MASARYKOVA
UNIVERZITA

FAKULTA INFORMATIKY

Webová aplikace pro evidenci rezervací

Bakalářská práce

RADIM STEJSKAL

Brno, podzim 2023

MASARYKOVA
UNIVERZITA

FAKULTA INFORMATIKY

Webová aplikace pro evidenci rezervací

Bakalářská práce

RADIM STEJSKAL

Vedoucí práce: Mgr. Luděk Bártek, Ph.D.

Brno, podzim 2023


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Prohlášení

Prohlašuji, že tato bakalářská práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Radim Stejskal

Vedoucí práce: Mgr. Luděk Bártek, Ph.D.

Poděkování

These are the acknowledgements for my thesis, which can span multiple paragraphs.

Shrnutí

This is the abstract of my thesis, which can
span multiple paragraphs.

Klíčová slova

keyword1, keyword2, ...

Obsah

Seznam tabulek

Seznam obrázků

Úvod

První počítačové rezervační systémy byly navrženy a spravovány převážně aeroliniemi, avšak na jejich základech byly postupně vybudovány celé globální distribuční systémy, hojně využívané v odvětví cestovního ruchu [1]. Dnes již rezervace svých komodit spravuje online celá řada podniků a institucí.

Aktuální webové rezervační systémy bychom mohli rozdělit do několika skupin podle počtu možných souběžných rezervací. První skupina umožňuje takových rezervací více a zahrnuje systémy vhodné například pro restaurace, kina či hotely, kde je zpravidla maximální počet souběžných rezervací omezen počtem stolů, míst k sezení či množstvím dostupných pokojů. Další skupinou jsou systémy schopné jediné rezervace v daný časový úsek a své využití nachází při sjednávání schůzek na úřadech či v lékařských zařízeních. Poslední a zároveň nejobecnější skupina kombinuje obě předešlé strategie a jejími zástupci jsou obzvláště systémy pro sportovní střediska, která mohou zároveň nabízet jak skupinové, tak individuální lekce či pronajmutí jednotlivých sportovišť.

Pro účel této práce jsem se rozhodl věnovat implementaci rezervačního systému vhodného primárně právě pro sportovní střediska. Motivací mi je především zkušenost se systémy obdobného typu, které působí buď zastarale či postrádají některou užitečnou funkci.

1 Analýza

Výsledná aplikace není určena pro konkrétního zákazníka, který by vznášel požadavky na její provedení, analýza tudíž vychází převážně z existujících rezervačních systémů a odhalování jejich nedostatků, případně užitečných funkcí, které systémy vzájemně postrádají. Získané poznatky budou sloužit jako inspirace pro požadavky na finální aplikaci.

1.1 Dostupná řešení

Pro každou z následujících aplikací je zformulován krátký přehled a následně jsou popsány její základní a případné zpoplatněné funkce. Na závěr jsou zdůrazněny unikátní vlastnosti zkoumaných systémů a zhodnoceny jejich pozitiva či negativa.

1.1.1 Reservanto

Jedná se o pravděpodobně nejrozšířenější rezervační systém na českém trhu. Je dostupný v několika variantách, a sice: *basic*, *lite*, *professional* a *premium*. Navzájem se jednotlivé varianty liší různými dovednostmi – od zasílání upomínek zákazníkům na blížící se rezervace přes zobrazování statistik až po online platby a správu jednotlivých uživatelských účtů. Rozdíly mezi krajními variantami *basic* a *premium* jsou propastné, jelikož *basic* z uvedených bonusových funkcí nenabízí žádnou a varianta *premium* již téměř připomíná komplexní systém pro plánování podnikových zdrojů (ERP systém).[2] Aplikace se snaží přizpůsobit povaze zákazníkova podniku pomocí možnosti výběru z několika zaměření. Každé zaměření pak určuje své rezervační zdroje – například v případě zaměření kurty je zdrojem jeden fyzický kurt. S plánem *premium* má uživatel možnost zaměření kombinovat, avšak všechny příslušné rezervační zdroje podléhají jedné možné rezervaci v danou dobu.

K dispozici jsou také obecná zaměření:

- jeden zákazník v libovolný čas,
- skupina zákazníků v pevný čas,
- události s více termíny,

avšak pro nejnižší variantu *basic* jsou prakticky nevyužitelná z důvodu omezeného počtu možných událostí v jednom týdnu.

Varianta *basic* je jako jediná dostupná zdarma, *lite* lze momentálně zakoupit za cenu 199 Kč za měsíc a každou vyšší variantu lze získat za přibližně trojnásobnou cenu varianty předchozí.

Reservanto nabízí poměrně vysokou flexibilitu a řadu funkcí, které přesahují rámec běžného rezervačního systému, příkladem je možnost sledování statistik rezervací a kompletního nastavení uživatelských oprávnění. Velkým přínosem je vlastní API (Application Programming Interface), které usnadní propojení systému Reservanto s dalšími podnikovými systémy.

Varianta *basic* je vhodná pouze pro velmi specifické případy a snaží se uživatele přimět k zakoupení placené verze pomocí všudypřítomných omezení, například nabízené API je tak dostupné až od nejvyšší verze *premium*. Funkci, která by prostřednictvím emailu informuje zákazníky o stavu rezervace, taktéž nabízí pouze placená verze. To samé platí i pro funkci správy uživatelů, která je dostupná až od nejdražší verze.

1.1.2 Booked4us

Booked4us je rezervační systém vyvinutý maďarskou společností se sídlem v Budapešti. Momentálně je dostupný ve třech jazycích, a to v nativní maďarštině, němčině a angličtině. Při počáteční konfiguraci může uživatel vybírat z pěti kategorií podniků, nebo lze ručně zadat svou vlastní. Pro samostatnou funkcionalitu je ale podstatná následná volba jedné ze dvou charakteristik kalendáře, a to buď kalendář volných rezervací časových úseků v rámci otevírací doby nebo předem plánovaných událostí. Obě varianty umožňují nastavení libovolné kapacity časových úseků či událostí. Každý z kalendářů tak lze dedikovat různým fyzickým prostorám, například tělocvičnám, a spravovat jejich rezervace odděleně. Uživatel však není limitován na jeden kalendář a může si vytvořit libovolný počet kalendářů o různých charakteristikách.[3]

Jelikož Booked4us nabízí pouze svou základní variantu, je cena určena výhradně délkou předplatného, které momentálně vychází na zhruba 12 € měsíčně. Uživatel si však zvlášť může připlatit za zasílání upomínacích zpráv zákazníkům pomocí SMS, kdy 100 zpráv měsíčně

vyjde na necelých 15 €. Dále si provozovatel systému účtuje téměř 30 € za každou hodinu poskytnuté technické podpory.

Systém působí z uživatelského hlediska velmi přehledně, čemuž pomáhá i možnost definice více než jednoho kalendáře. Zároveň možnost volby ze dvou charakteristik pro každý kalendář je užitečným nástrojem v situaci, kdy sportovní zařízení nabízí prostory s volným přístupem v otevírací době, jakými mohou být tenisové kurty, a zároveň pořádá skupinové lekce.

Aplikace nedovoluje jednotlivé události editovat či rušit přímo z kalendáře a správce je nucen provést jejich editaci přímo v záložce událostí. Dále po vytvoření opakující se události není možná správa jednotlivých výskytů události. Osobně mi přijde praktičtější řešení po vzoru kalendáře v aplikaci Outlook, kde editace či smazání události, jež je součástí série opakujících se událostí, nabídne volbu editace či smazání daného výskytu či celé série.

1.1.3 Reservio

Z přezkoumaných rezervačních systémů jednoznačně vyniká Reservio díky své uživatelské přívětivosti. Tento český produkt je dostupný v mnoha světových jazycích a nabízí uživatelům kompromis mezi funkcionalitou a komplexitou systému, uživatelské rozhraní je tedy velmi intuitivní. Systém funguje na principu vytváření událostí o dané kapacitě, na které se mohou uživatelé přihlásit.

Reservio je dostupný ve čtyřech variantách: *free*, *starter*, *standard* a *pro*. S každou vyšší variantou roste cena přibližně o 200 Kč za měsíc. Nejlevnější varianta *free* slouží spíše jako zkušební verze, jelikož má omezení na 40 současných rezervací a neumožňuje integraci přes API, která je dostupná až od nejdražší verze.

Události v Reservio mohou být jak jednorázové, tak opakující se, přičemž u opakovaných událostí lze manipulovat s jednotlivými výskyty. Každá událost může být přiřazena ke službě, na kterou je událost vázaná.[4] Toto řešení je částečně alternativou k možnosti definice více kalendářů, jakou nabízí systém Booked4us. Reservio však nemusí být optimální volbou pro sportovní střediska, kde může být často nutné události oddělovat na základě místa konání, například na základě určitého sálu. Při větším množství překrývajících se událostí se pak jediný kalendář může stát nepřehledným.

1.2 Shrnutí

Zkoumaná řešení nabízejí rozsáhlou škálu funkcí a mají široké uplatnění. Lze však konstatovat, že podstatným problémem všech z nich je pořizovací cena, respektive velmi omezené využití bezplatných verzí systémů. Zákazníci jsou často nuceni dokupovat vyšší verze pouze kvůli nízkému počtu možných pořádaných událostí a rezervací, jak je tomu například u systému Reservio. Pokročilejší varianty systémů jsou pak často až příliš komplexní kvůli dodatečným funkcím, které pro takové zákazníky nemusí být relevantní.

Co se samotných funkcí týče, možnost definice více kalendářů je pro podniky, jako jsou sportovní střediska, důležitou vlastností, avšak je spíše raritou mezi zkoumanými systémy. Samozřejmostí není ani přítomnost API pro snazší integraci s dalšími podnikovými systémy, nebo je dostupné až v nejdražších variantách. Největším kamenem úrazu pro většinu zkoumaných systémů je však absence podpory cenových hladin jednotlivých rezervací. Tato funkce je důležitá zejména kvůli aktuální popularitě karty *Multisport*, díky které má zákazník jednou denně událost buď kompletně zdarma, nebo může čerpat alespoň částečnou slevu. Ze zkoumaných systémů s takovým scénářem počítá jako jediný systém Resrvanto a to pouze u svých pokročilých variant.

Na základě těchto poznatků je v následující části navržen open source rezervační systém PlaceHolder.

1.3 Požadavky

Při vývoji software je důležité mít dobře specifikované požadavky na finální aplikaci. Za tímto účelem je často vytvářen dokument CONOPS (Concept of Operations) popisující vysokoúrovňový koncept a záměr vyvíjeného projektu z uživatelského pohledu, a dokument SRS (Sofware Requirements Specification), který popisuje požadavky na software v podrobnějších detailech.

Pro potřeby této práce se však zaměřím pouze na funkční a nefunkční požadavky vyvíjené aplikace PlaceHolder.

1.3.1 Funkční požadavky

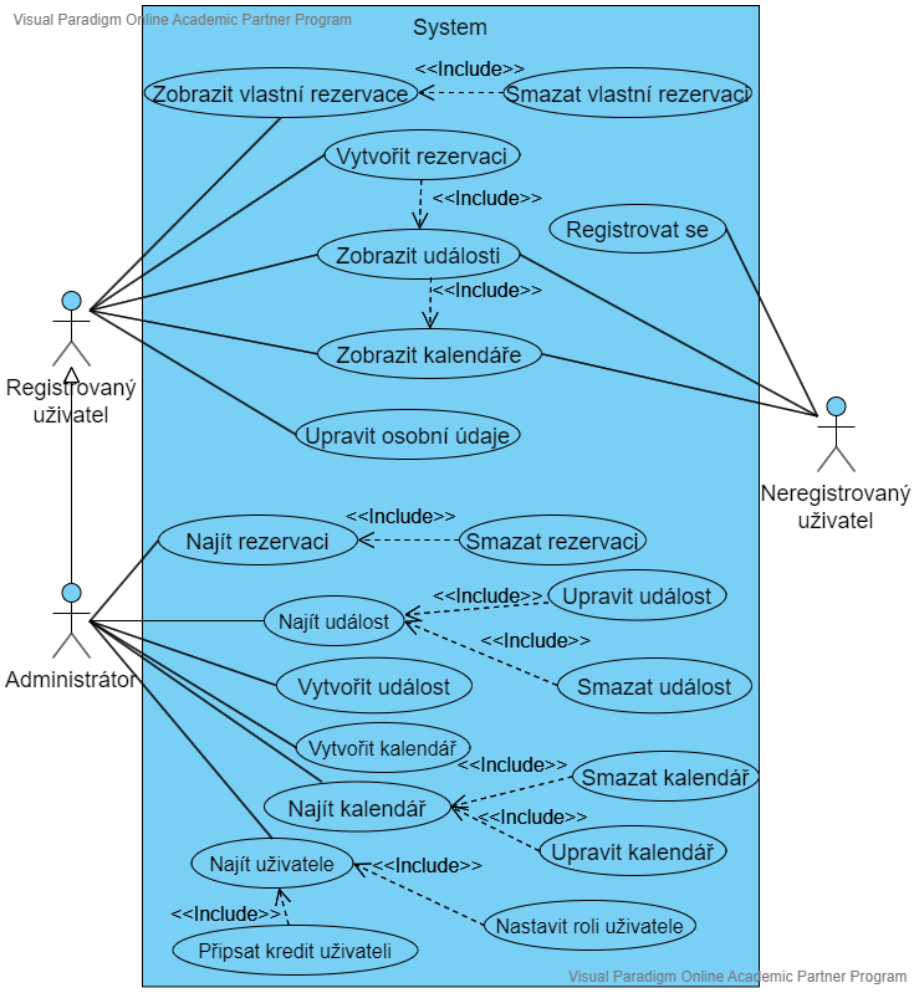
Cílem funkčních požadavků je specifikovat chování systému a srozumitelně definovat jeho jednotlivé funkce. Na základě jednotlivých popisů funkcí by pak vývojář měl být schopen funkce implementovat a ověřit jejich správnost. [5]

Výčet funkčních požadavků pro aplikaci PlaceHolder tedy vypadá následovně:

- Registrace nových uživatelů: Povinnými údaji pro registraci budou: jméno, příjmení, emailová adresa, heslo a údaj o vlastnictví karty Multisport. Emailová adresa bude v systému fungovat jako unikátní uživatelské jméno. Nový účet bude potřeba aktivovat pomocí zaslaného emailu o potvrzení registrace. Nový uživatel bude mít automaticky přidělenou roli *user*.
- Obnova hesla: V případě, že uživatel zapomene přihlašovací heslo, bude si moci vygenerovat heslo nové, které bude zasláno na emailovou adresu uživatele. Nové heslo bude nastaveno až pomocí odkazu v zaslaném emailu.
- Cenové hladiny a kreditový systém: Každá událost v systému bude mít dvě cenové hladiny, a to jednu základní a jednu pro uživatele splňující aktuální podmínky pro uplatnění karty *Multisport*. V systému bude figurovat virtuální měna – *kredity*, která bude uživateli manuálně připisována.
- Definice a správa lokací: Lokace budou předdefinovány jako číselník. K jednotlivým lokacím bude možné vytvořit jeden nebo více kalendářů, ke kalendářům pak budou náležet jednotlivé události. Odstranit kalendář bude možné pouze v případě, že nebude obsahovat žádné naplánované budoucí události.
- Uživatelské role a oprávnění: V systému budou existovat tři uživatelské role, a to: *guest*, *user* a *admin*. *Guest* jako nepřihlášený uživatel si bude moci pouze zobrazovat plánované události. *User* se bude moci přihlašovat na vypsané události, zobrazovat a rušit své vlastní budoucí rezervace a měnit své údaje (heslo, vlastnictví karty *Multisport*). *Admin* bude mít oprávnění k vytváření a editaci (názvu a popisu) budoucích událostí. Mazat bude možné pouze takové budoucí události, na které aktuálně neexistují žádné rezervace. Za tímto účelem bude smět *admin* zobrazovat všechny rezervace a budoucí rezervace rušit. Stejně

tak smí *admin* spravovat kalendáře dle zmíněných pravidel. *Admin* může taktéž nastavit uživatelům jednu z rolí *admin* či *user* a nastavovat nezáporný stav *kreditů* jednotlivých uživatelů.
- Vytváření a validace událostí: Každá nová událost má povinné údaje, kterými jsou název, začátek, konec, místo konání, kapacita události a cena pro jednotlivé cenové hladiny. Událost může být jak jednorázová, tak opakující se každý týden ve specifikované dny. Nová událost se nesmí časově krýt s existujícími událostmi, události však na sebe mohou přímo navazovat. Při pokusu o vytvoření série událostí, ve které by se některá událost kryla s již existující událostí, nebude série vytvořena vůbec.
- Správa jednotlivých událostí: Aplikace umožní editovat události. Bez omezení bude možné nastavit nový název a popis události.
- Správa periodických událostí: Události, které jsou součástí série událostí bude možné editovat buď jednotlivě nebo jako množinu budoucích událostí z celé série.
- Vytvoření a správa rezervací: Každý přihlášený uživatel má možnost vytvářet rezervace na vypsané události začínající v budoucnosti, pokud kapacita událost již není plně obsazena.
- Zajištění, že žádná rezervace není duplicitní: Aplikace by měla zabránit vytváření duplicitních rezervací a uživatelům oznámit, že na událost je již přihlášen.
- Správa vlastního účtu: Uživatel bude mít možnost měnit vlastní uživatelské údaje, a to: jméno, příjmení a heslo.
- Notifikace: Každý uživatel po vytvoření rezervace na událost obdrží potvrzovací email s detaily události. Pokud se svou rezervaci rozhodne později zrušit, je uživateli zaslán taktéž email o zrušení rezervace.
- Nasazení aplikace: Aplikace bude připravena pro spuštění pomocí platformy Docker a Podman.

Popis funkčních požadavků je dále znázorněn na obrázku 1.1 pomocí diagramu případů užití.



Obrázek 1.1: Diagram případů užití

1.3.2 Nefunkční požadavky

Nefunkční požadavky se narozdíl od funkčních požadavků nezaměřují na konkrétní chování systému, ale zabývají se spíše oblastmi jako je výkon, použitelnost, spolehlivost, zabezpečení, udržitelnost a další.[5] Jelikož aplikace nebude nasazena do ostrého provozu, je obtížné tyto požadavky přesně definovat, přesto se v následující části pokusím definovat alespoň některé z nich.

- Použitelnost: Grafické rozhraní aplikace by mělo být pro uživatele intuitivní a podporovat zobrazení na mobilních zařízeních, tabletech a běžných monitorech.
- Dokumentace: Aplikace bude mít vlastní dokumentaci a dokumentaci pro API.
- Flexibilita: Systém se bude skládat z oddělené serverové a klientská aplikace tak, aby bylo možné klientskou aplikaci kompletně nahradit.
- Zabezpečení: Aplikace bude pro autentizaci a autorizaci používat JSON Web token (JWT), které budou obsahovat roli uživatele. Na základě uživatelské role bude řízen přístup ke koncovým bodům API.

1.4 Metodika vývoje

Metodik pro vývoj software existuje hned několik a jednou z nejstarších, i když stále aktuální, je takzvaný vodopádový model(*waterfall*). *Waterfall* je jednou z tradičních metodik, avšak pro její uplatnění je potřeba znát dopředu přesné zadání projektu a zároveň klade důraz na podrobnou dokumentaci. Samotný vývoj se skládá z několika fází, které jsou vykonávány kaskádovitě za sebou. Před započetím každé další fáze je nutné její pečlivé naplánování a úspěšné dokončení fáze předchozí.

Jako první popsal *waterfall* Dr. Winston W. Royce a není divu, že o něm ve své publikaci [6] píše jako o velmi riskantní strategii. Tuto domněnku odůvodňuje skutečností, že testování produktu, které má ověřit jeho správnou funkcionalitu, se provádí až téměř na samotném konci vývojového cyklu. Nutné úpravy, které z výsledků testů plynou, pak mohou být natolik zásadní, že je ohrožena buď konzistence požadavků na software, nebo je nutné provést změny v designu. Druhá varianta tak může mít za následek výrazný nárůst času potřebného k dodání produktu a tím i jeho ceny.

Na druhé straně proti tradičním metodám, jakou je kromě *waterfall* také například spirálový model, stojí metody agilní. Nejpopulárnější z nich je v současnosti SCRUM, avšak ten se v jednočlenném týmu aplikuje poměrně obtížně, převážně kvůli své potřebě přidělení rolí členům týmu. Samotným Manifestem pro Agilní vývoj software [7]

se však může inspirovat i individuální vývojář, konkrétně jeho dvěma následujícími body:

- funkční software před vyčerpávající dokumentací,
- reakce na změnu před rigidním dodržováním plánu

a následně si pro vlastní potřeby některou z agilních metod adaptovat.

Pro účel této bakalářské práce jsem se tedy rozhodl držet se agilní vývojové metodiky zvané *kanban*, jelikož tato metodika může být bez problémů využita i jednočlenným týmem. Původně byl *kanban* vyvinut inženýry z automobilky Toyota a měl za cíl navýšit efektivitu výrobních linek. Tuto metodiku lze však ze své podstaty využít pro vizualizaci a zefektivnění téměř jakéhokoliv pracovního procesu a proto se také stal populárním i v oblasti vývoje software [8].

Základem metody je vizuální model zprostředkovaný pomocí fyzické či virtuální tabule, která je rozdělena do několika sloupců. Sloupce označují jednotlivé fáze vývojového procesu a jednotlivé úkoly, které spolu tvoří zadání pro vyvíjený software, jsou popsány na lístečcích. Členové týmu aktualizují tabuli přesunem lístečků do sloupců, které korespondují s aktuálním stavem úkolů. Pro zachování efektivity a plynulosti pracovního procesu je však klíčové stanovit určitá omezení, jako je maximální počet úkolů, které se současně mohou nacházet ve fázi vývoje, a tím předejít hromadění nedokončené práce [9].

Pro vlastní účely jsem si definoval čtyři následující sloupce: *to-do*, *implementing*, *testing* a *done*. Maximální počet rozpracovaných úkolů je pak nastaven na tři, což se však odvíjí od zvolené granularity jednotlivých úkolů.

2 Návrh aplikace

Na základě stanovených požadavků pro výsledný rezervační systém v této kapitole navrhnu architekturu projektu z nejvyšší vrstvy abstrakce a následně nastíním design jednotlivých komponent.

2.1 Architektura

Definice toho, co je to vlastně architektura systému může být poměrně nejednoznačná. Pro někoho může představovat provázanost jednotlivých komponent na nejvyšší úrovni. Jiný pohled je takový, že architektura systému se zabývá rozhodnutími o návrhu, která je potřeba vyřešit v ranné fázi projektu. Kvalitní architektura projektu je však každopádně základem pro jeho další co možná nejsnazší rozšiřitelnost či úpravy.[10]

Nadále se tedy budu držet zavedených architektonických vzorů a uvedu ty, které jsou využity při vývoji systému PlaceHolder.

2.1.1 Vzor klient-server

V tomto architektonickém vzoru jsou základem dvě hlavní komponenty. Jsoum jimi klient, který je žadatelem o službu, a server, který služby poskytuje. Tyto dvě komponenty mohou být umístěny v rámci jednoho systému nebo komunikovat přes síť na odděleném hardwaru.

Vzor klient-server přináší řadu výhod. Jednou z hlavních je oddělení zodpovědností mezi klientem a serverem. Klient se stará o zobrazení dat a interakci s uživatelem, zatímco server poskytuje datovou logiku a zajišťuje přístup k databázi. Další výhodou je škálovatelnost, jelikož serverových částí aplikace může existovat více a mohou být rozprostřeny na různých fyzických serverech s vlastní výpočetní kapacitou, což umožňuje aplikaci lépe zvládat zátěž. V neposlední řadě je flexibilita. Klient a server může být naprogramován v různých programovacích jazycích a v případě potřeby je možné klientskou část systému kompletně nahradit jinou. V tomto případě je však nutné dodržet stanovený komunikační protokol mezi klientskou a serverovou aplikací [11].

Komunikačním protokolem pro celý systém bude HTTP a klientská aplikace bude se serverovou aplikací komunikovat pomocí vystaveného API. K podrobnějšímu popisu fungování komunikačního protokolu se dostaneme v kapitole Implementace.

2.1.2 Vrstevnatý architektonický vzor

Na základě stanovení architektury klient-server můžeme uvažovat i samostatnou architekturu pro serverovou aplikaci. Zvoleným architektonickým vzorem bude právě vrstevnatý návrhový vzor (*Layered pattern*), který rozděluje aplikaci na oddělené vrstvy s různými funkcemi. Každá vrstva poskytuje své služby a komunikuje pouze s vrstvami, které se nacházejí přímo nad a pod ní [11]. Počet vrstev přímo definice vrstevnaté architektury nespecifikuje, pro potřeby vyvíjené aplikace budou však stačit následující tři:

- Datová vrstva (Data layer) - Datová vrstva je nejnižší vrstvou, stará se o přístup k datům a komunikaci s databází. Na této úrovni mohou být umístěny DAO (Data Access Objects) pro komunikaci s databází.
- Aplikační vrstva (Application layer) - Tato vrstva poskytuje logiku aplikace a zpracovává požadavky z prezentační vrstvy. Patří sem tak právě komponenty pro logické zpracování požadavků a správu business pravidel, často nazývané jako *services*.
- Prezentační vrstva (Presentation layer) - Jedná se z pohledu abstrakce o nejvyšší vrstvu a se stará o komunikaci s uživatelským rozhraním, v našem případě se bude starat o vyřizování požadavků přicházejících z klientské aplikace.

Každá vrstva má jasně definovanou roli a odpovědnost, což umožňuje lepší modulárnost a údržbu kódu. Tato struktura zároveň umožňuje snadnější testování aplikace, protože každá vrstva může být testována samostatně.

2.2 Design

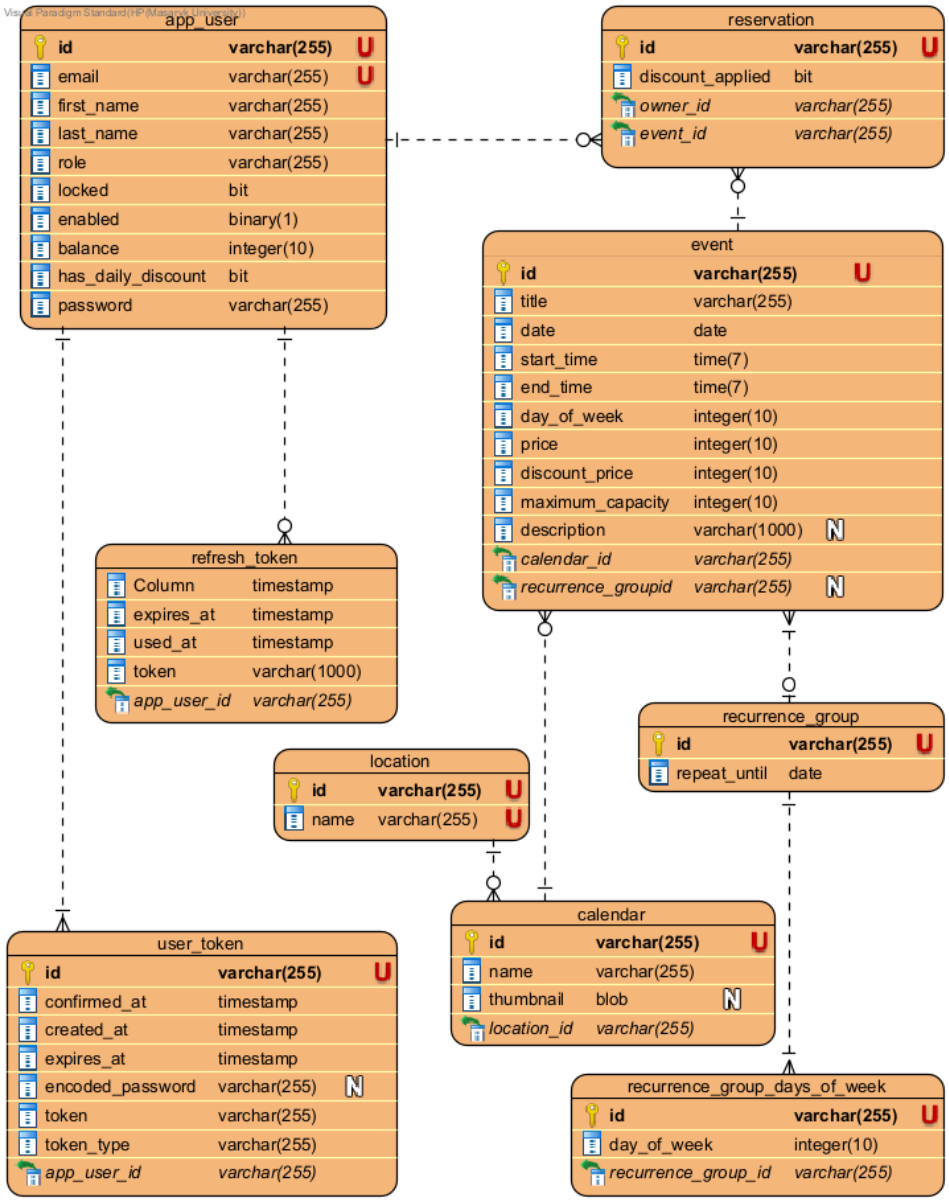
V této kapitole se zaměřím na návrh serverové části aplikace (backend) a na návrh komunikačního rozhraní. V tomto případě budu postupovat od nejnižší vrstvy, což znamená, že jako první si definuji

entity, které budou ve finální aplikaci figurovat pomocí ERD (Entity Relationship Diagram). Následně navrhnu třídy reprezentující tyto entity s využitím diagramu tříd a nakonec definuji API pomocí nástroje Swagger.

2.2.1 Databáze a ERD

Způsobů pro uchování dat je samozřejmě mnoho a záleží na povaze vyvíjeného software, kterou z nich si vybereme. Pro menší aplikace, u kterých se nepředpokládá obsluha více uživatelů si vývojář může vystačit se souborovým systémem coby úložištěm. Databáze jsou naproti tomu určeny k uchovávání mnohem větších souborů uspořádaných informací – někdy i obrovských objemů. Databáze umožňují více uživatelům najednou rychlý a bezpečný přístup k datům [12].

Správný návrh databázové struktury je klíčový pro úspěšný vývoj softwarového systému. Přesnější řečeno, pro efektivní a spolehlivé ukládání a zpracování dat je třeba nejprve navrhnout ERD. ERD poskytuje přehled o datové struktuře systému a definuje entity, vztahy a atributy, což umožňuje vytvořit účinnou a flexibilní databázovou strukturu. Podoba ERD pro aplikaci PlaceHolder je znázorněna na obrázku 2.1.



Obrázek 2.1: Entitně-relační diagram

V systému bude figurovat devět následujících entit:

- `app_user`: Tato entita reprezentuje uživatele v systému. Každý uživatel má atribut *email*, který slouží zároveň jako uživatelské jméno a jako kontaktní email, na který budou zasílány notifikace pro daného uživatele. *email* je zároveň kandidátním klíčem, což znamená, že bude v systému unikátní. Atribut *password* pak představuje *hash* hesla uživatele. Atributy *first_name* a *last_name* reprezentují křestní jméno a příjmení uživatele, *user_role* pak uchovává roli uživatele v systému. Atribut *locked* označuje, zda uživatel provedl ověření svého účtu a zda se tím pádem může přihlásit do systému. Defaultní hodnota je po vytvoření nového účtu nastavena na *false*. Atribut *locked* značí, zda je daný uživatelský účet zablokován či nikoliv, defaultní hodnota je nastavena na *false*. Hodnota posledního atributu *has_daily_discount* znamená, zda má uživatel nárok na denní slevu, což reflektuje aktuální benefit karty *Multisport*.
- `refresh_token`: Slouží k uchování bezpečnostního tokenu a ke kontrole, zda byl daný token již použit – nepoužitý token má hodnotu atributu *used_at* rovnu *null* (více v kapitole Autorizace a autentizace). Atributy *created_at* a *expires_at* jsou časové značky vytvoření a expirace daného tokenu. Atribut *user_id* je cizím klíčem odkazující na relaci `app_user` a uchovává identifikátor vlastníka tokenu. Poslední atribut *token* je pak právě samotný unikátní token.
- `user_token`: Tato entita představuje uživatelské tokeny, které jsou využívány pro potvrzení registrace nového uživatele či pro vygenerování nového hesla. O jaký typ tokenu se jedná určuje atribut *token_type*, jenž může nabývat hodnot REGISTRATION či PASSWORD. Pokud je typ tokenu PASSWORD, bude atribut *encoded_password* obsahovat *hash* hesla, které se po uplatnění tokenu nastaví uživateli *user_id*. V případě, že typ tokenu je REGISTRATION, bude hodnota *encoded_password* rovna *null*. Atribut *token*, pomocí kterého se *user_token* uplatní, je hodnota UUID (Universal Unique Identifier). Dalšími atributy *created_at*, *expires_at* a *confirmed_at* jsou časové údaje o vytvoření, expiraci a potvrzení tokenu.

- `location`: Jedná se o entitu představující fyzické prostory sportovního střediska. Každá lokace má vlastní jméno, zastoupené atributem *name*.
- `calendar`: Jedná se o reprezentaci kalendáře, na který budou vázány konkrétní události. Jak je patrné z atributu *location_id*, jeden kalendář náleží právě jedné lokalitě, avšak jedna lokalita může mít více kalendářů. Důvodem je modelový scénář, kdy sportovní středisko má jednu halu (lokaci) s více kurty. V takovém případě lze pro každý kurt dedikovat vlastní kalendář a příslušné události tak držet odděleně. Díky atributu *name* entity `location` se pak může zákazník orientovat v tom, kde se akce, na kterou je přihlášen, fyzicky koná.
- `recurrence_group`: Tato entita byla zavedena pro správu opakovaných událostí. Hodnota atributu. Časový atribut *repeat_until* označuje datum, do kdy se bude daná událost opakovat.
- `recurrence_group_days_of_week`: Tato entita slouží k uchování dní v týdnu, ve které se vyskytuje opakovaná událost. Jednotlivé dny pondělí až neděle jsou mapovány na čísla 1 až 7 a jsou označeny atributem *day_of_week*. Cizí klíč *recurrence_group_id* pak odkazuje na entitu `recurrence_group`. Jedná se tak o alternativu k ukládání pole dní v týdnu přímo v entitě `recurrence_group`, důvodem pro tento přístup je potenciálně vyšší kompatibilita při změně databázového systému.
- `event`: Entita reprezentuje událost, tedy předmět rezervací v systému. Každá událost má svůj název jako atribut *title* a maximální kapacitu *maximum_capacity*, která koresponduje s maximálním počtem rezervací na událost. Dále *event* obsahuje časové údaje o začátku a konci události, kterými jsou atributy *start_time* a *end_time* a datum konání události *date*. Zdánlivě nadbytečný atribut *day_of_week* reprezentuje den v týdnu konání události a je zaveden z důvodu optimalizace při kontrole kolidujících událostí. Volitelným atributem je pak popis události *description*. Dalšími atributy jsou cena události *price* a cena po slevě *discount_price*. Události jsou vázány na konkrétní kalendář, což znázorňuje cizí klíč *calendar_id*. V případě opakované události odkazuje cizí klíč *recurrence_group_id* na entitu `recurrence_group`, která definuje její výskyty. Pokud je událost jednorázová, je hodnota atributu *recurrence_group_id* rovna *null*.

- `reservation`: Poslední entitou systému je rezervace, která obsahuje cizí klíče *event_id* a *owner_id* odkazující na entity `event` a `app_user`. Atribut *discount_applied* značí, zda byla na rezervaci uplatněna *discount_price* z entity `event`.

Pro všechny entity je v rámci vyšší přehlednosti a ucelenosti zaveden syntetický atribut *id* jako primární klíč s hodnotou náhodného UUID.

2.2.2 Diagram tříd

Diagram tříd poskytuje statický pohled na třídy systému, jejich atributy, operace a vzájemné vztahy. Při vývoji software se běžně uvažují dva typy tohoto diagramu. Jedním z nich je analytický model a druhým model návrhový. Rozdíl spočívá v jejich komplexitě, návrhový model často obsahuje mnohonásobně více tříd než model analytický. Dále ve stručnosti popíšu analytický model, jeho rozšířená varianta je pak k nahlédnutí v elektronické příloze.

Jak bylo nastíněno dříve, serverová aplikace bude rozdělena do tří základních vrstev, kdy spolu budou komunikovat vrstvy sousední. Tato skutečnost se promítne i do diagramu tříd, kde každá z entit bude mít svoji vlastní reprezentaci. Pro komunikaci s databází jsou zavedeny repozitáře, které implementují základní CRUD operace přímo nad databází a podporují tzv. *pagination*, což je technika používaná k rozdělení velkého seznamu dat do menších částí, nazývaných stránky, což umožňuje procházet a zobrazovat data po částech. Každá stránka obsahuje omezený počet položek, což usnadňuje navigaci a zvyšuje výkon aplikace.

Prostřední vrstva je reprezentována třídami nesoucí název *service*. Tato vrstva komunikuje bezprostředně vrstvou repozitářů a vystavuje veřejné metody, které jsou potřeba pro logické zpracování příchozích požadavků.

Tím se dostáváme ke třídám pojmenované jako *controller*, které po vzoru reprezentační vrstvy implementují metody potřebné pro klientskou aplikaci. Aby se zamezilo publikování atributů entit, které nejsou pro klientskou aplikaci relevantní, nebo by jejich zveřejnění představovalo bezpečností riziko, jsou zavedeny třídy se sufixem DTO (*data transfer object*), které slouží jako návratové hodnoty pro controllery.

2.2.3 REST API

API je rozhraní, které umožňuje různým softwarovým komponentám vzájemně komunikovat za pomoci souboru pravidel a protokolů pro komunikaci. V dnešní době je nejvíce populárním způsobem implementace API takzvané REST (Representational State Transfer) API.

Koncept REST s sebou přináší řadu architektonických požadavků a omezení pro přenos dat mezi klientem a serverem pomocí standardních HTTP metod (např. GET, POST, PUT, DELETE) a formátů dat (např. JSON, XML), přičemž každý zdroj dat má mít jedinečnou URL. Základním principem REST API je takzvaná *statelessness*, což vyjadřuje stav nezávislosti serveru na stavu klienta. To znamená, že server neuchovává žádné informace o předchozích požadavcích od klienta a každý požadavek je vyhodnocován samostatně bez jakéhokoliv předchozího kontextu. Tento princip umožňuje větší flexibilitu a škálovatelnost systému, jelikož požadavky mohou být obsluhovány různými servery bez nutnosti synchronizace stavu mezi nimi [13].

3 Implementace

3.1 Použité technologie

Pro vývoj rezervačního systému bylo zásadní vybrat vhodné technologie pro vývoj webových aplikací, které podporují rychlý vývoj, udržitelnost, a umožňují splnit stanovené funkční a nefunkční požadavky. Klíčové technologie využité pro tento projekt zahrnují Javu, Spring boot, React JS a PostgreSQL. Pro nasazení aplikace slouží Docker, což si popíšeme podrobněji v pozdější kapitole o nasazení aplikace.

3.1.1 Spring Framework

Spring je rozsáhlý a v dnešní době velmi populární aplikační rámec pro platformu Java, který byl vyvinut jako reakce na některé nedostatky rámce Java Enterprise Edition. Spring výrazně zjednodušuje vývoj pondikových aplikací s využitím návrhových vzorů *dependency injection* a *inversion of control*, což zároveň podporuje udržování modularity, testovatelnosti a udržitelnosti kódu.

Spring nabízí řadu modulů pro různé typy aplikací. Pro vývoj systému PlaceHolder coby webové aplikace je například velmi vhodný modul Spring Web, který mimo jiné umožňuje rychlé vytvoření REST API a snadnou integraci s dalšími moduly, jako je Sprig Security. Spring Security poskytuje zabezpečení aplikace pomocí funkcí pro autorizaci a autentizaci a stará se o ochranu proti běžným útokům, jako je *Cross-Site Request Forgery* (CSRF). Dalším podstatným modulem Spring Frameworku je Spring Data JPA, který zjednodušuje přístup k datům a poskytuje abstrakci nad nízko úrovňovými databázovými operacemi a usnadňuje tím například používání transakcí a implementaci základních operací *create*, *read*, *update* a *delete* (CRUD). Spring Data JPA zároveň standardně používá Hibernate pro objektově-relační mapování (ORM), což je technologie zajišťující konverzi dat mezi relační databází a nativními objekty v Javě.[14]

3.1.2 React JS

React JS je populární knihovna programovacího jazyku JavaScript pro vývoj uživatelských rozhraní webových aplikací. Byl vyvinutý týmem společnosti Facebook (dnes Meta) a je široce používán pro vytváření moderních, responzivních a efektivních webových aplikací. React umožňuje vývojářům pracovat s komponentami, což jsou znovupoužitelné a izolované bloky, které spolu tvoří celou aplikaci [15]. Jedním z klíčových prvků této knihovny je virtuální DOM (*Document Object Model*), který umožňuje efektivní aktualizaci uživatelského rozhraní bez nutnosti překreslování celé stránky. Tím se zlepšuje výkon aplikace celková aplikace působí svižněji.[16] Knihovna také poskytuje bohatý ekosystém rozšíření, jako jsou Redux a React Router, které dále rozšiřují možnosti React JS a usnadňují vývoj komplexních webových aplikací.

3.1.3 Vite

3.1.4 PostgreSQL

3.1.5 Docker

3.2 Autorizace a autentizace

JWT

3.3 Návrhové vzory

3.4 Uživatelské rozhraní

4 Testování

4.1 Metody testování

4.2 Testovací scénáře

5 Nasazení aplikace

6 Závěr

Bibliografie

1. The ineluctable middlemen. *The Economist* [online]. 2012, vol. 404, no. 8793 [cit. 2022-11-07]. ISSN 0013-0613. Dostupné z: https://www.economist.com/business/2012/08/25/the-ineluctable-middlemen.
2. RESERVANTO, s.r.o. *Porovnání variant* [online]. [B.r.]. [cit. 2023-03-15]. Dostupné z: https://www.reservanto.cz/PorovnaniVariant.
3. BOOKED4.US. *Functions* [online]. [B.r.]. [cit. 2023-03-15]. Dostupné z: https://booked4.us/en/functions/.
4. RESERVIO, s.r.o. *Všechny Funkce Rezervačního Systému* [online]. [B.r.]. [cit. 2023-03-15]. Dostupné z: https://www.reservio.com/cs/funkce.
5. *Difference between functional and non-functional requirements* [online]. [cit. 2023-03-11]. Dostupné z: https://www.javatpoint.com/functional-vs-non-functional-requirements.
6. ROYCE, Dr. Winston W. Managing the Development of Large Software Systems [online]. 1970 [cit. 2016-12-09]. Dostupné z: http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf.
7. *Manifesto for Agile Software Development* [online]. Utah, USA: Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Steve Mellor, Robert C. Martin, Ken Schwaber, Jeff Sutherland, Dave Thomas, 2001 [cit. 2022-11-07]. Dostupné z: https://agilemanifesto.org/.
8. *Introduction to kanban* [online]. Austin, Texas: Rachaelle Lynn, 2001 [cit. 2022-11-07]. Dostupné z: https://www.planview.com/resources/guide/introduction-to-kanban.
9. PANÁK, Jaroslav. *Návrh a zavedení agilní metodiky vývoje software do IT týmu malého podniku*. Brno, 2018. Dostupné také z: https://is.muni.cz/th/ql3ew/. Diplomová práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce Josef PROKEŠ.

10. FOWLER, Martin. *Software architecture guide* [online]. 2019-08-01. [cit. 2023-03-15]. Dostupné z: https://martinfowler.com/architecture.

11. WALKER, Vicki; RESELMAN, Bob; BUTANI, Anand. *14 software architecture design patterns to know* [online]. Red Hat, Inc., 2022 [cit. 2023-03-15]. Dostupné z: https://www.redhat.com/architect/14-software-architecture-patterns.

12. *Database defined* [online]. Oracle s.r.o., [b.r.] [cit. 2023-03-15]. Dostupné z: https://www.oracle.com/database/what-is-database/.

13. *What is a REST API* [online]. Red Hat, Inc., 2022 [cit. 2023-03-15]. Dostupné z: https://www.redhat.com/en/topics/api/what-is-a-rest-api.

14. TEAM, Spring Framework. *Introduction to the Spring Framework* [online]. 2009. [cit. 2023-03-30]. Dostupné z: https://docs.spring.io/spring-framework/docs/3.0.0.M4/reference/html/ch01s02.html.

15. META. *React - A JavaScript library for building user interfaces* [online]. 2021. [cit. 2023-03-30]. Dostupné z: https://17.reactjs.org/.

16. FU, React Kung. *React: The Virtual DOM* [online]. 2015. [cit. 2023-03-30]. Dostupné z: https://www.codecademy.com/article/react-virtual-dom.