 77. Dále má pole referenci na figurku, která se na poli nachází. Metody jsou pak nastavení pole jako aktivní a nastavení figurky.

Figurka (`Piece`)

Figurka má svůj typ, což je výčet možných figurek. Dále má figurka referenci na hráče, kterému patří a na pole, na kterém se nachází. Implementovanou metodou je poté provedení tahu na jiné pole.

Šachovnice (`Chessboard`)

Šachovnice má referenci na hráče, protihráče a jednotlivá pole. Dále si tento model uchovává pole šachových polí kam je možné táhnout. Tento model má na starosti inicializaci po začátku hry, odeslání uskutečněného tahu a spracování příchozího tahu.

7 Implementace aplikace

Implementaci lze rozdělit na dvě části - serverová a klientská část aplikace. Server slouží jako prostředník mezi klienty, kteří spolu komunikují v rámci hry. Dále pracuje s databází, do které ukládá data. Tyto data jsou následně poskytována prostřednictvím veřejného *API*. Co se týče klientské části, jedná se o webovou aplikaci, která je spuštěna v prohlížeči. Pro uskutečnění hry jsou potřeba dva klienti, kteří se rozhodnou zahrát si na základě výzvy jednoho hráče druhým. Jako vývojové prostředí pro implementaci byl zvolen *JetBrains WebStorm*, který je vhodný jak pro psaní serveru tak klienta.

7.1 Struktura projektu

Projekt je rozdělen na serverovou a klientskou část. Dále se v projektu nachází soubory, které jsou pro obě části společné. Jedná se o modely a herní logiku. Ta je nutná pro klienta aby bylo možné skrz UI hru ovládat a na serveru je nezbytná pro kontrolu.

7.1.1 Server

Zdrojové soubory serverové části projektu se nachází ve složce src. První věc, která se zde nachází jsou konfigurační soubory - globální a lokální. V globálním konfigu se nachází konfigurace, která je určena pro výchozí hodnoty a je součástí verzování - je tedy vždy kopírován s projektem. Hodnoty, které se mají v daném projektu změnit se nachází v konfigu lokálním. Lokální konfig se nachází v .gitignore a je tedy při verzování ignorován. Dále se na serveru nachází controllery. Ty slouží pro komunikaci s *API* a sokety. Routery pak definují *API*. Každá routa má svojí cestu a HTTP metodu. Důležitou složkou je public. Zde se nachází soubory, které se stahují ze serveru na klienta. Jedná se o minifikovaný bundle reactového kódu a knihovnu *babylon.js*.

7.1.2 Klient

Klientská část je rozdělena do několika složek podle architektury redux - akce, reducery, store, komponenty a stránky, což jsou také komponenty. Dále se zda nachází konstanty, které jsou napříč aplikací používány. Stejně jako na serveru i zde se nachází konfigurační soubory - globální a lokální.

7.2 Sokety

Pro komunikaci v reálném čase jsou v aplikaci použity sokety. Jedná se o implementaci web soketů, konkrétně knihovnu Socket.IO. Soket je inicializován ihned při spuštění aplikace a následně klient obdrží první zprávou seznam všech přihlášených hráčů v *lobby*. Zároveň naslouchá na události *login* a *logout*, které upravují seznam hráčů, podle toho jak se odhlašují stávající a přihlašují noví hráči. Uživatel samotný sice ještě v *lobby* přihlášen není ale tím, že je seznam hráčů připraven ještě před vstupem, se zminimalizuje prodleva pro následné vykreslení *lobby*.

Dále ve hře se sockety používají pro výměnu informací o provedených tazích. Při každém tahu pošle klient informace o tomto tahu na server. Ze serveru je přeposlán soketem protihráči. Ten po přijetí soketu daný tah vykreslí a je následně na tahu. Ukázka této soketové komunikace mezi klienty prostřednictvím serveru je ke shlédnutí v příloze C

Kompletní výčet soketových zpráv využitých při komunikaci je přiložen jako příloha D.

7.3 REST API

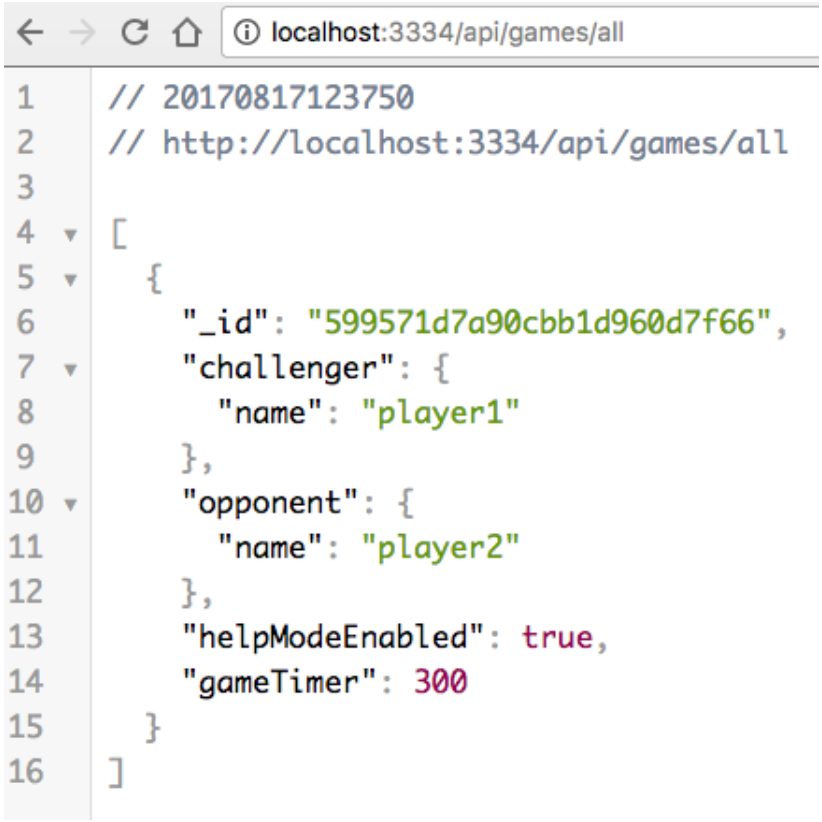
Kromě soketů, lze se serverem komunikovat také pomocí REST API. Toto API se skládá ze dvou částí - veřejného a privátního. Veřejné API slouží k poskytování informací uživatelům. Je přístupné pomocí metody GET a uživatel si vybírá o jaké informace má zájem prostřednictvím URL adresy. V rámci aplikace je pro ukázku pouze jedna URL, která poskytuje seznam všech uskutečněných her. Do budoucna lze toto veřejné API rozšířit o různé informace, které by mohli být pro uživatele zajímavé. REST API nabízí možnost nad aplikací stavět další aplikace, které pomocí tohoto API mezi sebou komunikují. Nejčastěji to můžou být např. mobilní aplikace. Ukázka zobrazení informací z API pomocí prohlížeče viz. obr. 7.1. Privátní část API pak slouží pro účely aplikace samotné. Opět je implementována jedna API metoda a tou je POST na login do *lobby*. REST API dokumentace viz. zdroj. kód 7.1

7.4 Herní algoritmy

Základní představení herních algoritmů proběhlo již v předchozí kapitole. Zde budou tyto algoritmy vysvětleny podrobněji, včetně popisu jejich implementace.

7.4.1 Inicializace scény

Vlastní renderování šachovnice je uskutečněné v *React* komponentě `Game`, kde se pracuje s javascriptovou knihovnou pro práci s 3D scénou - *babylon.js*. Nejprve dojde k vytvoření a přípravě prázdné scény - nastavení parametrů (světlo, kamera, definice povrchů...), načtění modelů figurek atd. Komponenta `Game` má referenci na model `ChessBoard`, což je statická třída prezentující model šachovnice. Komponenta na této třídě volá inicializační metody. Jako parametry předává oba hráče, ze kterých je díky barvě určen hráč, který je na tahu. Dále je předána reference na objekt scény, která je dále poskytována dalším modelům. Díky této referenci na scénu, mohou se scénou modely pracovat, např. přesouvat figurky.



Obrázek 7.1: Ukázka zobrazení informací z REST API

Zdroj. kód 7.1: Dokumentace REST API

```
'GET' api_url/api/games/all

returns:
    '200' all games from database
    '500' error occured

//------------------------------------------------------

'POST' api_url/api/login

body: {
    username: "unique username"
}

returns:
    '200' once username is unique and login is successful
    '400' login is unsuccessful - username already exists
    '500' login is unsuccessful - error occured
```

Zdroj. kód 7.2: Listener na výběr objektu v babylon.js

```
window.addEventListener("click", function () {
    // Try to pick an object
    var pickResult = scene.pick(scene.pointerX, scene.pointerY);

    if (pickResult.pickedMesh !== null) {
        if (pickResult.pickedMesh.id.indexOf("field") >= 0) {
            pickResult.pickedMesh.attrs.poleRef.setActive();
        }
        if (pickResult.pickedMesh.id.indexOf("piece") >= 0) {
            pickResult.pickedMesh.attrs.figurkaRef.pole.setActive();
        }
    }
});
```

Při spuštění hry je nutné vykreslit scénu ve výchozím stavu. Dojde tedy k sestavení šachovnice, nejprve vytvořením všech šachových polí. To je realizováno průchodem maticí a střídavého přepínání barvy mezi černou a bílou. Dále je nutné umístit na správná pole všechny figurky obou hráčů. Figurka se při inicializaci umístí na pole stejně jako při hře voláním metody `moveTo` na figurce. Z důvodu aby nebyla volána další logika ze hry, je při inicializaci nastaven příznak `ChessBoard.ini`. Po dokončení inicializace vytvořením šachovnice je tento příznak zrušen.

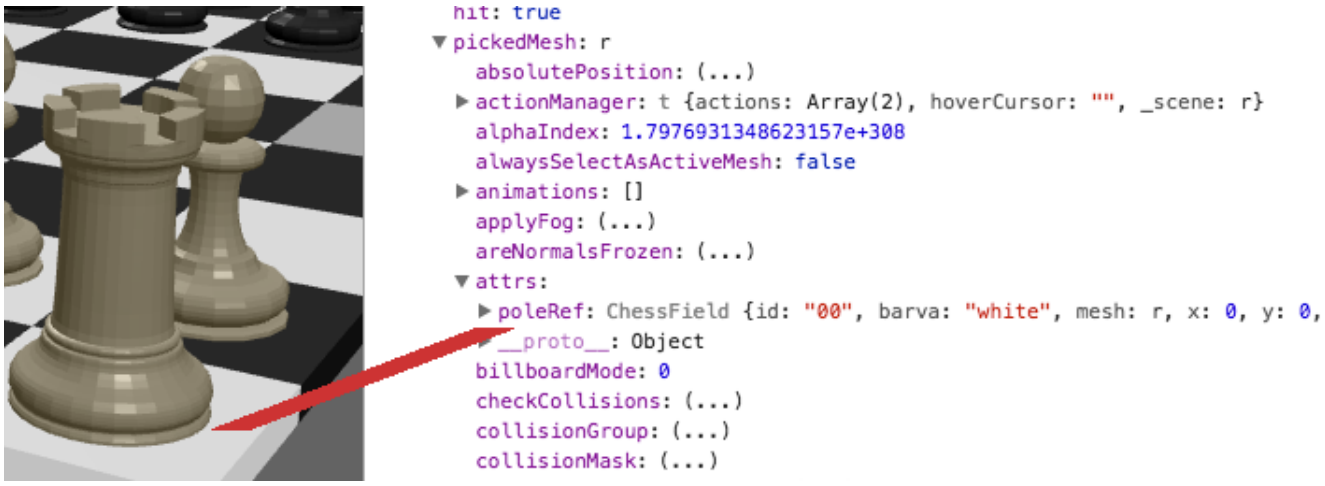
Ukázka zdrojového kódu vytvoření šachového pole je přiložena v příloze E. Nejprve se vytvoří barva, která je v Babylonu prezentována objektem `BABYLON.Color3`. Pro vytvoření se jako parametry posílají barevné složky RGB. Následně se volá `BABYLON.Mesh.CreateBox` a tím vzniká požadované 3D šachové pole. Dále se přiřadí vytvořený materiál, který je definován barvami. Nakonec se nastaví souřadnice pole a volá se funkce, která nastaví barvu, která se poli nastavuje ve chvíli, kdy je označeno myší.

Jakmile je inicializace hotová tak se již vyčkává na interakci uživatele.

Pro výběr figurky či pole je registrován listener (zdroj. kód 7.2), ve kterém se odchytí uživatelem vybraný objekt. Přitom se rozlišuje zda je vybraný objekt figurka či pole. V obou případech se na poli zavolá metoda `setActive`. Pokud je vybrána figurka, pak se k poli přistupuje přes referenci, kterou figurka na pole má. Snímek obrazovky zachycující referenci vybraného objektu (objekt typu `pickedMesh`) na instanci třídy `ChessField` v prohlížeči je na obr. 7.2.

7.4.2 Tah figurkou

Po zavolání metody `setActive` na třídě `ChessField` se buď označí pole jako aktivní nebo se provede tah, v případě, že už je aktivní jiné pole a na aktuálně kliknuté pole lze provést tah. V obou případech se jako první věc po kliknutí nejprve zjistí, zda je hráč na tahu. Pokud ne tak se volání metody ignoruje. Je-li však hráč na tahu, pak se dále zjišťuje zda je již nějaké pole vybrané jako aktivní a pokud ano tak je odznačeno. Zároveň se volá metoda na pročištění polí, v případě, že je možná rošáda. Dále je zkontrolováno, zda pole nemá nastaven příznak, že je možné na pole táhnout. Pokud ano je proveden tah a funkce končí. V opačném případě se podle reference na figurku zjistí, zda je na poli umístěna figurka patřící hráči. Pokud ano volá se další



Obrázek 7.2: Ukázka reference na pole z 3D scény

metoda šachového pole a to `getMoves` Každý typ figurky má povolené jiné kroky, proto se v této metodě nachází switch, který podle typu figurky dále volá příslušnou metodu na objektu `Moves`.

`Moves` je statická třída, která obsahuje metody pro získání možných tahů pro všechny typy figurek. Součástí této logiky je i zjišťování šachu. Konkrétně pak výpočet možných tahů pro krále je celkem složitý, protože je nutné kontrolovat aby se nedostal hráč do šachu. Pro každý typ figurek je implementovaná metoda `get` název figurky `Moves`. Princip algoritmů všech figurek je stejný až na jednu vyjímku, kterou je Pěšec. Ten je velmi specifický a je pro něj napsán zvláštní algoritmus. Dále je nutné u krále počítat již se zmíněným šachem a dále také s rošádou.

Potom co je vybrána figurka, je podle jejího typu zavolána metoda `getMoves` s názvem figurky. V parametru je posláno pole, na kterém daná figurka stojí. Uvnitř metody se pak z pole vytáhnou jeho souřadnice X a Y. Dále je postup takový, že se zkouší pole ve všech směrech, kterými lze figurkou táhnout. Např. věž se pohybuje po řadách a sloupcích, střelec po diagonálách. Pro ověření zda figurka může nějakým směrem táhnout se volá metoda `addMove`, s tím, že v parametrech se zasílají souřadnice cílového pole. Souřadnice cílového pole jsou vypočítány na základě testovaného pohybu. V případě bílé věže, která stojí na souřadnicích x:0, y:0 by se pak například pro posun dopředu použili souřadnice x=x+1, y=y atd. Metoda `addMove` je volána na aktivním poli a díky souřadnicím z parametru je možno dostat referenci na cílové pole. Metoda vrací true v případě, že na pole lze táhnout a jelikož je volaná ve while cyklu pak se bude volat až dokud nevrátí false. Ten se vrátí buď v případě, že se na poli již nachází figurka nebo dojde k posunu mimo šachovnici. Ukázku získání možných tahů věže je možno vidět na zdroj. kód 7.3

V případě pěšce je algoritmus specifický. Je nutné psát kód zvlášť pro obě barvy hráče, jelikož pěšec se posouvá stále kupředu narozdíl od ostatních typů figurek, které se mohou pohybovat různými směry. Dále se rozlišuje posun dopředu po řadě, kdy je podmínka, že na cílovém poli nesmí být figurka a naopak posun diagonálně, kdy pro změnu na poli musí být protihráčova figurka. Také se kontroluje zda je pěšec na startovní řadě. Pokud ano tak má možnost se posunout o dvě pole. Opět pouze pokud se na cílovém poli nenachází žádná figurka. Ukázka tohoto algoritmu viz. zdroj. kód 7.4. Celý výše popsaný algoritmus na výpočet možných tahů je schématicky vyobrazen na obr. 7.3

Zdroj. kód 7.3: Ukázka zdrojového kódu tahu věže

```
static getRookMoves(field, x, y) {
    var x2 = x;
    var y2 = y;
    while (field.addMove(x2 - 1, y2)) {
        x2 = x2 - 1;
    }
    x2 = x;
    y2 = y;
    while (field.addMove(x2 + 1, y2)) {
        x2 = x2 + 1;
    }
    x2 = x;
    y2 = y;
    while (field.addMove(x2, y2 - 1)) {
        y2 = y2 - 1;
    }
    x2 = x;
    y2 = y;
    while (field.addMove(x2, y2 + 1)) {
        y2 = y2 + 1;
    }
}
```

Při provedení tahu je nutné na cílovém šachovém poli nastavit aktuální figurku. Metoda nastavení figurky se volá i při inicializaci, proto je přítomna podmínka, která zruší referenci na figurku v původním poli pouze v případě, že je původní pole nastavené (při inicializaci neexistuje). Pokud cílové pole obsahuje soupeřovu figurku, pak se na její referenci volá metoda `.mesh.dispose()`. Tento příkaz se postará o to, že je figurka odebrána ze scény. Nakonec dojde k nastavení reference a nových souřadnic figurky ve scéně. Ukázka tohoto algoritmu viz. zdroj. kód 7.5.

7.4.3 Odpočet času a synchronizace

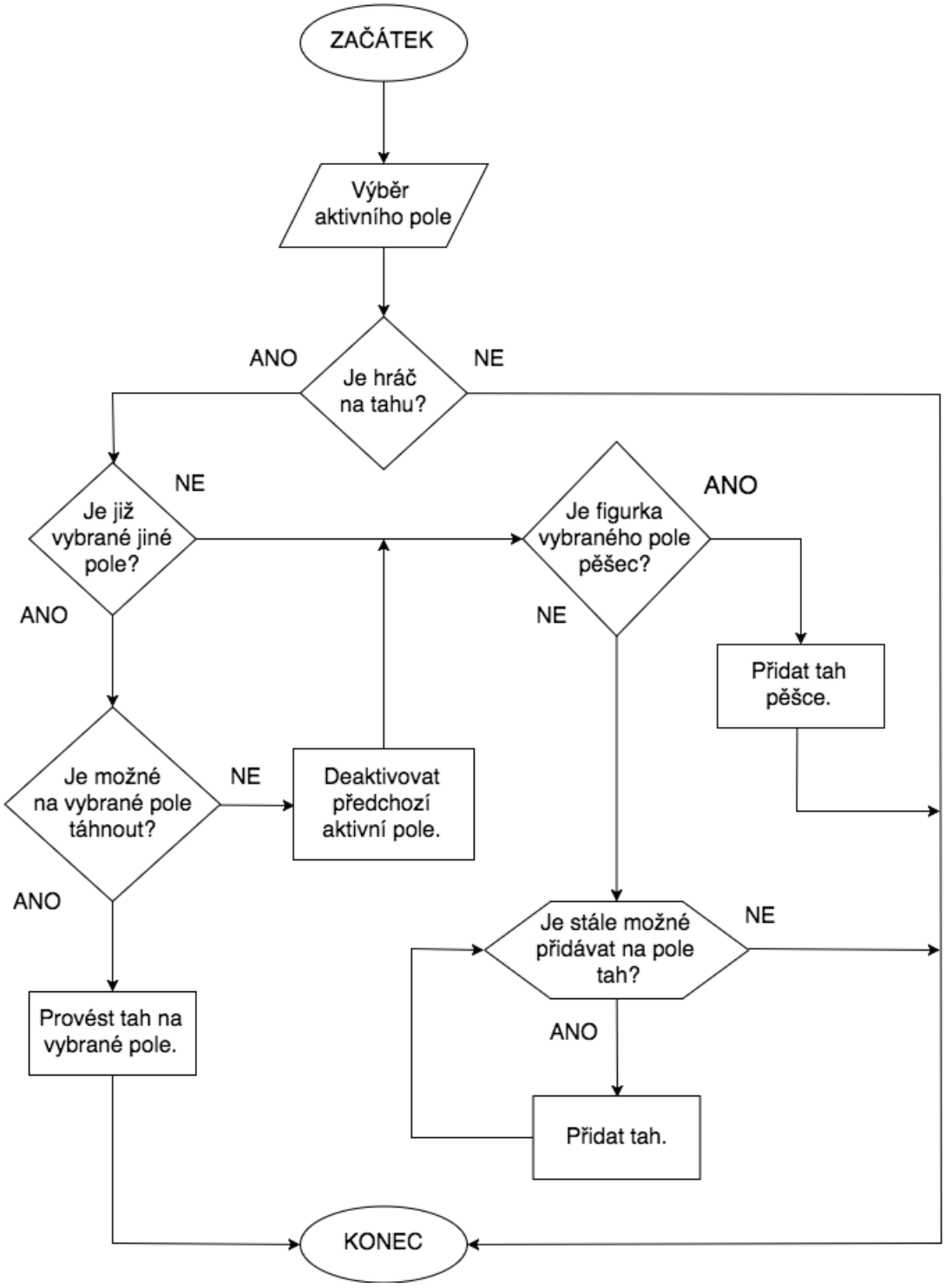
Zbývající časy obou hráčů (v sekundách) jsou uloženy na serveru. Inicializovány jsou při startu hry, hodnota je nastavena podle nastavení hry. Odpočet času probíhá současně na serveru i na klientovi. Oba časy jsou pak porovnávány. Toto je nutné z důvodu bezpečnosti a optimalizace. Počítat čas pouze na klientovi nestačí z toho důvodu, že zaslání nového času po skončení kola lze na klientovi podvrhnout. Klient si může v prohlížeči zobrazit jaká data jsou ze serveru přijímána a jaká jsou odesílána. Není problém např. údaj o času změnit a na server by pak klidně mohl přijít falešný čas. Na druhou stranu počítat pouze na serveru také není dostačující a to z důvodu synchronizace. Data mohou ze serveru na klienta nějakou dobu přenášet a mohlo by se tak stát, zejména pokud by uživatel měl pomalé připojení, že čas by byl ze serveru odeslán, začal by se odpočítavat ale na klienta by dorazil až za několik sekund. Stejná situace nastává ve chvíli kdy hráč tah odehraje a posílá údaje o tahu na server. Opět to může nějakou dobu při

Zdroj. kód 7.4: Ukázka zdrojového kódu tahu pěšce

```
static getPawnMoves(field, x, y) {
    const playerColor = ChessBoard.getPlayer().color
    if (playerColor === "white") {
        if (this.isClear(x + 1, y)) {
            field.addMove(x + 1, y)
            if (field.x === 1 && this.isClear(x + 2, y)) {
                field.addMove(x + 2, y)
            }
        }
        if (this.containsPiece(x + 1, y - 1, "black")) {
            field.addMove(x + 1, y - 1)
        }
        if (this.containsPiece(x + 1, y + 1, "black")) {
            field.addMove(x + 1, y + 1)
        }
    }
    if (playerColor === "black") {
        if (this.isClear(x - 1, y)) {
            field.addMove(x - 1, y)
            if (field.x === 6 && this.isClear(x - 2, y)) {
                field.addMove(x - 2, y)
            }
        }
        if (this.containsPiece(x - 1, y - 1, "white")) {
            field.addMove(x - 1, y - 1)
        }
        if (this.containsPiece(x - 1, y + 1, "white")) {
            field.addMove(x - 1, y + 1)
        }
    }
}
```

Zdroj. kód 7.5: Ukázka nastavení figurky na poli

```
setFigurka(figurka) {
    if (figurka.pole)
        figurka.pole.figurka = null
    figurka.pole = this
    if (this.figurka) {
        this.figurka.pole = null
        this.figurka.mesh.dispose()
    }
    this.figurka = figurka
    figurka.mesh.position.x = this.mesh.position.x
    figurka.mesh.position.z = this.mesh.position.z
}
```



Obrázek 7.3: Schéma algoritmu výpočtu tahů

pomalém připojení trvat. O tuto dobu by v odpočítaném čase byl klient ochuzen. Optimální řešení tedy je zaprvé ujistit se, že se začne na serveru odpočítavat čas až ve chvíli kdy klient obdrží data o tahu a za druhé příchozí čas od klienta zkontrolovat se serverovým.

Algoritmus odpočtu času je tedy takový:

- při začátku hry inicializovat na serveru
- zaslat data na oba klienty
- klient, který je na tahu, pošle na server potvrzení, že data obdržel a začne odpočítávat
- server začne po přijetí potvrzení odpočítávat
- po provedení tahu zasílá klient údaje o tahu včetně aktuálního času
- server zkontroluje přijatý čas se svým časem
- pokud se časi liší o víc jak 5 sekund použije server svůj (odfiltrování podvržených dat)

Zde je prostor pro různé složitější optimalizace. Zvolený algoritmus stále není dokonalý, protože např. při odeslání potvrzení od klienta může také dojít k delší odezvě. Proto je např. možnost při přihlášení do *lobby* vyzkoušet poslat nějaké requesty a spočítat průměrnou odezvu. Na základě tohoto měření lze buď hráče vyloučit z *lobby* pokud by odezva byla opravdu špatná, protože by ostatním hráčům znepříjemňoval hru. Pokud by odezva nebyla tak dramatická nicméně byla by znatelná, pak lze např. do odečítaného času zahrnout průměrný čas odezvy. V případě, že s odezvou hráče začne být problém až během hry, pak zafunguje pěti sekundová kontrola na serveru. Nemůže se tedy stát že hráč uvidí ubíhat protihráčův čas a po uskutečnění tahu by se tento čas např. zmenšil o polovinu z důvodu, že se od protihráče druhou polovinu času pouze posílala data. Na druhou stranu to znamená, že hráč, který má problémy s připojením, může odehrát tah a zbývající čas se mu může zkrátit o dobu komunikace. Z toho důvodu by se mohlo provádět zmíněné měření a v případě velké odezvy zobrazit upozornění o problému s připojením. Priorita je určitě udržet spojení dokud je to možné aby nedošlo ke ztrátě hry a znevýhodnit hráče, který má odezvu v pořádku by nebylo spravedlivé. Proto je toto nejlepší cesta. Na druhou stranu je možné v případě problémů s připojením hru přerušit a nabídnout možnost dohrát hru později. Nicméně to poskytuje prostor pro další podvody typu, že hráč neví jak táhnout tak by záměrně vypnul internet a tah promyslel.

7.5 Použité moduly

Aplikace je rozdělena na serverovou a klientskou část. Seznam modulů, které jsou vyžadovány pro běh aplikace, se nachází v souboru package.json a to zvlášť pro server a pro klienta. Všechny tyto moduly je nutné před spuštěním aplikace nainstalovat příkazem npm install, jak ve složce s klientem tak ve složce serverové.

Obsah obou souborů package.json a tedy seznam všech použitých modulů je v přiložen v příloze F. Tyto moduly jsou popsány níže, zvlášť klientské a serverové.

7.5.1 Klientské moduly

V případě klienta je většina modulů určená pro samotný vývoj aplikace (devDependencies). Nachází se zde *Babel* a jeho různé podmoduly. Dále je v seznamu *React* společně s *reduxem*. Poslední položkou je *Webpack*. Všechny tyto moduly s jejich významem jsou popsány v teoretické části. V druhé části souboru jsou pak moduly potřebné pro samotný chod aplikace nikoliv vývoj. Zde je modul pro práci s cookies, dále *socket.io*, implementující komunikaci přes web soket a nakonec *superagent*, který slouží pro komunikaci s *REST API* pomocí *AJAX* requestů.

7.5.2 Serverové moduly

Na straně serveru je stejně jako na klientovi použit *socket.io*. Dále databáze *mongoDB* a node framework *express*. Za zmínku pak stojí např. *body-parser*, který slouží k parsování requestů nebo *winston* využívaný k logování. V modulech pro vývoj se pak nachází také *Babel*, protože i serverovou část je nutné transpilovat z ES6. *Nodemon* je pak využitý pro restartování serveru v případě změn.

8 Nasazení a spuštění aplikace

V této poslední kapitole bude vysvětleno jak probíhá nasazení aplikace na server a bude prezentován výsledný vzhled aplikace. Nasazení aplikace bylo uskutečněno na virtuálním serveru hostovaném firmou Wedos, který běží na operačním systému *FreeBSD*. Aby bylo možné nasazenou aplikaci zpřístupnit, byla zakoupena doména [1] , na které lze aplikaci otestovat.

8.1 Nasazení aplikace

Jako první věc je nutné nainstalovat *Node.js* a *MongoDB*. Instalace těchto závislostí je rozdílná pro různé operační systémy. Na systému *FreeBSD* stačí použít balíčkový systém *pkg* [16]. Instalace *Node.js* a *MongoDB* (včetně spuštění) probíhá zadáním následujících příkazů do příkazové řádky (terminálu), viz. zdroj. kód 8.1.

Zdroj. kód 8.1: Příkazy pro instalaci Node.js a MongoDB

```
// zavolat kdekoliv na serveru

// instalace Node.js a npm
pkg install node

// instalace MongoDB
pkg install mongodb

// spusteni MongoDB
service mongod start
```

Nyní už je možné vše zbývající nainstalovat pomocí balíčkovacího systému *npm*, který se nainstaloval společně s *Node.js*. Příkazy s *npm* jsou již pro všechny operační systémy společné. Nyní se nainstalují moduly, jejichž seznam se nachází v package.json, na klientovi i na serveru. Příkazy viz. zdroj. kód 8.2

Zdroj. kód 8.2: Příkazy pro instalaci modulů

```
// zavolat v rootu projektu - instalace serverovych modulu
npm install

// prepnout do klienta
cd src/client

// instalace klientskych modulu
npm install
```

[1] www.chessplay.eu

Po instalaci všech modulů je ještě potřebné správně nastavit aplikaci. Pro nastavení slouží konfigurační soubory. V serverovém se vyplňují porty, na kterých běži aplikace a sokety. V klientském pak příslušné URL adresy. V případě vysazování na lokálním stroji se vyplní localhost se zvolenými porty. Při vysazování na server je pak nutné využít doménu. O použití domény pojednává další podkapitola. Ukázka vyplnění konfigurace viz. zdroj. kód 8.3

Zdroj. kód 8.3: Příkazy k vyplnění konfigurace

```
// otevrit lokalni serverovy konfiguracni soubor
nano src/configurations/local.js

// vyplnit port, na kterem aplikace bezi a port, na kterem bezi sokety
module.exports = {
    environment: 'prod',
    prod: {
        port: 3000,
        socketPort: 3001
    }
}

// otevrit lokalni klientsky konfiguracni soubor
nano src/client/src/configurations/local.js

// vyplnit url, na ktere aplikace bezi a url, kde bezi sokety
export default {
    apiUrl: 'localhost:3000',
    socketUrl: 'localhost:3001'
}
```

Po nastavení aplikace již lze provést build klientských kódu, tzv. bundlu. Bundle je jeden *JavaScriptový* soubor obsahující veškerý kód transpilovaný do klasického *JavaScriptu*. Tento soubor se načítá do prohlížeče každého klienta, který aplikaci spustí. Pro provedení buildu se používá modul *Webpack*, který navíc výsledný soubor minifikuje. Minifikace znamená, že je soubor zmenšen na co nejmenší velikost. Dojde tedy např. k vymazání mezer a uživatel soubor stáhne rychleji. *Webpack* je vhodné instalovat na vysazovací server globálně, tzn. lze ho zavolat odkudkoliv, není vázán na konkrétní projekt. Příkazy viz. zdroj. kód 8.4.

Zdroj. kód 8.4: Příkazy pro sestavení buildu klienta

```
// lze volat odkudkoliv - globalni instalace knihovny webpack
npm install -g webpack

// zavolat v klientovi (src/client) - minifikovany build klientske
   casti
webpack --env.minify
```

Nyní je již možné aplikaci spustit. Příkaz viz. zdroj. kód 8.5.

Zdroj. kód 8.5: Příkaz pro spuštění aplikace

```
// zavolat v rootu projektu

node_modules/babel-cli/bin/babel-node.js src/index.js --presets es2015,
   stage-2
```

8.2 Vystavení na doménu

Má-li být aplikace dostupná na nějaké doméně, je nutné nastavit aplikační server. Ukázka bude provedena pro server *NGINX*. Ještě před samotným nastavením na straně serveru je nutné změnit DNS záznamy domény. Toto nastavení se může u různých registrátorů lišit. V případě wedosu se nastavení DNS záznamů nachází v detailu domény. Zde se nastaví A záznam na IP adresu serveru, kde aplikace běží.

Pokud je DNS záznam správně nastaven, dojde při vyplnění domény v prohlížeči k přesměrování na vysazovací server. Přesměrování na běžící aplikaci pak provádí *NGINX*. Aby *NGINX* věděl, kam přesměrovat, je nutné nastavit v jeho konfiguračních souborech danou instanci. V instanci je uveden název domény a lokální IP adresa s portem běžící aplikace. Příkazy viz. zdroj. kód 8.6.

Zdroj. kód 8.6: Příkazy pro přidání NGINX instancí

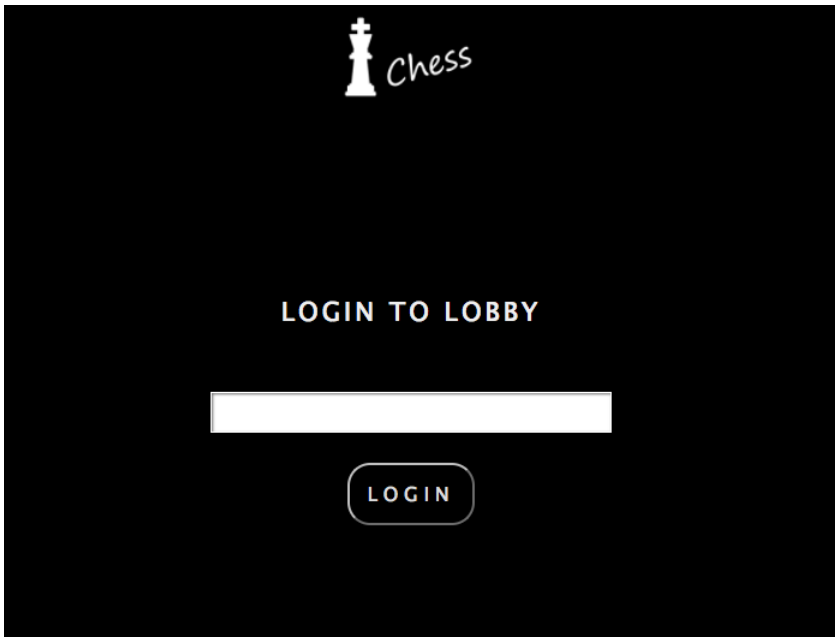
```
// otevrit konfiguracni soubor NGINXu
nano /usr/local/etc/nginx/nginx.conf

// vlozit nove instance - pro aplikaci a sokety
    server {
        listen 80;
        server_name chessplay.eu www.chessplay.eu;
        location / {
            proxy_pass http://192.168.4.1:3000;
            proxy_http_version 1.1;
        }
    }
    server {
        listen 80;
        server_name socket.chessplay.eu www.socket.chessplay.eu;
        location / {
            proxy_pass http://192.168.4.1:3001;
            proxy_http_version 1.1;
        }
    }

// restartovat nginx
service nginx restart
```

8.3 Průvodce aplikací

Tato kapitola je věnována výslednému vzhledu uživatelského rozhraní aplikace. Jako první při spuštění se zobrazí přihlašovací obrazovka (obr. 8.1), na které je možno vidět přihlašovací formulář a hlavičku obsahující logo, které se zobrazuje na všech stránkách aplikace.



Obrázek 8.1: Přihlašovací obrazovka

Na dalším obr. 8.2 je pak možno vidět *Lobby* s přihlášenými hráči, z nichž je následně jeden vyzván ke hře.

Před odesláním výzvy je nejprve nutné hru nastavit pomocí dialogového okna (obr. 8.3). Nastavit je možné dobu trvání hry a povolit/zakázat pomocný herní režimu (zvýraznění polí, na která lze táhnout)

Po odeslání výzvy se zobrazí zpráva oběma zúčastněným hráčům (obr. 8.4 a 8.7).

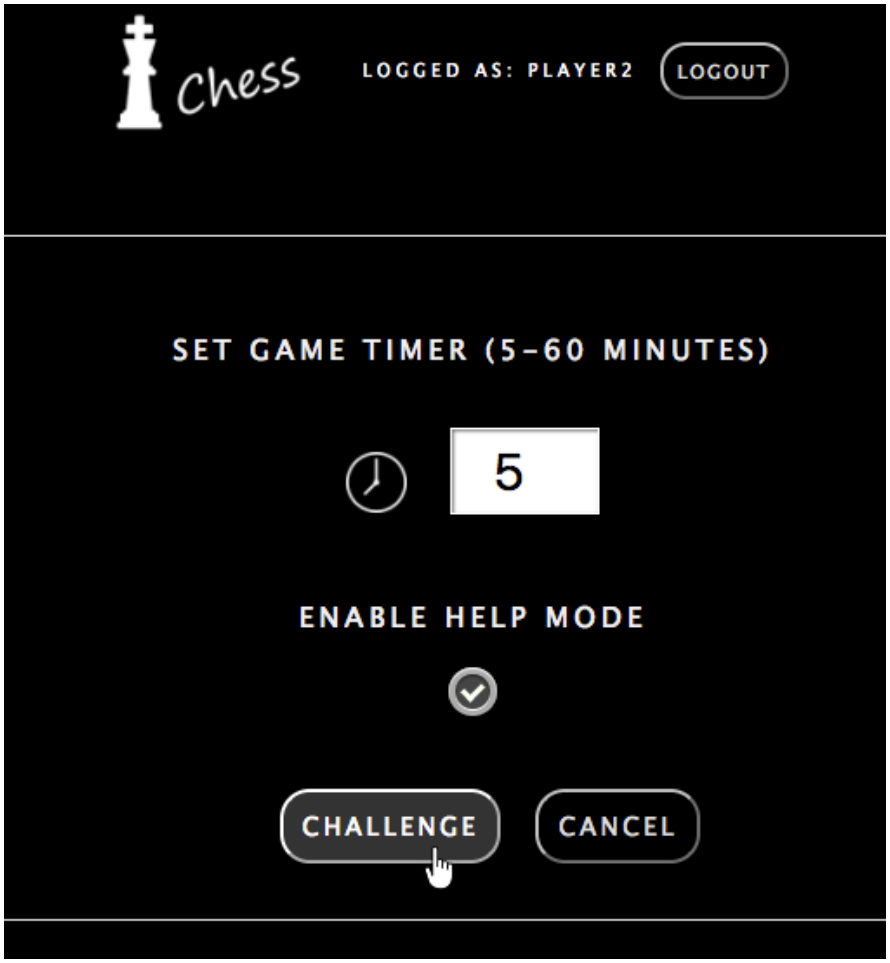
Na dalším snímku (obr. 8.9) se nachází již samotná hra z pohledu hráče, který čeká na protihráčův tah. Čekání je znázorněno ikonou přesípacích hodin na herní liště.

Naopak hráči, který je na tahu, je zobrazena ikona pěšce, symbolizující, že hráč může táhnout. V případě, že je hráč v šachu, také svítí červeně ikona krále, viz. obr. 8.6

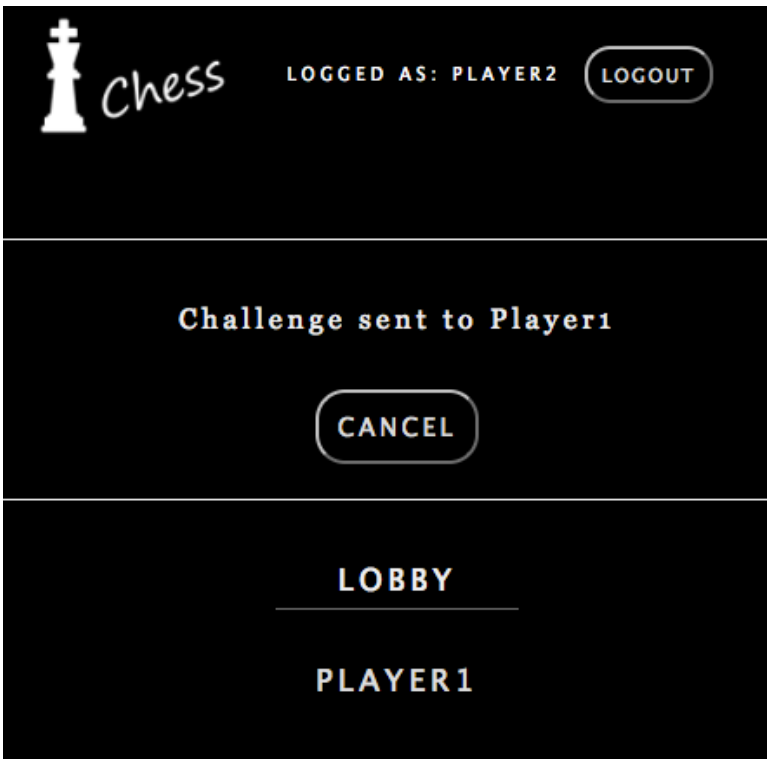
Jak vypadá zvýraznění polí v pomocném herním režimu, je možné vidět na obr. 8.8

Ve chvíli kdy hra končí, je hráčům zobrazeno dialogové okno informující o konci hry a jejím výsledku - výhra/prohra. Okno dále obsahuje herní statistiky. Výherní okno je přiloženo na obr. 8.9.

Obrázek 8.2: Seznam hráčů v lobby



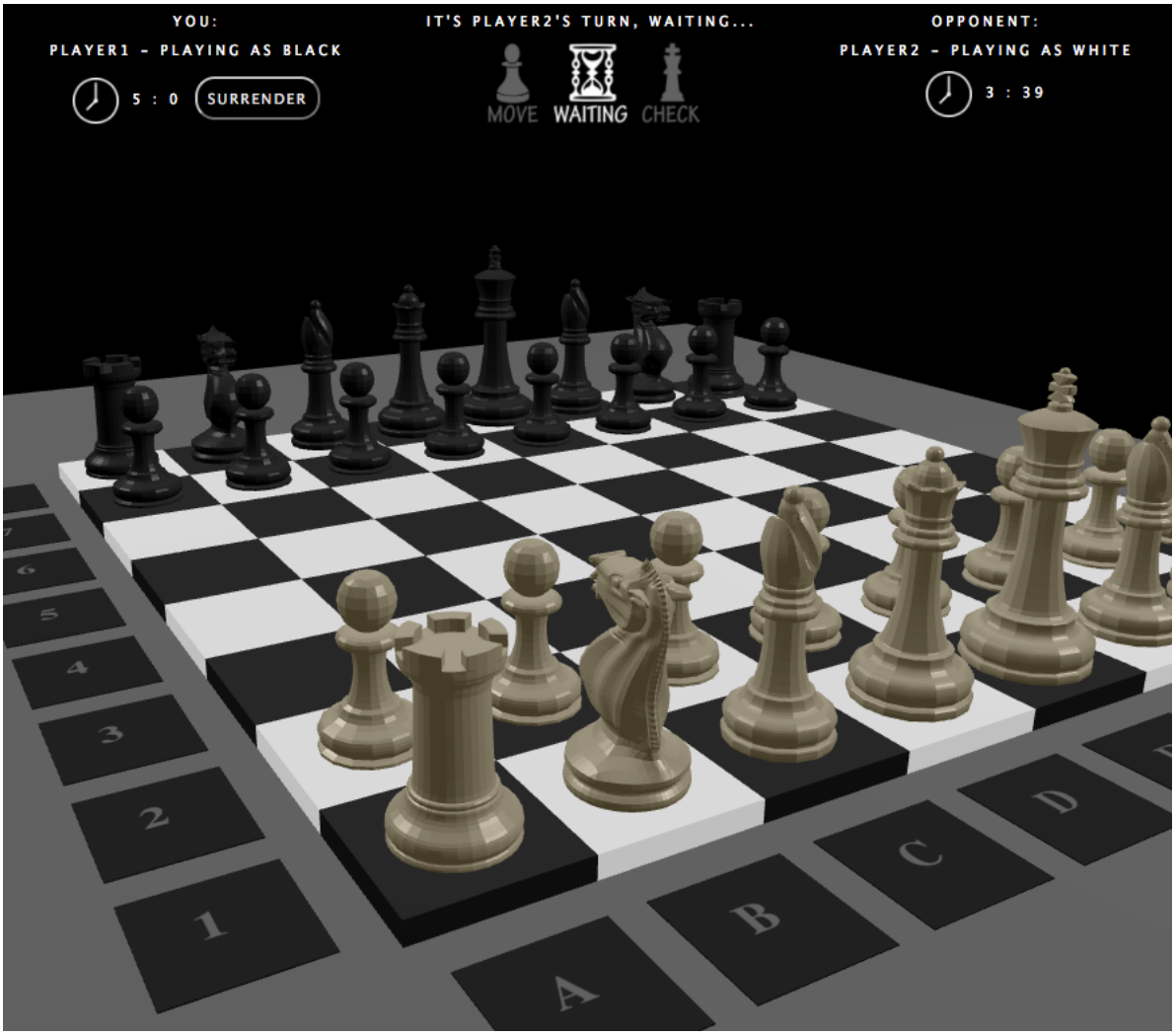
Obrázek 8.3: Dialogové okno s nastavením hry



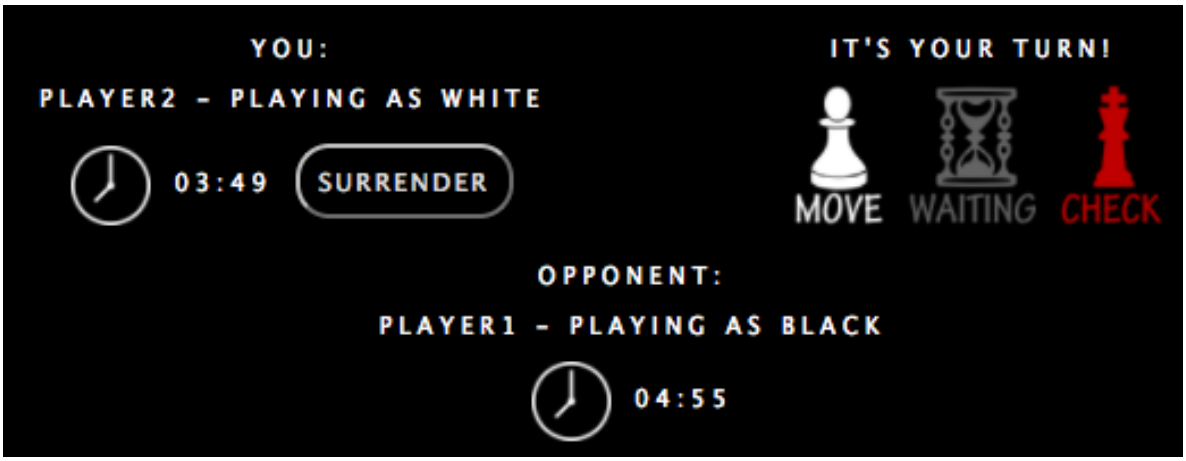
Obrázek 8.4: Odchozí výzva ke hře

8.4 Testování aplikace

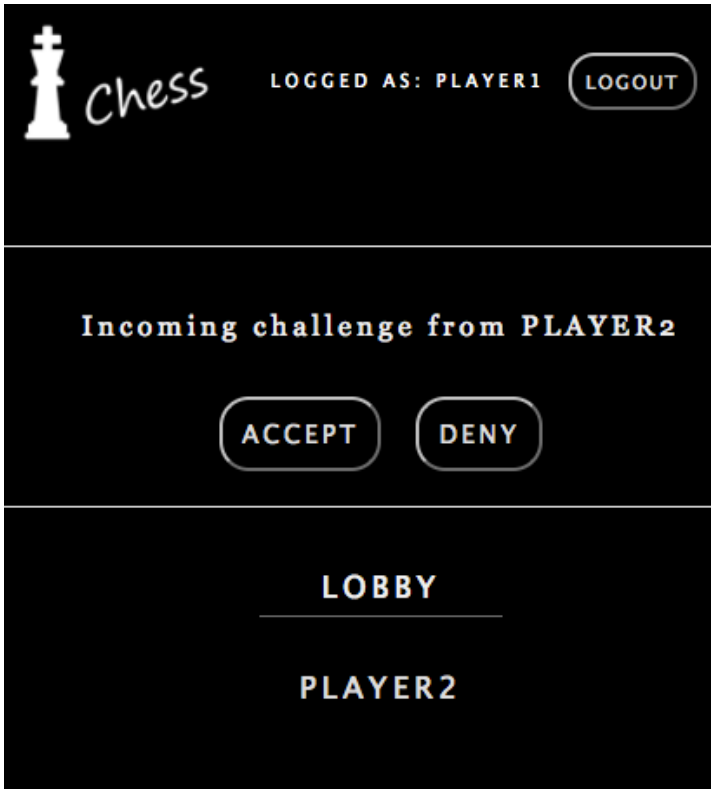
Funkčnost aplikace byla otestována na prohlížečích *Google Chrome*, *Mozilla Firefox* a *Safari*. Testování proběhlo na operačních systémech *Windows 7*, *macOS Sierra* a *Manjaro Linux*. Všechny testy dopadly úspěšně.



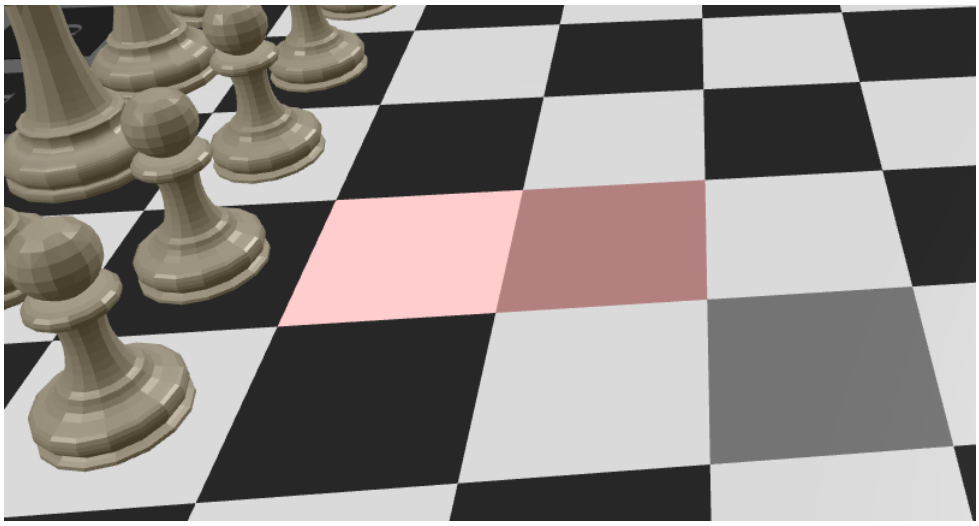
Obrázek 8.5: Hra z pohledu čekajícího hráče



Obrázek 8.6: Herní lišta hráče, který je na tahu a v šachu



Obrázek 8.7: Příchozí výzva ke hře



Obrázek 8.8: Zvýraznění polí v pomocném režimu

Obrázek 8.9: Okno se statistikami zobrazené vítězi

Závěr

V rámci diplomové práce se podařilo implementovat webovou aplikaci, 3D šachy pro dva a více hráčů, pracující v reálném čase. Hra obsahuje základní šachovou funkcionalitu. Implementace a následné testování aplikace dopadlo úspěšně. Testování funkčnosti aplikace proběhlo na několika operačních systémech a v různých prohlížečích. Tím se potvrdilo, že je aplikace multiplatformní. Cíl diplomové práce byl tedy splněn.

Na aplikaci plánuji dále pracovat a rozšířit ji o další funkce. Např. by bylo vhodné, aby si uživatelé mohli zvolit barevné schéma aplikace. Podle toho by se pak změnil vzhled jednotlivých stránek aplikace a 3D scény. Chtěl bych dále vybudovat portál, který bude navštěvovat jak tuzemská tak zahraniční komunita. Hráči by na tomto portálu měli možnost účastnit se různých turnajů, zápasit o lepší příčky v žebříčcích, porovnávat své výsledky ve statistikách atd.

Dále bych rád naprogramoval mobilní aplikaci, aby si tito hráči mohli pohodlně zahrát rychlou partičku např. na cestě do práce, v autobusu či metru.

Pro mě osobně je tato práce začátek profesní cesty, na které bych rád zdokonaloval návrh a implementaci podobných aplikací.

Seznam použité literatury

[1] CHALUPA, Ivan. *Historie šachu*. Praha: Lika klub, 2012. ISBN 978-80-86069-76-0.

[2] ISAACSON, Walter. *Steve Jobs*. Praha: Práh, 2011. ISBN 978-80-7252-352-8.

[3] HOROVÁ, Iva. *3D modelování a vizualizace v AutoCADu pro verze 2009, 2008 a 2007*. Brno: Computer Press, 2008. ISBN 978-80-251-2194-8.

[4] DERAKHSHANI, Dariush. *Maya: průvodce 3D grafikou*. Praha: Grada, 2006. Průvodce (Grada). ISBN 80-247-1253-9.

[5] CATLIN, Hampton. a Michael Lintorn. CATLIN. *Pragmatic guide to Sass*. Dallas, Tex.: Pragmatic Bookshelf, c2011. ISBN 978-1934356845.

[6] FLANAGAN, David. *JavaScript: the definitive guide*. 6th ed. Sebastopol, CA: O'Reilly, 2011. ISBN 978-0596805524.

[7] ZAKAS, Nicholas C. *Understanding ECMAScript 6: the definitive guide for JavaScript developers*. ISBN 978-1593277574.

[8] STEFANOV, Stoyan. *React : up & running: building web applications*. ISBN 978-1491931820.

[9] EISENMAN, Bonnie. *Learning React Native: building mobile applications with JavaScript*. ISBN 978-1-4919-2900-1.

[10] VEPSÄLÄINEN, Juho. *Webpack: From apprentice to master*. ISBN 978-9526868806.

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

[12] HAHN, Evan. *Express in action: writing, building, and testing Node.js applications*. ISBN 978-1617292422.

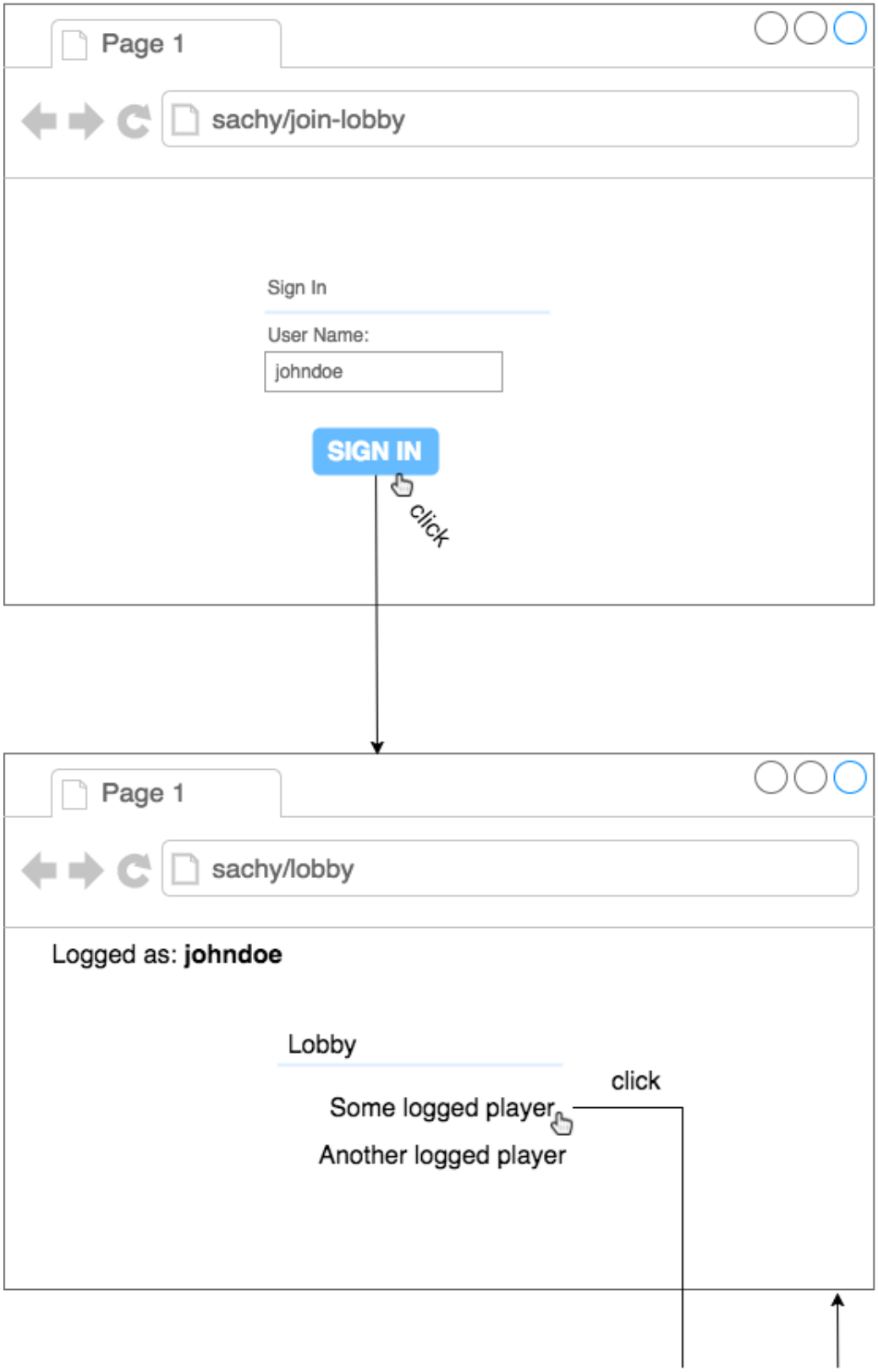
[13] MASSÉ, Mark. *REST API design rulebook*. Sebastopol, CA: O'Reilly, 2012. ISBN 9781449310509.

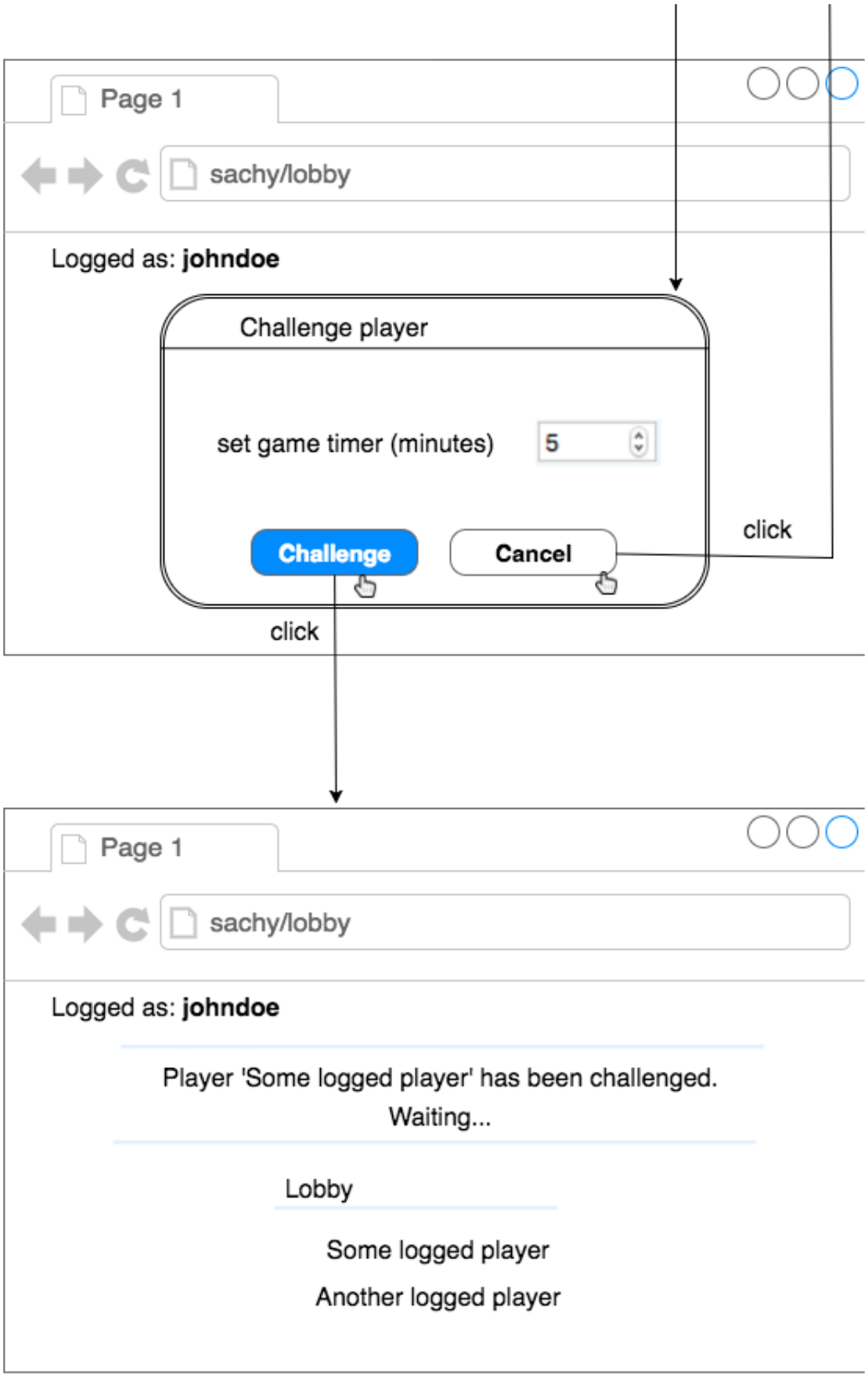
[14] CHODOROW, Kristina. *MongoDB: the definitive guide*. Second edition. ISBN 978-1-4493-4468-9.

[15] RAI, Rohit. *Socket. IO Real-Time Web Application Development*. New Edition. Birmingham: Packt Publishing, Limited, 2013. ISBN 9781782160786.

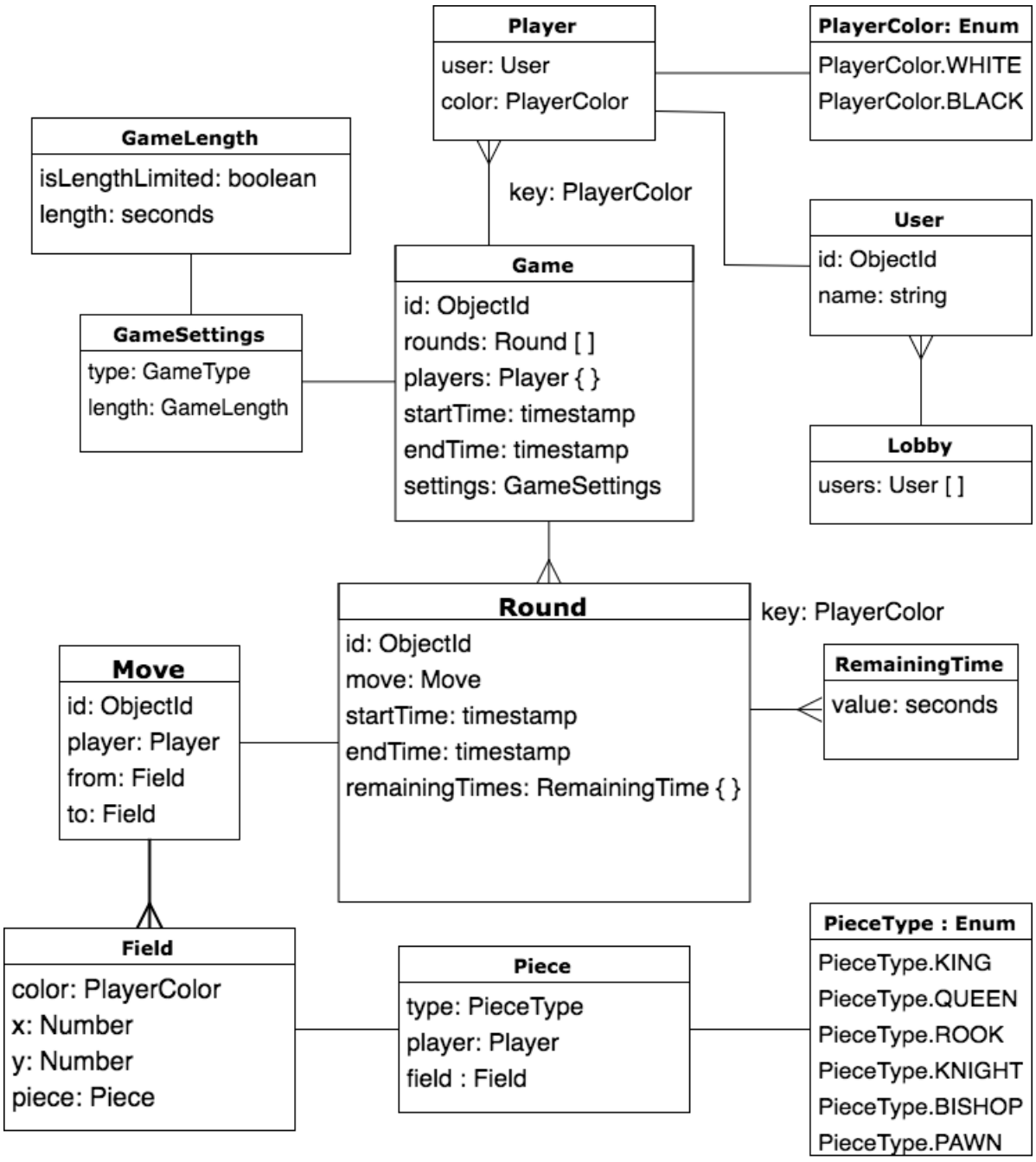
[16] LUCAS, Michael. *Síťový operační systém FreeBSD: podrobný průvodce*. Brno: Computer Press, 2003. ISBN 80-7226-795-7.

Příloha A - Ukázka drátěných modelů lobby




Page 1
sachy/lobby
Logged as: johndoe
Challenge player
set game timer (minutes)
5
Challenge
Cancel
click
click
Page 1
sachy/lobby
Logged as: johndoe
Player 'Some logged player' has been challenged.
Waiting...
Lobby
Some logged player
Another logged player

Příloha B - Logický model aplikace



Příloha C - Ukázka socketové komunikace

```
// klient: ChessBoard.js - metoda move(fieldTo)
if (!this.ini) {
    this.clearMoves()
    this.setActivePlayer(this.getOpponent())
    const gameId = Store.getInstance().getState().game.gameId
    var move = {
        gameId,
        fields: {
            from: this.activeField.id,
            to: fieldTo.id
        }
    }
    if (castlingMove) {
        move.castlingMove = {
            rookField: castlingMove.rookField.id,
            rookDestinationField: castlingMove.rookDestinationField.id
        }
    }
    Socket.emitMove(move)
}

// klient: configureSocket.js
emitMove(move){
    const data = {
        move,
        socketId
    }
    instance.emit('move', data);
}

// server: socketRouter.js
socket.on('move', function (moveData) {
    const gameId = moveData.move.gameId
    const playerSocketId = moveData.socketId
    let opponentSocketId = getOpponentSocketId(gameId, playerSocketId)
    if (isValidMove(gameId, moveData)) {
        io.to(opponentSocketId).emit('move', moveData.move);
    }
});

// klient: configureSocket.js
instance.on('move', function (move) {
    ChessBoard.ChangeTurn(move)
});
```

Příloha D - Dokumentace socketového API

```
// SERVER
{
  /**
  *   ON
  *   param socket
  */
  "connection": {
    "id": "2C5URXY-PEdgo4IqAAAA"
    //socketId
  },
  /**
  *   EMIT
  *   param response
  */
  "connectionResponse": {
    "onlinePlayers": [
      {
        "id": "507f191e810c19729de860ea", //ObjectId
        "name": "player1"
      }
    ],
    "socketId": "2C5URXY-PEdgo4IqAAAA"
  },
  /**
  *   ON
  *   param player
  **/
  "login": {
    "id": "507f191e810c19729de860eb", //ObjectId
    "name": "player2"
  },
  /**
  *   ON
  *   param player
  **/
  "logout": {
    "id": "507f191e810c19729de860eb", //ObjectId
    "name": "player2"
  },
  /**
  *   ON
  *   param players
  **/
  "challenge": {
    "challenger": {
      "id": "507f191e810c19729de860ea", //ObjectId
      "name": "player1"
```

```
    },
    "opponent": {
      "id": "507f191e810c19729de860eb", //ObjectId
      "name": "player2"
    }
  },
  /**
  *   EMIT
  *   param challenger
  **/
  "incomingChallenge": {
    "id": "507f191e810c19729de860ea", //ObjectId
    "name": "player1"
  },
  /**
  *   EMIT
  *   param opponent
  **/
  "sentChallenge": {
    "id": "507f191e810c19729de860eb", //ObjectId
    "name": "player2"
  },
  /**
  *   ON
  *   param players
  **/
  "challengeAccepted": {
    "challenger": {
      "id": "507f191e810c19729de860ea", //ObjectId
      "name": "player1"
    },
    "opponent": {
      "id": "507f191e810c19729de860eb", //ObjectId
      "name": "player2"
    }
  },
  /**
  *   ON
  *   param players
  **/
  "challengeCanceled": {
    "challenger": {
      "id": "507f191e810c19729de860ea", //ObjectId
      "name": "player1"
    },
    "opponent": {
      "id": "507f191e810c19729de860eb", //ObjectId
      "name": "player2"
    }
  },
```

```
  /**
  *   ON
  *   param move
  **/
  "move": {
    "gameId": "607f191e810c19729de860ff",
    "round": 1,
    "timestamps": {
      "roundStart": "2017-01-02T00:00:00.00+01:00",
      "roundEnd": "2017-01-02T00:00:05.00+01:00"
    },
    "movingPlayer": {
      "id": "507f191e810c19729de860ea", //ObjectId
      "name": "player1",
      "color": "white",
      "timeRemaining": 295 //seconds
    },
    "opponent": {
      "id": "507f191e810c19729de860eb", //ObjectId
      "name": "player2",
      "color": "black",
      "timeRemaining": 300 //seconds
    },
    "fields": {
      "from": "e2",
      "to": "e4"
    }
  }
}
```

Příloha E - Ukázka vytvoření šachového pole

```
var getOverColor = function (emissiveColor) {
    if (emissiveColor.r === 0)
        return new BABYLON.Color3(0.3, 0.3, 0.3);
    else
        return new BABYLON.Color3(0.5, 0.5, 0.5)
}

// Over/Out
var makeOverOut = function (mesh) {
    mesh.actionManager = new BABYLON.ActionManager(scene);
    mesh.actionManager.registerAction(new BABYLON.SetValueAction(
       BABYLON.ActionManager.OnPointerOutTrigger,
        mesh.material, "emissiveColor", mesh.material.emissiveColor));
    mesh.actionManager.registerAction(new BABYLON.SetValueAction(
       BABYLON.ActionManager.OnPointerOverTrigger,
        mesh.material, "emissiveColor", getOverColor(mesh.material.
           emissiveColor)));
}

var initField = function (x, y) {
    const color = isBlack ? BABYLON.Color3.Black() : new BABYLON.Color3
       (0.7, 0.7, 0.7);
    let field = BABYLON.Mesh.CreateBox("field" + x + "" + y, 70, scene)
       ;
    let fieldMaterial = new BABYLON.StandardMaterial("ground", scene);
    fieldMaterial.diffuseColor = new BABYLON.Color3(0.4, 0.4, 0.4);
    fieldMaterial.specularColor = new BABYLON.Color3(0.4, 0.4, 0.4);
    fieldMaterial.emissiveColor = color;
    field.material = fieldMaterial;
    field.position.x -= x * 70 - 300;
    field.position.z -= y * 70 - 300;
    field.position.y -= 25;
    makeOverOut(field);
    fields[x + "" + y] = new Field(field, x + "" + y, isBlack);
}
```

Příloha F - Použité javascriptové moduly

```
{ // SERVER
    "dependencies": {
        "async": "^2.5.0",
        "bluebird": "^3.5.0",
        "body-parser": "^1.17.2",
        "express": "^4.15.3",
        "mongodb": "^2.2.26",
        "socket.io": "^2.0.1",
        "strftime": "^0.10.0",
        "winston": "^2.3.1"
    },
    "devDependencies": {
        "babel-cli": "^6.9.0",
        "babel-core": "^6.9.0",
        "babel-preset-es2015": "^6.9.0",
        "babel-preset-stage-0": "^6.5.0",
        "eslint": "^3.1.1",
        "nodemon": "^1.9.2"
    }
}
{ // KLIENT
    "dependencies": {
        "cookies-js": "^1.2.3",
        "socket.io-client": "^2.0.3"
    },
    "devDependencies": {
        "babel-loader": "^7.0.0",
        "babel-plugin-react-html-attrs": "^2.0.0",
        "babel-plugin-transform-class-properties": "^6.24.1",
        "babel-plugin-transform-decorators-legacy": "^1.3.4",
        "babel-preset-es2015": "^6.24.1",
        "babel-preset-react": "^6.24.1",
        "babel-preset-react-hmre": "^1.1.1",
        "babel-preset-stage-0": "^6.24.1",
        "babili-webpack-plugin": "0.0.11",
        "react": "^15.5.4",
        "react-dom": "^15.5.4",
        "react-redux": "^5.0.5",
        "react-router-dom": "^4.1.1",
        "react-router-redux": "^4.0.8",
        "redux": "^3.6.0",
        "redux-logger": "^3.0.6",
        "redux-thunk": "^2.2.0",
        "superagent": "^3.5.2",
        "webpack": "^3.0.0"
    }
}
```