Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Mobilní aplikace pro rezervace objektů

Plzeň, 2013

Lukáš Gemela

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 13. května 2013

Lukáš Gemela

Abstract

Currently there is no information system that enables effective lending and reservation management of movable and immovable properties in a general way. This thesis is dedicated to solving this problem by using the unique possibilities of mobile devices. First, the problem is generalized and the usability of mobile device technologies is analyzed. Afterwards, a specific implementation of the information system is introduced. The system is based on the client-server architecture in which the client is represented by a mobile device using the Android operating system.

Obsah

1 Úvod **1**

2 Problematika správy výpůjček **2**
2.1 Motivace 2
2.2 Existující příbuzné systémy 4
2.3 Generalizace problému výpůjček 5
2.4 Shrnutí 15

3 Mobilní zařízení **16**
3.1 Mobilní zařízení jako pracovní nástroj 16
3.2 Technické prostředky mobilních zařízení 17
3.3 Mobilní aplikační platformy 23
3.4 Vývoj mobilních aplikací 27
3.5 Shrnutí 31

4 Požadavky na systém správy rezervací **32**
4.1 Využitelnost mobilních zařízení 32
4.2 Obecný popis systému 36
4.3 Funkce webového rozhraní 38
4.4 Funkce klientské aplikace 50
4.5 Mimofunkční požadavky 54

5 Možnosti návrhu webových aplikací **55**
5.1 Vícevrstvá architektura 55
5.2 Aplikační vrstva 58
5.3 Vrstva persistence dat 60
5.4 Prezentační vrstva 65
5.5 Architektura orientovaná na služby 75
5.6 Technologické nástroje pro vývoj 78
5.7 Shrnutí 80

6 RAT - Reserve A Thing! **81**
6.1 Informační systém RAT 81
6.2 RAT server 82
6.3 RatDroid 99
6.4 RatSharp 104
6.5 Testování a nasazení systému 105

7 Závěr **106**

A Rozvržení mobilní aplikace **107**

B Popis webových REST služeb **113**

C Uživatelská příručka **118**
C.1 Webové rozhraní 118
C.2 Aplikace RatDroid 124

D Administrátorská dokumentace **132**
D.1 Nasazení webové aplikace 132
D.2 Nasazení mobilní aplikace *RatDroid* 135

E Obsah přiloženého média **137**

1 Úvod

Téměř v každé rozsáhlejší organizaci obvykle existuje množina inventárního majetku, který je dostupný členům této organizace k zapůjčení nebo dočasnému využívání. Mluvíme-li o tomto majetku, máme na mysli jak věci movité, jako je například výpočetní technika nebo kancelářské pomůcky, tak věci nemovité, jako jsou místnosti, budovy atp. Za modelovou organizaci si můžeme představit například neziskové organizace, malé a střední podniky nebo vzdělávací instituce. Termínem „výpůjčka" předmětu poté nemyslíme předání do déletrvajícího užívání v řádech týdnů nebo měsíců, ale krátkodobé zapůjčení předmětu jeho žadateli s tím předpokladem, že se očekává jeho opětovné navrácení v předem domluveném termínu tak, aby si předmět mohl půjčit jiný žadatel.

Tato diplomová práce se zabývá problémem správy výpůjček movitých i nemovitých objektů a implementací jeho řešení za pomoci unikátních možností mobilních zařízení. Nejprve bude tento problém zobecněn a následně budou nastíněny základní požadavky kladené na vznikající systém tak, aby bylo možné správu výpůjček efektivně řešit (Kap. 2). V Kap. 3 bude čtenář seznámen s nejrozšířenějšími mobilními technologiemi a platformami. Následně se v Kap. 4 čtenář seznámí s úplnou specifikací požadavků, které bude systém implementovat za využití technologií mobilních zařízení. V realizační části (Kap. 5) budou rozebrána obecná architektonická řešení informačních systémů s ohledem na platformu Java, aby byla poté ukázána konkrétní implementace informačního systému pro půjčování a rezervaci objektů (Kap. 6).

2 Problematika správy výpůjček

V této kapitole rozebereme stávající problém, zobecníme jej do podoby logického modelu a následně nastíníme základní požadavky na systém tak, aby bylo možné správu výpůjček efektivně řešit.

2.1 Motivace

Motivací pro vytvoření této práce byla jistá nepraktičnost používání stávajícího systému správy výpůjček movitých věcí na Katedře informatiky a výpočetní techniky při Západočeské Univerzitě v Plzni. Naskytla se otázka, zdali by nebylo možné učinit správu výpůjček efektivnější, jednodušší a průchodnější jak pro uživatele, tak pro správce tohoto systému. Problematika správy výpůjček a rezervací však může být vztažena na libovolnou organizaci, kde k těmto aktivitám pravidelně dochází, a lze tedy říci, že se jedná o obecný problém.

2.1.1 Případová studie

Na katedře existuje množina movitých objektů (notebooků a dataprojektorů), které si mohou zaměstnanci a spolupracovníci katedry na omezenou dobu zapůjčit. Toto technické vybavení je dostupné v sekretariátu katedry a je možné je od sebe rozlišit podle papírového štítku s identifikačním číslem, který je fyzicky připevněn na vybavení.

Na katedře je zaveden „výpůjční list“, kde každý zaměstananec přebírající předmět potvrzuje převzetí předmětu a zavazuje se, že jej do dohodnutého termínu opět vrátí. Je třeba definovat datum a čas výpůjčky a vrácení objektu. Výpůjční list je veden v podobě papírového formuláře. Typický scénář výpůjčky je tedy takový, že vyučující požadující notebook pro svou přednášku musí nejprve nalézt identifikační číslo vyžadovaného notebooku a poté jej musí spolu se svým jménem a datem výpůjčky a datem vrácení zanést do jednoho řádku papírového formuláře. Při fyzickém vrácení notebooku poté vyučující obvykle stvrdí tento akt vlastním podpisem řádky s údaji o vypůjčení objektu.

Je také možné zápisem do jiného formuláře vytvářet rezervace vybavení. Při vytváření rezervací se musí zaměstnanec manuálním procházením již vytvořených záznamů ujistit, že jeho rezervace se nekryje s jinou již dříve vytvořenou rezervací. Je třeba specifikovat rezervované zařízení, datum a časové rozmezí rezervace. Slov-

ním popisem je také možné rezervace opakovat v čase – zaměstnanci si rezervují vybavení podle rozvrhových akcí a ty se v rámci semestru opakují. Tyto pravidelné rezervace jsou poté zaměstnanci sekretariátu katedry znázorňovány v týdenním papírovém kalendáři, kde je graficky podchycena délka trvání rezervace a jména všech účastníků.

Správa nemovitých objektů (tj. místností) je řešena pomocí celouniverzitního informačního systému IS/STAG, který umožňuje mapovat rozvrhové akce (jako jsou např. přednášky, semináře, zkoušky) na místnosti a následně vytvářet kalendáře místností v rámci semestru. Tento informační systém nicméně neumožňuje jakkoliv řešit rezervace těch místností, které nejsou tímto systémem evidovány (např. zasedací místnost na katedře) nebo jejichž rezervace jdou nad rámec standardních rozvrhových akcí.

2.1.2 Nevýhody stávajícího řešení

Stávající systém je zaveden pouze nad omezenou množinou vybavení. Pokud by však měl být počet notebooků a dataprojektorů zvýšen, případně by mělo být umožněno realizovat výpůjčky i nad jinými předměty, řešení v podobě papírových formulářů začne narážet na své hranice použitelnosti. Dále se s ním pojí celá řada nedostatků:

- movité objekty nelze na základě papírového formuláře efektivně rezervovat do budoucnosti.
- záznamy o zapůjčení jsou dostupné pouze na jednom místě v papírové podobě.
- nelze efektivně sledovat využitelnost předmětů.
- samotný akt výpůjčky a rezervace je s ohledem na nutné vyplňování papírových formulářů poměrně zdlouhavý a obtěžující.
- chybí aktuální přehled jaké objekty jsou půjčeny, rezervovány, nevráceny atp.
- uživatelé mají obecně při vyplňování papírových formulářů sklony k chybám a nepřesnostem.
- je nutné udržovat týdenní kalendář rezervací a manuálně provádět kontrolu, jestli se rezervace nepřekrývají.
- neexistuje jednotný časový formát pro definování opakovaných rezervací.

Realizace rezervací nemovitých objektů pak zcela chybí. Bylo by vhodné zobecnit problém výpůjčky movitých a nemovitých objektů, zavést jednotnou terminologii tak, aby byly navržené postupy obecně použitelné, a navrhnout řešení v podobě nějakého druhu informačního systému.

2.2 Existující příbuzné systémy

Teoretická řešení propůjčování zdrojů ve velmi obecné rovině existují a jsou popsána v rámci tzv. *Inventory managementu.* Tento pojem pochází z logistiky a označuje sadu postupů zabývajících se otázkami dohledu nad skladovými zásobami, jejich objednávání a udržování jejich počtu v udržitelných mezích. Systémy, které se nejvíce blíží potřebám této práce, se poté zpravidla označují jako inventární systémy (*Inventory Systems* nebo *Stock Systems*).

Ty lze charakterizovat jako sadu hardwarových a softwarových nástrojů, které automatizují proces sledování inventárních objektů. Druh sledovaných objektů může být vpravdě jakýkoliv počitatelný předmět (například oblečení, knihy, technické vybavení). Moderní inventární systémy jsou téměř výhradně založeny na technologii skenovatelných čárových kódů, QR kódů nebo RFID štítků (viz dále).

Takových systémů je celá řada, obvykle je inventární systém součástí celého balíku rozsáhlých podnikových systémů jako je například Microsoft Dynamics nebo Oracle SCM. Všechny ale mají jedno společné – jsou zaměřeny na výrobu a logistiku inventáře, který je určen k prodeji, nikoliv k účelům zapůjčování zaměstnancům (nebo členům organizace).

Správou takového druhu objektů, které se obvykle označuje slovem *assets*, se zabývá jiná disciplína – tzn. *Asset Management* [1]. V podnikové terminologii se slovo asset překládá jako tzv. *provozní aktivum*, neboli objekt, který generuje přidanou hodnotu. Mezi aktiva můžeme zařadit pozemky, budovy, ale i stroje, softwarové licence a další zařízení. Asset management se poté zabývá správou těchto aktiv během jejich životního cyklu tak, aby jejich provozování generovalo maximální přidanou hodnotu.

Asset Management systémů je celá řada – z těch největších zde jmenujme např. Maximo Asset Management od IBM, Software Asset Management od společnosti Microsoft a z těch ryze českých například ALVAO Asset Management. Tyto systémy zpravidla umožňují přidělovat jednotlivá aktiva mezi zaměstnance v rámci evidence majektu (aby bylo vždy jasné, kdo má danou SW licenci, počítač atd. aktuálně v držení) – žádný z nich ovšem neposkytuje možnost plánování, rezervací ani okamžitého vyzvednutí z podnětu zaměstnanců. Navíc naším cílem není majetek evidovat (i když nějaká forma evidence bude pravděpodobně nutná), ani řídit

jeho životní cyklus, ale pouze řídit jeho „časový“ oběh mezi uživateli.

Jak je vidět, analogie řešení našeho problému se v podnikové informační sféře nehledá snadno. Správě movitých objektů se pravděpodobně nejvíce blíží různé knihovní systémy, jejichž součástí je i výpůjční protokol, případně další specializovaný software pro různé půjčovny movitých objektů. Jako příklad aplikace umožňující rezervace nemovitých objektů bychom mohli jmenovat MS Outlook. Ten umožňuje rezervovat místnosti a další předem definované zdroje pro naplánované schůzky a jiné časové události. Autorovi této práce se však nepodařilo najít žádný stávající software, který by celistvě řešil nastíněný problém vypůjčování majetku v obecné rovině.

2.3 Generalizace problému výpůjček

Úkolem této kapitoly je zobecnit problém výpůjček a definovat termíny, se kterými by bylo možné dále pracovat.

2.3.1 Objekt, Uživatel, Rezervace

Protože řešíme výpůjčky jak nemovitých, tak movitých předmětů, nebudeme tyto pojmy již nadále používat a shrneme jej do jednoho termínu **objekt**, a to následovně:

Definice 1. *„Objekt“ je movitý nebo nemovitý majetek organizace, který lze dočasně půjčit uživatelům.*

Aktem „zapůjčení objektu“ budeme tedy nadále rozumět nejen fyzické zapůjčení např. notebooku, ale i zapůjčení místnosti pro prezentaci, konferenčního sálu pro přednášku, celé budovy pro různé akce atp. Nyní definujme termín pro uživatele systému:

Definice 2. *„Uživatel“ je člen organizace nebo jiná osoba, která je oprávněna k vypůjčování objektů.*

Zcela záměrně neomezujeme uživatele pouze na hranice organizace. Tím nám možný počet užití dramaticky narůstá například o parkovací místa pro návštěvníky nebo online místenku k sezení v autobuse. Jediné, co je nutné pro koncový systém řešit, je právě oprávněnost k výpůjčce.

Nyní je třeba zavést logiku rezervací objektů. Aktem rezervace objektu uživatel říká, že je to on, kdo bude žádaný objekt v budoucím časovém úseku využívat a žádný jiný uživatel není oprávněn do tohoto časového úseku jakkoliv zasahovat nebo v tomto úseku objekt sám využívat.

Definice 3. *„Rezervace“ je časový úsek počínající v přítomnosti nebo budoucnosti, ve kterém daný objekt může mít vypůjčený pouze daný uživatel.*

Rezervací může mít jeden uživatel libovolné množství a libovolný počet rezervací může být vázán k jednomu objektu – jedinou podmiňující prerekvizitou pro vytvoření rezervace je neexistence jiné rezervace ve stejném časovém úseku. Výsledkem je vlastně kalendář časových událostí – rezervací, a to jak pro uživatele, tak pro objekt.

2.3.2 Životní cyklus rezervace

Nyní je nutné si položit otázku, jak zobecnit vlastní vyzvedávání a vracení objektů. Je třeba si uvědomit, že tato činnost se již vlastně nevztahuje na konkrétní objekty, ale na jejich rezervace. Samozřejmě, objekt zde hraje roli předmětu rezervace, bez něj by vlastně rezervace samotná postrádala smysl – nicméně je to právě rezervace, která bude uživateli vytvářena, upravována a nakonec fyzicky „vyzvednuta“ převzetím objektu.

Již bylo řečeno, že k samotnému vytvoření výpůjčky nebo rezervace dochází v okamžiku vyplnění kolonky v papírovém formuláři. Tím uživatel vlastně „potvrdil“ svou výpůjčku a je nyní opravdu oprávněn k tomu, aby mohl žádaný objekt převzít. Co ale u např. výše zmíněného příkladu parkovacího stání? Tam k převzetí dříve rezervovaného objektu dochází už samotným aktem příjezdu auta, u rezervace místnosti fyzickým obsazením místnosti atp. Je přinejmenším pošetilé v těchto případech vyžadovat po uživatelích jakoukoliv další akci. Jednak je to zbytečný proces navíc a jednak je mizivá šance, že by uživatelé v těchto situacích modelem předepsanou akci skutečně prováděli. Řešením je tedy rozdělení objektů do dvou kategorií:

Definice 4. *Povinně potvrzované objekty při vyzvednutí jsou objekty, jejichž rezervace budou pro vyzvednutí vyžadovat po majitelích těchto rezervací danou akci. Nepotvrzované objekty při vyzvednutí žádnou akci vyžadovat nebudou a k jejich vyzvednutí dojde posunem počátku časového úseku rezervace do současnosti.*

Čtenář by neměl nabýt dojmu, že povinně potvrzované předměty jsou vždy ty movité a nepotvrzované ty nemovité. Díky tomu, že vynucením potvrzení o vyzvednutí vzniká vlastně nový záznam o aktivitě uživatele, je možné si představit scénáře, kdy se i u nemovitých objektů uplatní potvrzování a naopak:

- chceme sledovat **reálnou** užívanost objektů.
- v současnosti běžící rezervace, které jejich majitel nepotvrdí (a objekt si tak vlastně nevyzvedne), mohou „propadnout“ a objekt si může v překrývajícím časovém rámci rezervovat jiný uživatel.
- movité objekty se pohybují v omezeném prostoru, není potřeba jejich oběh potvrzováním komplikovat.

Z podobných důvodů je dále zavedeno i potvrzování při vracení objektu:

Definice 5. *Povinně potvrzované objekty při vracení jsou objekty, jejichž rezervace budou pro vrácení objektu vyžadovat po majitelích těchto rezervací danou akci. Nepotvrzované objekty při vrácení žádnou akci vyžadovat nebudou a k jejich vrácení dojde posunem konce časového úseku rezervace do současnosti.*

Výhody jsou zřejmé – rezervaci je možné uživatelem předčasně ukončit a ostatní uživatelé tak budou mít v reálném čase informaci o tom, zdali je jimi sledovaný objekt aktuálně k dispozici.

Zavedením povinně potvrzovaných a nepotvrzovaných objektů, ať už při vyzvednutí či při vracení, nám umožní jistou formu genericity a flexibility celého modelu – záleží jen na konkrétní situaci a použití, reálném druhu objektů a přání uživatelů.

2.3.3 Periodické rezervace

Nejrůznější lidské aktivity se v čase často opakují – nejinak je to i s našimi rezervacemi. Příkladem může být např. každotýdenní setkání týmu s potřebou rezervace místnosti, přednáška s rezervací projektoru apod. Abychom tento jev nějak podchytili, zavedeme pojem periodické rezervace:

Definice 6. *Periodická rezervace je taková rezervace, která se opakuje v čase s pevně danou periodou a pevně daným počtem opakování.*

Periodická rezervace může být samozřejmě vedena k libovolnému typu objektu – pokud bude objekt povinně potvrzovaný při vyzvednutí, musí uživatel vyzvednutí rezervace potvrzovat. Opakování rezervace však způsobí, že tak musí učinit na počátku každého časového úseku rezervace. Obdobná situace bude platit i pro vracení objektů.

Zde však nastává určitý nesoulad. Rezervace sama o sobě sice k sobě váže objekt a uživatele, dosud definovaný model však v případě cyklické rezervace pokulhává – jak definovat periodičnost? Od jakého data? Po jak dlouhou dobu? Jaký

časový úsek mají zabrat jednotlivé periody? Co když uživatel bude chtít některý časový úsek přeskočit nepříklad z důvodu výluky v rozvrhu?

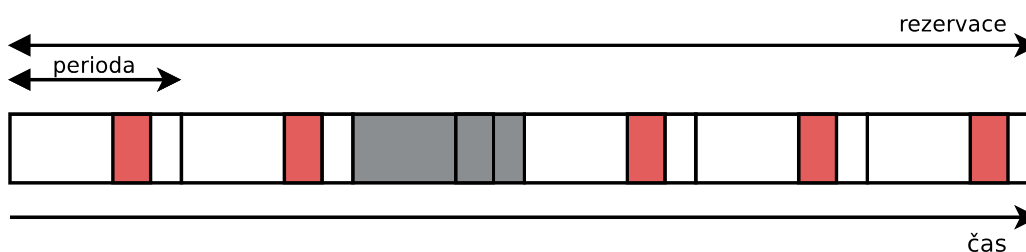
Z tohoto důvodu upřesníme předcházející definici následovně:

Definice 7. *Každá rezervace se skládá z jedné nebo více časových period, kde každá perioda obsahuje časový úsek rezervace v dané periodě.*

a přidáme definici další, která nám umožní jednotlivé periody „vypínat“:

Definice 8. *Neplatná perioda je taková perioda, ve které se časový úsek rezervace interpretuje tak, jako by žádná rezervace v této periodě nevznikla.*

Vysvětlení těcho definic je názorně ukázáno na Obr. 2.1. Každá rezervace se skládá z jedné a více časových period, přičemž periody mají konstantní délku. V těchto periodách jsou poté zahrnuty jednotlivé časové úseky samotné rezervace objektu (znázorněno červenou barvou). Uživatel může v těchto úsecích používat předmět, který si zarezervoval. Pokud je některá z period neplatná (na obrázku znázorněno šedou barvou), je i časový úsek rezervace v této periodě obsažený neplatný a bude se interpretovat tak, jako by žádná rezervace v dané periodě nikdy nevznikla.



Obrázek 2.1: Struktura periodické rezervace.

2.3.4 Potvrzení vyzvednutí objektu

Samotný akt vyzvednutí rezervace se zdá jako poměrně triviální událost. Nicméně narozdíl od vytvoření rezervace je tato událost svázána s aktuálním časem a časem počátku vyzvedávané rezervace. Jistě by nemělo být možné vyzvedávat objekty, jejichž rezervace ještě nezačala. Protože však uživatelé sotva budou vyzvedávat objekty v přesně daný čas počátku rezervace, je třeba umožnit předčasné vyzvednutí, případně vyzvednutí po termínu začátku rezervace. Dále je třeba se ptát, co se stane s rezervacemi, které nebudou vyzvednuty.

Aby tyto problémy mohly být uspokojivě vyřešeny, je nejprve nutné definovat základní časové konstanty rezervací:

Definice 9. *„Minimální délka rezervace“ je minimální délka časového úseku rezervace v každé své periodě.*

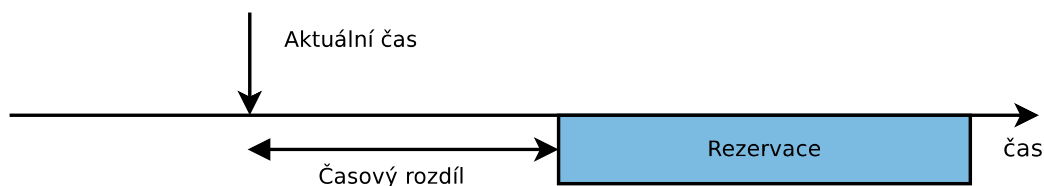
Definice 10. *„Maximální délka rezervace“ je maximální délka časového úseku rezervace v každé své periodě.*

Definice 11. *Maximální délka rezervace je menší nebo rovna časové délce periody rezervace.*

Rezervacím jsou tímto definovány meze pro jejich časové úseky ve všech periodách rezervace.

Předčasné vyzvednutí

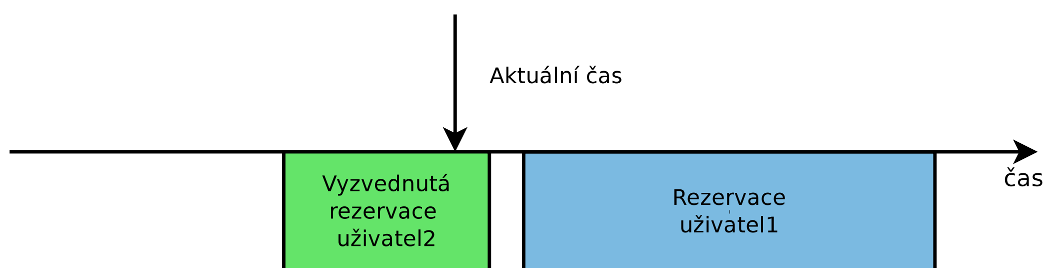
Na Obr. 2.2 je znázorněna časová osa, aktuální čas a situace předčasného vyzvednutí rezervace. Je třeba se nejprve ptát, jak velký časový rozdíl uživatele dělí od času vyzvednutí a času začátku rezervace objektu, která uživateli patří a zda taková rezervace vůbec existuje.



Obrázek 2.2: Předčasné vyzvednutí rezervace.

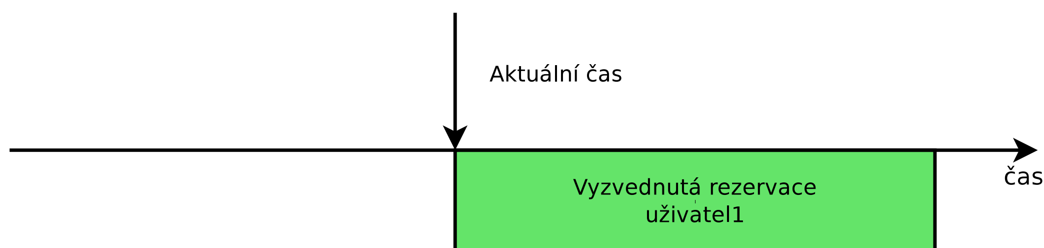
Pokud je tento rozdíl menší než minimální délka rezervace, znamená to, že mezi aktuální čas a čas počátku rezervace se již žádná nová rezervace nevejde. Nyní je třeba ještě zvážit další rezervace, které mohou do časového rozdílu zasahovat. Na Obr. 2.3 je znázorněna situace, kdy se *uživatel1* pokouší předčasně vyzvednout objekt na svou v minulosti vytvořenou rezervaci.

To však není možné, protože mezi aktuálním časem a časem počátku rezervace stále probíhá rezervace *uživatele2*. *Uživateli1* tedy nezbývá nic jiného, než vyčkat až blokující rezervace skončí a *uživatel2* vyzvednutý objekt nevrátí.



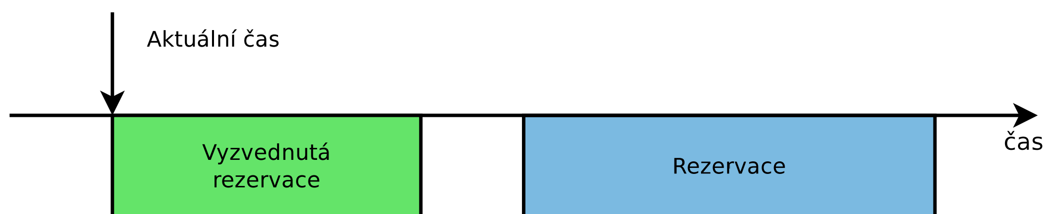
Obrázek 2.3: Kolize rezervací při vyzvedávání.

Pokud však žádná taková blokující rezervace neexistuje, je možné objekt vyzvednout i předčasně. Počátek rezervace se posune na aktuální datum a rezervace se označí jako vyzvednutá. Tuto situaci ilustruje Obr. 2.4.



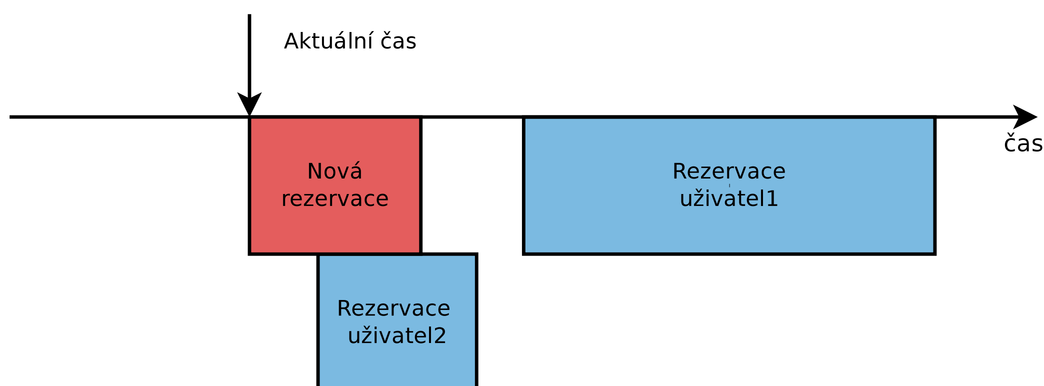
Obrázek 2.4: Úspěsné vyzvednutí rezervace.

Pokud je rozdíl od času vyzvednutí a času začátku rezervace objektu větší než minimální délka rezervace, uživatel vyzvedává objekt příliš brzy a není možné objekt vyzvednout. Není však důvod proč by uživateli nemohla být vytvořena rezervace nová s počátkem v aktuálním čase. Termín ukončení rezervace bude moci uživatel sám definovat. Nově vytvořená rezervace by měla být také automaticky vyzvednuta, jak ukazuje Obr. 2.5.



Obrázek 2.5: Vytvoření nové rezervace.

Vytvoření rezervace však musí splňovat podmínku o neexistenci jiné rezervace na stejný objekt ve stejném časovém úseku. Na Obr. 2.6 je znázorněna situace, kdy se *uživatel1* pokouší předčasně vyzvednout objekt na základě budoucí rezervace.

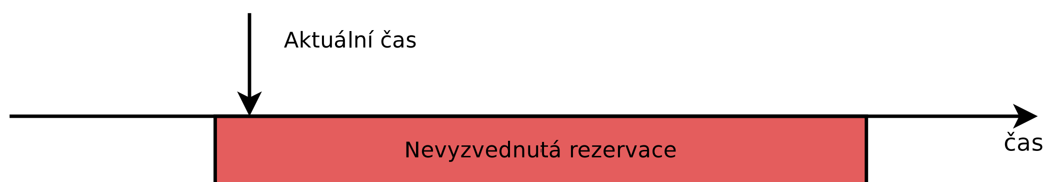


Obrázek 2.6: Kolize rezervací při vytváření.

Toto vyzvednutí je z důvodu přílišné vzdálenosti rezervace odmítnuto a *uživateli1* je umožněno vytvoření rezervace nové, s libovolnou délkou trvání. Ten tedy zadá předpokládanou dobu trvání nové rezervace a pokusí se rezervaci vytvořit. Nyní je nutné zkontrolovat, zda v časovém úseku nové rezervace neexistuje jiná rezervace na stejný objekt. Jak je vidět, tato rezervace skutečně existuje a patří *uživateli2* a proto vytvoření nové rezervace pro *uživatele1* skončí neúspěšně.

Vyzvednutí po termínu

Uživatelé z libovolného důvodu nemusí termín vyzvednutí objektu včas stihnout. Rezervace již začala a objekt není stále vyzvednut – stává se tedy nevyzvednutou. Tuto situaci ilustruje Obr. 2.7.



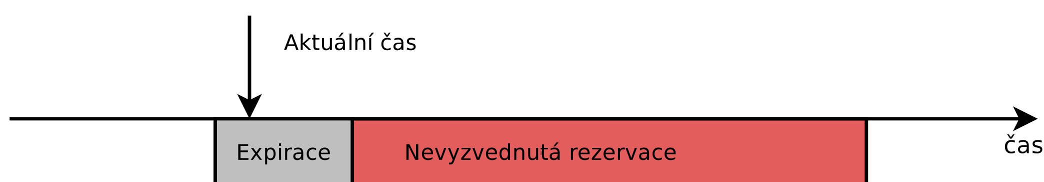
Obrázek 2.7: Nevyzvednutá rezervace.

Otázkou je, jak se takovým rezervacích zachovat. Jistě není žádoucí blokovat ostatní uživatele rezervací, která ztratila své původní poslání – zaručit jejímu majiteli výpůjčku objektu. Ten si však žádný objekt fyzicky nepůjčil, neprovedl potvrzení o vyzvednutí a tím pádem se celá rezervace stává neplatnou. Na druhou stranu, jistě chceme umožnit i vyzvednutí objektu po termínu rezervace.

Rozumným kompromisem bude zavedení další časové konstanty do našeho modelu:

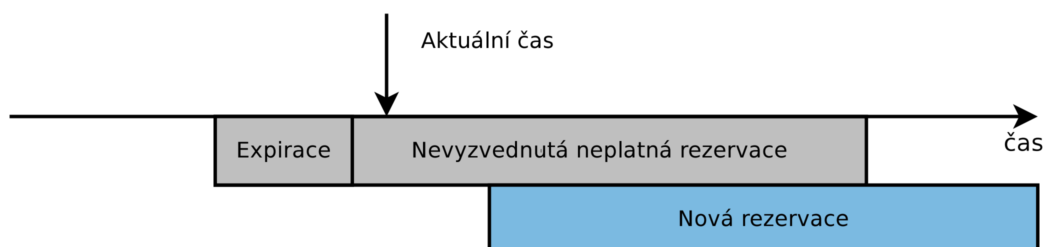
Definice 12. *„Expirace rezervace" je časový úsek od začátku nevyzvednuté rezervace, po který je rezervace chápána jako stále platná.*

Pokud je počátek rezervace roven aktuálnímu času a rezervace stále není vyzvednuta, je vyčleněn časový úsek, tzv. *expirace rezervace*, po který bude rezervace stále chápána jako platná. V tomto časovém úseku ostatní uživatelé nebudou moci vytvářet své rezervace na stejný objekt a rezervace může být stále vyzvednuta jejím majitelem. Tato situace je znázorněna na Obr. 2.8.



Obrázek 2.8: Expirace rezervace.

Jak ilustruje Obr. 2.9, až se čas dostane za dobu expirace rezervace a tato stále není vyzvednuta, bude umožněno ostatním uživatelům vytvořit vlastní rezervaci v časovém úseku této nevyzvednuté rezervace. Pokud se tak stane, je původní časový interval rezervace automaticky zneplatněn a majitel rezervace již nemůže objekt vyzvednout (tato změna se nepromítne do ostatních period v případě cyklické rezervace).



Obrázek 2.9: Zneplatnění rezervace.

Tento přístup má ještě jednu výhodu – i po vypršení doby expirace je stále možné rezervaci vyzvednout. Lze tak ale učinit pouze v případě, kdy jiní uživatelé nevytvořili nové rezervace do časového úseku rezervace původní.

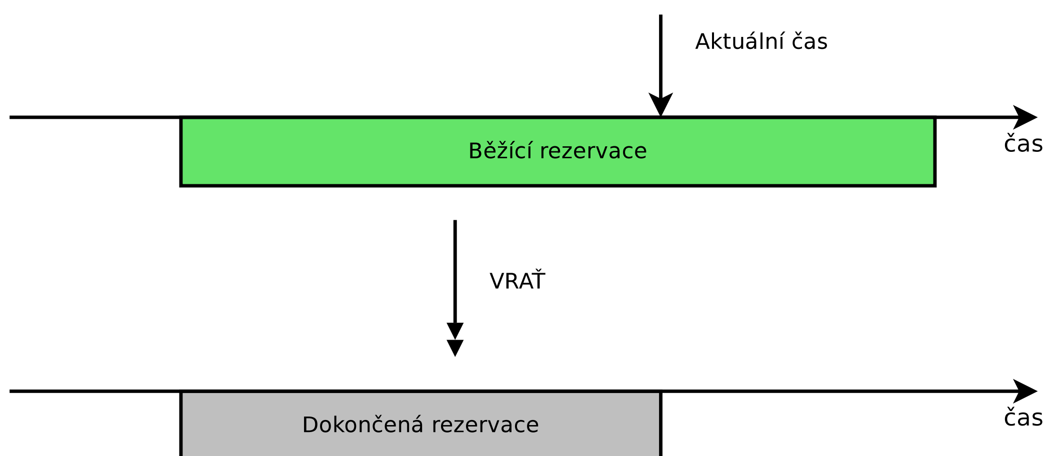
2.3.5 Potvrzení vrácení objektů

Jak již bylo zmíněno, administrátor systému bude moci vynutit potvrzení vrácení objektu. Tím se opět dostáváme do situace, kdy je nutné definovat chování

akce vrácení objektu v různých časových intervalech. Očividně nebude možné potvrzovat vracení těch objektů, které jsou vedeny jako povinně potvrzované při vyzvednutí a jejichž rezervace nebyla vyzvednuta.

Předčasné vrácení objektu

V tomto případě dojde ke zkrácení doby rezervace na čas vrácení objektu a celá výpůjčka tím bude dokončena. Je tak mimo jiné umožněno, aby i ostatní uživatelé využili čas po této rezervaci pro své výpůjčky. Případ předčasného vrácení je zobrazen na Obr. 2.10.



Obrázek 2.10: Předčasné vrácení objektu.

Vrácení objektu po termínu

Pokud je objekt vrácen po termínu konce rezervace, nedojde k žádné změně tohoto termínu – mohli bychom se dostat do kolize s jinou rezervací. Rezervace vráceného objektu bude pouze vedena jako ukončena.

Nevrácení objektu

V případě déletrvajícího nevrácení objektu uživatel mohl pouze zapomenout objekt vrátit nebo toto vrácení potvrdit, mohlo však také dojít ke zcizení majektu organizace. Je tedy žádoucí v případě nevrácení objektu o tom nějakým způsobem informovat zopodvědné osoby, aby mohly na tuto událost včas reagovat.

Jedním z možných přístupů je definování časového úseku, během kterého bude nevrácení objektu tolerováno. Následně je možné využít některého z komunikačních kanálů (e-mail, sms atp.) pro odeslání zprávy zodpovědným osobám nebo majiteli této nevrácené rezervace.

Dále je například možné zablokovat další vypůjčování objektu do doby, než je tento navrácen posledním půjčujícím uživatelem. Každopádně bude nutné umožnit rychlé dohledání uživatele, který měl objekt naposledy půjčen a nevrátil jej.

2.3.6 Lokalizace objektů

Je žádoucí nějakým způsobem poskytnout uživatelům další doplňková data o objektech – kde se objekt fyzicky nachází, jak vypadá a jak se z němu dostat. Tyto informace budou jistě velmi cenné zejména pro ty uživatele, kteří rezervace těchto objektů provádějí nepravidelně, nejsou stálou součástí organizace nebo jsou v ní nováčky. Zejména poloha budov, místností nebo objektů, které se často pohybují mezi více uživateli, se jeví jako velmi užitečná. Údaje by měly být co nejrychleji dohledatelné a nemělo by jich být zbytečně moc, aby se v nich uživatelé neztráceli.

2.4 Shrnutí

Nyní máme vytvořený potřebný obecný model pro to, abychom mohli začít s bližší specifikací požadavků na nový systém. Tento model je včetně již definovaných atributů entit a vzájemných vazeb zobrazen na Diagramu 2.1.

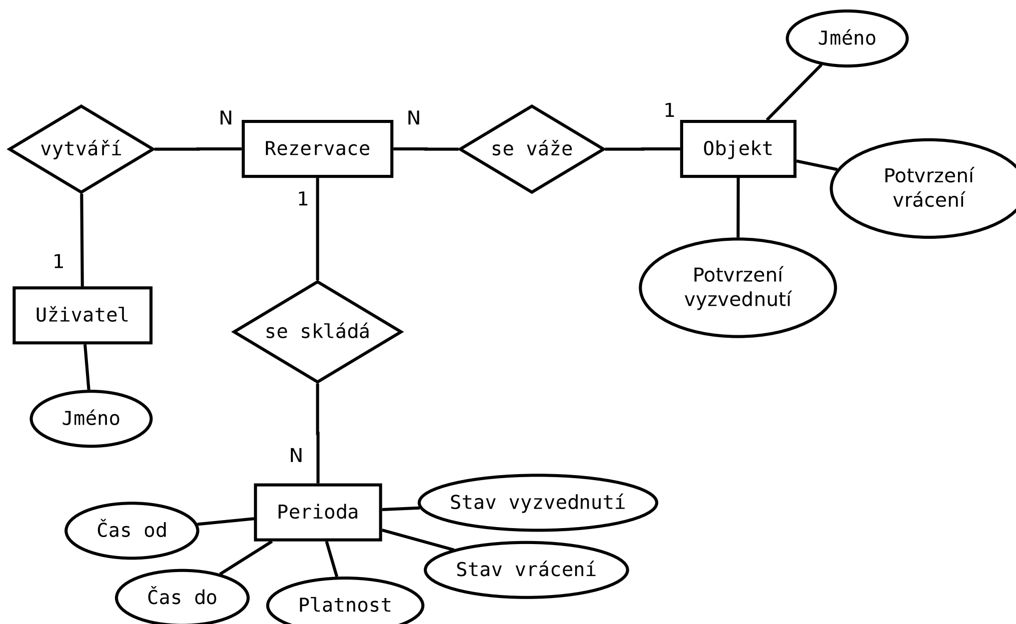


Diagram 2.1: Obecný model rezervací a výpůjček objektů.

K dosažení úplné komplexnosti nám však stále něco chybí – je třeba blíže určit, jaké technické řešení bude použito pro následující cíle:

- autentizace uživatelů.
- identifikace objektů.
- vyzvednutí a vrácení objektu.
- lokalizace objektů.

V následujících kapitolách si ukážeme, jak lze těchto cílů dosáhnout pomocí unikátních technických vlastností mobilních zařízení a pokusíme se definovat základní sadu mobilních technologií, která bude použita pro nový informační systém.

3 Mobilní zařízení

V dnešní době drtivá většina populace vlastní mobilní telefon. V této množině existuje nemalá podmnožina majitelů tzv. chytrých telefonů. Tyto telefony jsou unikátní tím, že využívají pokročilého operačního systému a obvykle mají vlastní aplikační rozhraní, které umožní instalaci programů třetích stran. Jejich konektivita se také neomezuje na pouhé GSM hovory – zcela běžně podporují technologie jako je například WiFi nebo 3G pro rychlý přístup do Internetu.

V posledních letech lze také pozorovat nástup tzv. tabletů. Tablet je zařízení, které sice obvykle neumožňuje komunikaci po klasické GSM síti, disponuje však mnohem větším displejem než chytrý telefon. V poslední době se ukazuje, že chytré telefony a tablety díky své mobilitě, jednoduchosti, delší výdrži na baterii a ergonomickému uživatelskému rozhraní skutečně začínají naplňovat význam termínu „osobní počítač“. Chytré telefony a tablety se stávají mnohem populárnější než klasické stolní počítače a notebooky a to dokonce do té míry, že někteří lidé již ani necítí potřebu pro svou běžnou agendu klasické počítače využívat.

V této kapitole se čtenář seznámí s uživatelskými a technologickými výhodami, které mobilní zařízení přinášejí. Budou rozebrány možnosti vývoje aplikací pro různé mobilní platformy a jejich uplatnění v rámci rozsáhlých informačních systémů.

3.1 Mobilní zařízení jako pracovní nástroj

Současná mobilní zařízení již dávno převzala úlohu různých specializovaných přístrojů jako jsou databanky, Pocket PC nebo palmtopy. Díky tomu, že zpravidla obsahují snadno dostupné aplikace pro plánování a organizaci času a současně kombinují klasickou telefonii s rychlým přístupem na Internet, staly se oblíbenou pomůckou manažerů a dalších zaměstnanců, u nichž je vyžadována okamžitá dostupnost a operativnost.

Nicméně využití mobilních zařízení je mnohem širší. Lze si je představit jako jakési mobilní terminály, přes které zaměstnanci interaktivně komunikují s centrálním systémem. Ten může sloužit pro např. správu skladových zásob, jako databáze zákazníků nebo k objednávání obědů ve školní jídelně – možnosti jsou skutečně široké. Dotykové ovládání je zpravidla intuitivnější než klasické přístupy, navíc menší dipleje nutí vývojáře aplikací k co největší jednoduchosti a kompaktnosti rozhraní – díky tomu může mobilní zařízení po rychlém zaškolení používat skutečně každý.

Je také možné využít různé technologie těchto zařízení, které klasické počítače zpravidla neumožňují, nebo je jejich použití značně neergonomické či pro běžné uživatele složité. Nezanedbatelnou výhodou je také to, že zaměstnanci obvykle mají tato zařízení prakticky pořád u sebe a ve většině případů i připojená k Internetu, což oproti staticky umístěným terminálům nebo webovému rozhraní řádově zvyšuje komfort užívání systému a zvyšuje jeho flexibilitu.

Mezi další možnosti mobilních zařízení v organizaci patří uplatnění tzv. BYOD (*Bring Your Own Device*) politiky. Tento přístup umožňuje v organizacích používat stejné zařízení pro osobní i pracovní účely [4]. IT oddělení nezakazuje používání vlastních zařízení v organizaci, naopak jej aktivně podporuje a snaží se poskytovat maximální podporu pro hladkou integraci zařízení s infrastrukturou. Tento přístup má samozřejmě svá rizika a záleží na každé organizaci zdali používání takových zařízení ve své infrastruktuře umožní, případně zdali nezvolí postup „z druhé strany“ – tedy zdali raději neposkytne zaměstnancům taková zařízení, která by mohli používat i k soukromým účelům.

3.2 Technické prostředky mobilních zařízení

Mobilní zařízení v sobě ukrývají celou řadu zajímavých technologií, se kterými se na klasických počítačích obvykle nesetkáme. Tato kapitola má za úkol seznámit čtenáře s těmito technologiemi a rozebrat jejich klady a zápory.

3.2.1 Lokalizace zařízení

Mobilní zařízení obvykle umožňují lokalizovat svou polohu, a to za použití několika technologií.

GSM lokalizace

Pokud mobilní zařízení podporuje sítě GSM (*Global System for Mobile Communications*), je možné použít tzv. GSM lokalizaci. U této lokalizační metody dochází k přenosu žádosti o lokalizaci z mobilního zařízení do sítě, kde se s využitím dat zjištěných z tohoto zařízení a sítě stanoví jeho poloha. Mobilní zařízení pak zpětně přijme informaci o poloze [3].

GSM síť má buňkovou strukturu, kdy každé buňce je přiděleno (v rámci tzv. *Location Area* unikátní) Cell ID číslo. To slouží pro určení přístupového bodu pro mobilní zařízení. Poloha všech základnových stanic (tzv. BTS – *Base transceiver*

station) vytvářející buňkovou strukturu je operátorovi známa. Poloha uživatele v síti může být stanovena s využitím metody Cell ID s přesností odpovídající velikosti buňky, ve které se zrovna mobilní zařízení nachází. Ta se pohybuje od 100m do 500m v městských oblastech a v jednotkách i desítkách kilometrů mimo města.

Nicméně přesnost lokalizace může být významně zvýšena využitím tzv. *Timing Advance* (zkratkou TA). Tento parametr představuje čas šíření signálu mezi mobilním zařízením a jednotlivými BTS. Při znalosti rychlosti šíření signálu může být určena jejich přibližná vzdálenost. Mobilní zařízení však v praxi přijímá signál od více BTS, což umožňuje výpočet průniku buněk, které tyto stanice vytvářejí. Pro přesné určení polohy je třeba tří (a více) stanic. Dosáhne se tak zvýšení přesnosti určování polohy na hodnoty, která může dosáhnout zhruba 50m.

Tento postup uživatelského určení polohy není díky své malé přesnosti příliš efektivní a v praxi jej nahradily jiné metody lokalizace. Používá se však stále u kombinovaných metod lokalizace k dalšímu zpřesňování polohy zařízení (a je nutný pro vlastní funkčnost GSM sítě).

GPS lokalizace

GPS (anglicky *Global Position System*) patří mezi tzv. globální navigační družicové systémy (anglicky *Global Navigation Satellite System*, zkratkou GNSS). Původně byl vyvinut pro vojenské účely a v současné době je pod kontrolou Ministerstva obrany Spojených států amerických [2]. GPS umožňuje za pomoci přijímače GPS signálu získání třídimenzionálních souřadnic přijímače, jeho aktuální rychlost vzhledem k povrchu Země a také čas. Přijímač může být použit kýmkoliv a kdekoliv na planetě, ve dne i v noci, na souši, na moři i ve vzduchu. Existují samozřejmě i další GNSS systémy: jmenujme například ruský GLONASS, čínský Compass nebo nově vznikající systém Evropské unie - Galileo. V současné době je však GPS zdaleka nejpoužívanější GNSS systém se zdaleka nejširší podporou výrobců lokalizačních zařízení.

GPS funguje díky flotile vysílacích družic. Každá družice je vynesena na přesně stanovenou oběžnou dráhu tak, aby bylo bylo zajištěno, že lokalizační zařízení bude moci z jakéhokoliv místa na Zemi v jakoukoliv denní dobu přijímat signály z nejméně tří družic. Čtvrtý signál družice je pak nezbytný pro synchronizaci hodin přijímače s hodinami družic. Je nezbytné zmínit, že lokalizační zařízení hrají roli pouze pasivních přijímačů družicového signálu.

V současné době jsou GPS přijímače integrovány do celé řady zařízení a ty jsou velmi široce využívány – jmenujme alespoň autonavigace, přístroje pro zabezpečení majektu, sledování osob, geodetické přístroje nebo například systémy určené

k optimalizaci zdrojů, jako je například monitorování dopravy. Většina dnes prodávaných chytrých telefonů a tabletů GPS přijímač obsahuje, a díky možnosti tento přijímač aktivně využívat aplikacemi třetích stran poskytují pro uživatele široké možnosti použití, počínaje navigací, přes nejrůznější inteligentní vyhledávače založené na aktuální poloze zařízení a konče u různých podob sociálních či geolokačních her, jako je například Foursquare nebo Geocaching.

Nevýhody GPS

Asi největší nevýhodou GPS je jeho primární využití pro armádu. Pro civilní využití je dostupná pouze část služeb tohoto systému s omezenou přesností lokalizace a jeho provozovatel může kdykoliv i funkčnost této části omezit nebo ji jednoduše vypnout.

GPS signál má vysokofrekvenční charakter (originální návrh předpokládá dvě nosné frekvence na 1575.42 MHz a 1227.60 MHz, což je již v pásmu UHF). Díky tomu však obtížně prochází některými pevnými materiály, a proto je uvnitř budov pro přijímač díky celé plejádě fyzických bariér složité až nemožné zachytit signál alespoň ze čtyř družic současně a následně stanovit polohu. Navíc se zde mohou nacházet zdroje rušení signálu GPS. Situace se poněkud lepší, pokud se přijímač pohybuje poblíž okna. GPS signál poměrně dobře prochází sklem a tak je možné, že při dobrém výhledu např. v případě výškové budovy s velkými okny bude GPS lokalizace pravděpodobně fungovat. Tento případ je však spíše výjimkou a na GPS zařízení uvnitř budov se obecně nelze spolehnout. Konkurenční GNSS systémy se díky své technologické příbuznosti s GPS potýkají se stejnými obtížemi. Řešením by však mohla být tzv. IPS navigace (viz dále).

Lokalizace IPS

IPS (*Indoor Positioning System*) je navigační systém navržený tak, aby fungoval uvnitř budov. Narozdíl od GPS nevyužívá pozicového signálu družic, ale rovnou několika rozličných technologií, které spolu mohou fungovat i současně. Pod pojmem IPS se neskrývá žádná konkrétní technologie, žádný uchopitelný standard – v současné době probíhá boj o to, kdo první přijde s prvním, široce používaným řešením IPS lokalizace.

Jedna z možných technických implementací je využít sítě Wi-Fi, která je dnes široce rozšířená a naprostá většina mobilních zařízení se k ní dokáže připojit. Dosud uskutečněné realizace byly obvykle založeny na principu indikátorů síly přijatého signálu (RSSI - *Received Signal Strength Indicator*) a seznamu přístupových bodů, ke kterým je mobilní zařízení připojeno [5] [6]. Bohužel tento přístup se ukázal jako

neefektivní – síla signálu se totiž mění v čase velmi nerovnoměrně a pro rychle se pohybující objekty skokově. Problémem také je, že různá zařízení indikují sílu signálu různě, a je tedy velmi těžké z tohoto údaje cokoliv predikovat.

Proto se objevily pokusy, jak detekci polohy zařízení více zpřesnit použitím další technologie široce používané v mobiních zařízeních. Tou technologií je Bluetooth. Například systém uvedený v [7] sítě Wi-Fi používá pouze jako páteřní komunikační linky mezi centrálním serverem a jednotlivými Bluetooth uzly. Ty mají rádiový dosah přibližně deset metrů a jejich rozmístění po budově musí být předem známo. Při vstupu do budovy se uživatel musí připojit k nejbližsímu Bluetooth uzlu – a tím mu předá unikátní 48bit Bluetooth ID svého zařízení. To je uloženo na centrální server. Při požadavku lokalizace tohoto registrovaného zařízení se všechny Bluetooth uzly pokusí připojit k zařízení s již uloženým Bluetooth ID – uzel, kterému se to podaří, odesílá zpět své unikátní ID na server, který poté vyhodnotí polohu hledaného zařízení a odesílá tuto informaci zpět žadateli.

Přesnost tohoto systému je přibližně deset metrů (což odpovídá rozsahu jednotlivých Bluetooth uzlů) a s větším počtem uzlů a jejich menším dosahem by se přesnost jistě dala ještě zvýšit. Velkou nevýhodou tohoto přístupu je ovšem velká náročnost na infrastrukturu sítě.

V poslední době se objevují studie, jak provést lokalizaci uvnitř budov na základě magnetického pole Země [8] [9]. K tomu je použita další, dnes již běžně se vyskytující technologie v mobilních zařízeních – magnetometr neboli kompas. Zjednodušeně řečeno, každý čtvereční centimetr Země vyzařuje magnetické pole – a toto pole je modulováno betonovými a ocelovými konstrukcemi moderních budov. Vestavěné kompasy v mobilních zařízeních jsou obvykle natolik citlivé, aby zaznamenaly změny v magnetickém poli. Pokud je vytvořena mapa těchto magnetických polí, přesná navigace pomocí zabudovaného kompasu je poté velmi jednoduchá. K vytvoření této mapy je opět možné použít vestavěného kompasu. Přesnost těchto systémů je zhruba 1,5m a nikterak výrazně ji neovlivňují ani menší kovové předměty [11]. Tato technologie je také již komerčně využívána [10].

Otázkou zůstává, jak bude magnetické mapování ovlivňováno většími umělými objekty (např. auty, dočasnými kovovými konstrukcemi apod.) a nakolik bude trvalé. Magnetické pole Země se totiž každých několik let mění. Celá oblast IPS navigace tedy zůstává po technické stránce ne zcela uspokojivě vyřešena.

3.2.2 Digitální fotoaparát

Každé mobilní zařízení dnes integruje alespoň jeden digitální fotoaparát (který současně slouží jako videokamera). Použití těchto zařízení je široké, od využití

jako klasického fotoaparátu, přes videohovory až po například bezpečnostní kameru v automobilu. My však zde toto zařízení uvádíme kvůli něčemu jinému – vestavěný fotoaparát lze lehce použít jako integrovanou čtečku optických kódů.

Optické kódy

Optický kód kóduje data popisující objekt, ke kterému je optický kód přiložen. Lze rozdělit do dvou skupin – jednorozměrné (čárové) a dvojrozměrné QR kódy (*Quick Response Codes*). Liší se zejména svojí odolností proti chybám, počtem znaků, který je možno kódovat, a délkou textu, který může kód nést [12]. Tyto kódy byly původně čitelné pouze specializovanými čtečkami a skenery, dnes je však možné pomocí pokročilé analýze obrazu využít i běžně dostupné stolní skenery a fotoaparáty – tedy i ty integrované v mobilních zařízeních. Sejmutí optického kódu vyžaduje přímou viditelnost mezi čtecím zařízením a kódem samotným.

Optické kódy jsou díky své jednoduchosti a finanční nenáročnosti využívány v mnoha oblastech lidské činnosti – v logistice a obchodu, v evidenčních systémech nebo např. slouží k veřejné prezentaci a marketingu. Hlavním využitím je unikátní identifikace objektu, ke kterému je kód přiložen – takový kód budeme dále nazývat jako *tag*.

3.2.3 Rádiové technologie

Mobilní zařízení mohou integrovat technologie, které umožňují komunikaci na krátké vzdálenosti pomocí rádiových vln. V této kapitole budou tyto technologie uvedeny v širším kontextu a poté bude rozebráno jejich konkrétní využití v mobilních zařízeních.

Technologie RFID (*Radio Frequency Identification*)

Dle [13] je RFID technologie umožňující (obdobně jako optické kódy) identifikaci objektů, nicméně nevyžaduje k tomu přímou viditelnost. Přenos dat zde neprobíhá opticky, ale za pomoci rádiových vln. RFID systémy mohou automaticky identifikovat několik objektů umístěných ve stejném prostoru na základě přečtení jejich tagů, a to bez lidské asistence. RFID zařízení lze rozdělit do dvou skupin: *aktivní* a *pasivní*.

Aktivní zařízení vyžadují zdroj napájení, buď z elektrické sítě nebo pomoci baterie. v obvyklém případě je životnost tagu omezena životností baterie (a životnost tagu je měřena v počtu operací čtení, které tag musí spolehnivě dokončit).

Příkladem aktivního tagu může být vysílač v letadle, který identifikuje zemi jeho původu.

Protože jsou však baterie drahé, relativně rozměrné a s omezenou životností, činí aktivní tagy poněkud nepraktickými. Proto existují tagy pasivní. Tyto tagy nevyžadují žádný zdroj napájení. Skládají se ze tří částí: antény, mikročipu připojeného k anténě a ochranného obalu. Strana, která chce tag přečíst, je odpovědná za jeho napájení. Anténa tagu slouží k zachytávání energie a odeslání požadovaného identifikátoru objektu. Poskytnutí tohoto identifikátoru zajistí mikročip.

Existují dva přístupy, jak zajišťovat napájení pasivního tagu – pomocí principu magnetické indukce a elektromagnetických vln. Oba dva poskytují dostatek energie pro napájení mikročipu v pasivním tagu (typicky od $10\mu W$ do $1mW$), liší se však ve vzdálenosti, po kterou mohou tuto energii poskytnout. Díky tomu dále rozlišujeme pasivní zařízení podle dosahu na tzv. *far-field* a *near-field*.

NFC (*Near-field communication*)

NFC je sada standardů, které si kladou za cíl definovat mechanismy, pomocí kterých mohou mobilní zařízení z bezprostřední blízkosti (do 20cm) komunikovat s dalšími přístoji, a současně umožnily číst data ze stávajích pasivních RFID zdrojů. Naprosto záměrně není využito technologií jako je Wi-Fi nebo Bluetooth – jejich rádiový dosah je příliš veliký a vyžadují párování. Je upřednostněna jednodušší konfigurace, byť za cenu nižších rychlostí. NFC technologie (narozdíl od RFID) podporuje šifrování. Mimo klasických RFID tagů NFC podporuje i čtení některých standardů tzv. chytrých karet.

NFC umožňuje několik režimů přenosu ve třech režimech: *Reader/Writer* - čtení nebo zápis do pasivního zařízení, *Peer-to-peer* neboli obousměrná komunikace dvou aktivních zařízení, a režim *Card emulation*, ve kterém je emulováno NFC pasivní zařízení. NFC také nad rámec RFID definuje vlastní sadu tagů. Ty se od sebe liší velikostí paměti, použitým čipem, přenosovou rychlostí a možností přepisování. Technologie už z principu není kompatibilní s *far–field* RFID tagy.

Díky všem těmto vlastnostem je možné tuto technologii uplatnit např. k bezkontaktním platbám, identifikaci, sdílení nebo např. pro rychlé nastavení složitějších druhů komunikace. V současnosti jsou k dispozici desítky typů chytrých telefonů s NFC čipy. Každé zařízení ale ne zcela implementuje všechny předepsané standardy, nebo je naopak rozšiřuje – tím vzniká vzájemná nekompatibilita, kdy některé typy tagů nemusí jít na těchto přístrojích přečíst, nebo nemusí navázat komunikaci s NFC zařízením od jiného výrobce [14].

3.3 Mobilní aplikační platformy

V této kapitole budou představeny nejrozšířenější aplikační platformy mobilních zařízení. Kapitola si neklade za cíl detailně rozebírat klady a nedostatky jednotlivých platforem, dojde však k bližšímu seznámení ze strany vývojářů aplikací pro tyto platformy a budou uvedeny předpoklady a požadavky, které na vývojáře jednotliví vydavatelé platforem kladou.

Pod pojmem „mobilní aplikační platforma" je možné si představit nejen sadu softwarového vybavení pro fyzické mobilní zařízení, včetně např. jádra systému nebo programovému rozhraní pro aplikace třetích stran, ale i třeba základní sadu standardů, které musí koncové zařízení splňovat, nebo definici metod instalace aplikací třetích stran. Protože každý dodavatel platformy si pro svůj produkt nastavil vlastní unikátní prostředí a jen velmi obtížně se stanoví hranice co je definováno platformou a co je již v režii výrobců koncových zařízení, nelze tento pojem příliš dobře zobecnit.

Lze však říci, že platformy se vzájemně liší jak funkcemi poskytovanými operačním systémem uživateli, tak i rozhraním operačního systému pro vývojáře mobilních aplikací a v neposlední řadě aplikacemi třetích stran, které jsou na dané platformě k dispozici.

3.3.1 Stávající mobilní platformy

V současné době lze hovořit o třech nejrozšířenějších platformách – iOS od společnosti Apple, Android a MS Windows Phone. Mezi další, minoritní platformy pak patří BlackBerry OS nebo například Bada od společnosti Samsung.

Apple iOS

Tento mobilní operační systém byl určen pouze pro mobilní telefony iPhone, později se však začal používat i na dalších mobilních zařízeních společnosti Apple, jako jsou iPod Touch nebo tablet iPad. Tento systém byl díky své jednoduchosti používání, orientací na dotykové ovládání a zaměřením na středněproudého zákazníka první svého druhu a odstartoval dnešní éru popularity mobilních zařízení. Obdobné systémy samozřejmě existovaly i před iOS, byly však spíše na okraji zájmu nebo byly primárně orientovány do podnikové sféry.

Společnost Apple plně kontroluje nejen vývoj tohoto uzavřeného systému, ale i kompletní výrobu a distribuci fyzických zařízení a má také ve své režii všechny

oficiální distribuční kanály obsahu zákazníkům, at' už se jedná o multimediální obsah nebo aplikace třetích stran. Použití systému na jiných zařízeních není možné.

Vývoj aplikací je možný v objektově orientovaném jazyce Objective-C, implementovaném jako rozšíření jazyka C. Vývojářské nástroje od programového prostředí až po použitý hardware, na kterém bude programátor aplikaci vytvářet, jsou plně pod kontrolou společnosti Apple.

Windows Phone

Windows Phone je obchodní název mobilního operačního systému společnosti Microsoft. V současné době jsou jako Windows Phone označovány systémy Windows Phone 7 a Windows Phone 8, které však nejsou vzájemně kompatibilní a interně využívají jiné jádro a jinou verzi aplikačního frameworku.

Společnost Microsoft je vydavatelem tohoto systému, sama ale žádné zařízení nevyrábí. Licence systému si kupují jednotliví výrobci mobilních zařízení a nasazují je do svých produktů. Výrobci musí dodržet požadavky na koncové zařízení předepsané společností Microsoft. Patří mezi ně např. použitý mikroprocesor, velikost displeje, uživatelská tlačítka, požadavky na fotoaparát atp. Díky tomu má vydavatel platformy částečnou kontrolu nad koncovým zařízením a také to dává do rukou vývojářů aplikací generickou specifikaci zařízení.

Vývoj aplikací je možný v jazyce C# nebo Visual Basic za pomoci *.NET Frameworku* ve verzi *Compact*, který je přizpůsobený na běh na mobilních zařízeních a jeho vydavatelem je také společnost Microsoft.

3.3.2 Mobilní platforma Android

Pro potřeby této práce se díky zdaleka největší rozšířenosti mezi uživateli, otevřenosti, unifikovanosti a širokým možnostem použití budeme nadále přednostně zabývat platformou Android.

Android je aplikační platforma určená pro mobilní zařízení, vyvíjená společností Google a sdružením výrobců mobilních zařízení (*Open Handset Alliance*) [15]. Zahrnuje operační systém založený na GNU/Linuxu, uživatelské rozhraní, sadu knihoven, podporu multimédií a koncové aplikace. Většina částí této platformy je uvolňována pod ASL (*Apache Software License*) licencí, jádro systému je licencováno pod GPL (*General Public Licence*). Jedná se tedy o open-source platformu.

Jednotliví výrobci mohou přispívat do zdrojového kódu nebo jej modifikovat pro svá zařízení. Andoid je zcela zdarma a kdokoliv si jej může stáhnout a upravit svým potřebám. Díky tomu je tento systém masově rozšířený a dalece přesáhl své původní určení – můžeme se s ním setkat i v digitálních fotoaparátech, televizích, ledničkách nebo i na běžných x86 počítačích.

Architektura

Architektura platformy Android je rozdělena do několika komponent, které se mohou vzájemně částečně prolínat.

Nejnižší vrstva je jádro operačního systému, které je odvozeno z Linuxového jádra. Je plně využito jeho vlastností jako je široká podpora hardwaru, podpora správy paměti, správa sítí, řízení oprávnění a multitaskingu nebo správa procesů. To přispívá ke stabilitě a široké kompatibilitě systému. Naopak jádro neobsahuje moduly pro grafické uživatelské rozhraní X Window System a ani úplný toolkit GNU knihoven.

Jádro je využíváno sadou knihoven napsaných v jazyce C nebo C++, ať už odvozených z jiných projektů nebo napsaných přímo pro Android. Jednou z těchto knihoven je např. standardní knihovna jazyka C, modifikovaná speciálně pro mobilní zařízení, nebo například implementace OpenGL.

Komponenta zajišťující běh uživatelských aplikací se označuje jako *Android Runtime*. Obsahuje virtuální stroj Dalvik (DVM – *Dalvik Virtual Machine*), který je optimalizován pro běh na mobilních zařízeních. Využívá základních vlastností Linuxového jádra, jako je správa paměti nebo práce s vlákny. V této vrstvě jsou také obsaženy základní knihovny programovacího jazyka Java. Knihovny se blíží platformě Java SE, neposkytují však balíky např. pro práci s XML (*eXtensible Markup Language*) nebo AWT – ty byly obvykle nahrazeny vlastní implementací nebo bylo použito různých projektů Apache (např. pro práci se síťovým rozhraním).

Ač by se tak na první pohled mohlo zdát, Dalvik není virtuální stroj určený pro běh klasického Java bytekódu. Překlad aplikace probíhá zkompilováním zdrojového kódu do Java bytekódu pomocí klasického *javac* kompilátoru. Poté se tento bytekód překompiluje pomocí Dalvik kompilátoru a výsledný Dalvik bytekód lze teprve spustit v DVM. Každá spuštěná aplikace běží ve svém vlastním procesu a s vlastní instancí virtuálního stroje.

Komponenta *Application framework* je samotné programové rozhraní pro uživatelskou aplikaci. Zpřístupňuje různé služby platformy, uživatelské rozhraní, upozorňovací stavový řádek, aplikace běžící na pozadí, úložiště dat, hardware použí-

vaného zařízení, sít'ové služby atp.

Android Software Development Kit

Pro vývojáře aplikací určených pro platformu Android je vytvořen balík různých nástrojů souhrnně nazývaný jako *Android SDK*. Je dostupný pro všechny hlavní platformy operačních systémů GNU/Linux, Windows i Mac OS a lze jej integrovat do celé škály programovacích prostředí. Součástí SDK je i emulátor, který umožní testovat aplikace bez přítomnosti fyzického zařízení.

Kompatibilita

Velkou nevýhodou Androidu je jeho „roztříštěnost". Narozdíl od ostatních platforem nevynucuje po výrobcích zařízení a vývojářích dodržování základní myšlenkové linie systému – existuje pouze obecná sada doporučení na hardware a grafické rozhraní. Díky variabilitě koncových zařízení co do výkonu, rozměrů obrazovky a integrovaného vybavení je pro vývojáře velmi těžké vyvíjet aplikace schopné běhu na všech zařízeních. Navíc se programové rozhraní u Androidu s každou jeho verzí velmi rychle mění. Zejména s nástupem tabletů (na které se díky velkým displejům musí uživatelské rozhraní koncipovat odlišně než na chytré telefony) došlo k velkým implementačním změnám uvnitř systému, které se velmi citelně dotkly i aplikačního frameworku.

Naštěstí existuje celá řada postupů, jak se problémům s kompatibilitou alespoň částečně vyhnout. Zejména je možné v manifestu mobilní aplikace specifikovat minimální verzi platformy, na jakou je možné aplikaci spustit. Dále je možné specifikovat například podporovaná rozlišení atp. Majitelům zařízení, která nesplňují nastavené podmínky, se aplikaci vůbec nepodaří nainstalovat, případně se jim vůbec nezobrazí v distribučním kanálu. Rozhraní aplikačního frameworku je zpětně udržováno, takže aplikace pro nižší verzi platformy lze bez problémů spustit na nových zařízeních.

Je také možné vytvářet rozvržení obrazovek aplikace přímo pro několik možných rozlišení. Pokud má aplikace zobrazovat bitmapy, jsou tyto konvertovány do čtyř různých verzí podle úrovně jemnosti a rozlišení displeje tak, aby se systém sám rozhodl, jakou verzi má zobrazit.

Pokud programátor potřebuje používat funkcionalitu vyšší verze a přesto musí být podporována i starší verze platformy, je to možné s použitím tzv. *Support Package Library*. Tato knihovna zpětně portuje funkcionalitu nových verzí Andoidu.

V neposlední řadě je možné využít bohatých možností emulátoru. Tomu lze nastavit širokou škálu možných rozlišení a je možné emulovat některé hardwarové funkce reálného zařízení. Na emulátoru jsou lehce spustitelné všechny existující verze platformy. Díky tomu je vždy možné dostatečně otestovat aplikaci a ujistit se, že se chová a vypadá stejně na široké škále zařízení, aniž by je vývojář fyzicky vlastnil.

3.4 Vývoj mobilních aplikací

Jak vyplývá z předcházejícího přehledu, s každou platformou se pojí zcela jiná sada technologií a využitelných programovacích jazyků. Navíc i různé verze té samé platformy mohou být mezi sebou navzájem nekompatibilní. Možnosti vývoje aplikací pro mobilní platformy jsou navíc limitovány dalšími restrikcemi ze strany vydavatele platformy (např. vývojářské nástroje nebo distribuční kanály aplikace).

3.4.1 Přenositelnost aplikací

Nabízí se otázka, zda by nebylo možné implementovat aplikaci tak, aby byla přenositelná napříč výše zmíněnými platformami. To by umožnilo snížit úroveň duplicity kódu a tím omezit časové a finanční náklady na vývoj. Navíc by nebylo nutné udržovat kód hned v několika jazycích.

HTML5 (*HyperText Markup Language*)

Částečná odpověď na tuto otázku spočívá v nově vznikající specifikaci jazyka HTML. Tzv. HTML5 je standard, který (kromě jiného) bude umožňovat přehrávání multimédií přímo ve webovém prohlížeči a v kombinaci s dalšími technologiemi jako je CSS a JavaScript vytvářet aplikace, které fungují i bez připojení k Internetu nebo k interní síti organizace [16]. Většina současných prohlížečů (včetně těch pro dříve zmíněné mobilní platformy) již HTML5 na alespoň elementární úrovni podporují. Pokud by se podařilo vyvinout aplikaci s využitím na již ustálených a implementovaných části HTML5 standardu, měla by tato aplikace být spustitelná na všech mobilních platformách, kde je dostupný dostatečně sofistikovaný webový prohlížeč.

Tento přístup se však potýká s celou řadou problémů. Takto vytvořená aplikace vyžaduje pro svůj běh další aplikační vrstvu v podobě prohlížeče – interpretera jazyka HTML5. To s sebou nese zvýšené nároky na systémové prostředky zařízení

a s tím související snižování výdrže baterie. Specifikace HTML5 stále není konečná a jeho podpora mezi prohlížeči je velice variabilní. Největší nevýhodou je však to, že aplikace nemá přímý přístup k aplikačnímu frameworku mobilní platformy, což prakticky znemožňuje využít technické prostředky zařízení, jejichž rozhraní jde nad rámec definovaného HTML5 standardu. Pokud je však aplikace dostatečně jednoduchá, je jistě možné služeb technologie HTML5 využít.

Xamarin toolkit

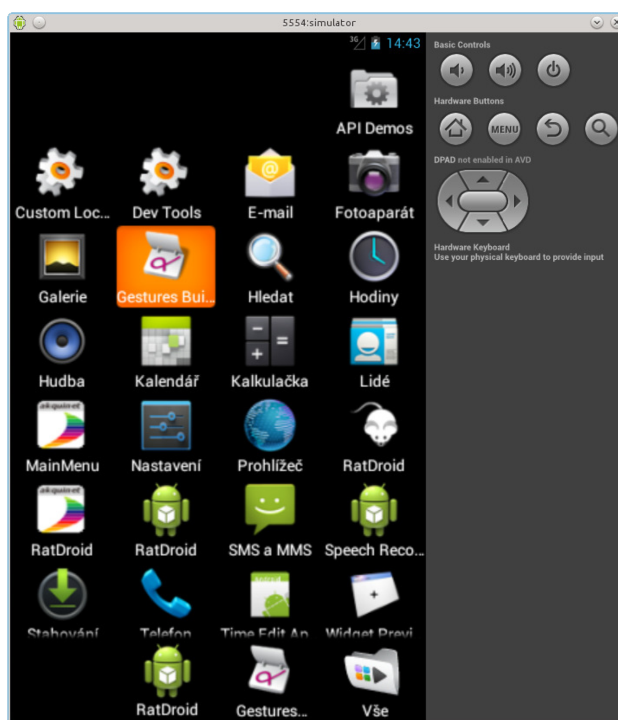
Efektivnějším řešením problému by bylo použití toolkitu společnosti *Xamarin*. Ta se rozhodla převzít projekt Mono, který byl původně vedený společností Novell. Mono je open-source projekt, jehož cílem je vytvořit multiplatformní implementaci *.NET frameworku*. Mono je podporováno na operačních systémech Microsoft Windows, Linux, Unix, OS X, Solaris, BSD, Android, iOS a také na některých herních konzolích. Součástí projektu jsou také nástroje pro podporu vývoje, multiplatformní C# kompilátor a implementace různých dalších knihoven [17].

Jazykem multiplatformní aplikace by se tedy stal C#. Knihovny toolkitu podporují integraci do platforem iOS a Android, do každé trochu jiným způsobem. Ve druhém případě lze zjednodušeně říci, že každá takto vytvořená aplikace pro Android je puštěna ve dvou běhových prostředích – *NET Mono runtime* a DVM, přičemž je využito JNI (*Java Native Interface*) rozhraní pro vzájemnou komunikaci obou prostředí. Protože je *Mono runtime* implementován v jazyce C, může velmi jednoduše volat systémové služby Linuxového jádra Androidu nebo využívat přidružených knihoven (např. pro OpenGL grafiku).

Technologie má samozřejmě své limity (nelze použít žádné Java knihovny, pouze částečná podpora Java generických tříd, některá standardní rozhraní nelze implementovat) a nezaručuje stoprocentní přepoužitelnost kódu, nicméně představuje zajímavou možnost jak vyrobit skutečně multiplatformní mobilní aplikaci. Toolkit je bezplatně uvolněn pouze k omezenému využití, další funkcionalitu je třeba dokoupit.

3.4.2 Vývojové prostředí

Vývoj aplikací pro mobilní zařízení se ve své elementární formě nijak zásadně neliší od klasického programování „plnohodnotných“ desktopových aplikací. Vydavatel platformy poskytuje sadu standardních vývojářských nástrojů, integrované vývojové prostředí a emulátor, na kterém je možné aplikace spouštět a debuggovat. Na Obrázku 3.1 je snímek obrazovky běžícího emulátoru mobilní platformy Android.



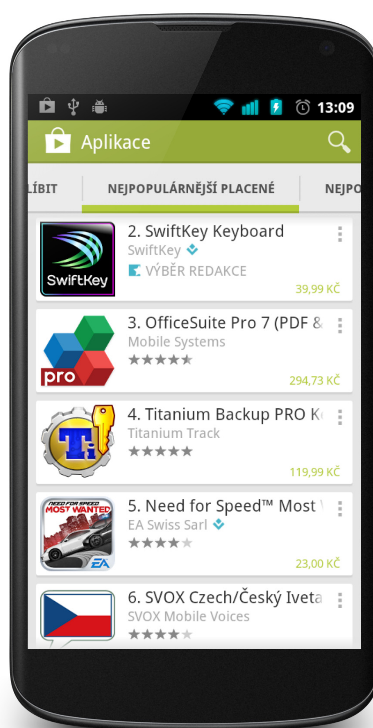
Obrázek 3.1: Běžící emulátor platformy Android.

Spouštět aplikace je samozřejmě možné i s fyzickým zařízením (zejména proto, že emulátor z principu nemůže emulovat všechny funkce zařízení jako je gyroskop, telefonní hovory atp.). Někteří vydavatelé ovšem omezují využití fyzických zařízení (např. Microsoft zavedl maximální počet takto nainstalovaných aplikací a podmiňuje použití reálného zařízení jeho online registrací).

Někdy je také možné využít vývojářské prostředí třetích stran, což se řídí licenčními podmínkami vydavatelů platformy. Kupříkladu u Androidu je možné využít hned několika možných prostředí nebo si i vytvořit prostředí vlastní. Nejobvyklejší volbou je zpravidla instalace zásuvného modulu do široce používaného vývojového prostředí Eclipse. Modul umožňuje nejen správu fyzických zařízení a emulátorů, ale i přímou instalaci a debuggování aplikace a integruje tak *Android SDK* toolkit a samotné vývojové prostředí.

3.4.3 Distribuce aplikací

Je obvyklé, že vydavatel platformy povoluje distribuci aplikací pouze přes svůj oficiální centrální distribuční kanál – např. na Obrázku 3.2 je ukázka aplikace určené pro distribuci a správu uživatelských aplikací pro mobilní platformu Android.



Obrázek 3.2: Ukázka distribuce aplikací pro platformu Android.

Vývojář mobilní aplikace musí zpravidla mít vytvořený vývojářský účet u vydavatele platformy a svou aplikaci neposkytuje přímo cílovým uživatelům, ale nahrává ji vždy přes tento účet do distribučního kanálu. Někteří vydavatelé neposkytují tyto účty zdarma nebo participují na ziscích z prodeje aplikace.

Tento model má několik výhod – uživatelé vidí všechny aplikace pro své zařízení na jednom místě. Pokud je aplikace placená, je finanční transakce uskutečněna vždy jen mezi zákazníkem a vydavatelem platformy – ten se poté postará o rozdělení této částky. Je obvyklé že tyto kanály se starají i o aktualizace aplikací. V neposlední řadě distribuční kanály částečně řeší i otázku bezpečnosti a jisté kvality aplikací – vydavatel platformy má kontrolu nad obsahem distribučního kanálu.

Na druhou stranu může být tento model až zbytečně omezující. To, jestli bude možné instalovat aplikace i z jiných zdrojů, záleží čistě na vydavateli platformy.

3.5 Shrnutí

Seznámili jsme se s širokými možnostmi mobilních zařízení, ať již po stránce využitelnosti v rámci informačních systémů, tak po stránce technologické. Byly představeny tři nejrozšířenější mobilní platformy s důrazem na možnosti vývoje aplikací pro tyto platformy a jejich následnou distribuci.

4 Požadavky na systém správy rezervací

Tato kapitola má na základě případové studie na Katedře informatiky a výpočetní techniky při Západočeské Univerzitě v Plzni a předcházejícího zobecnění problému výpůjček a rezervací objektů za cíl navrhnout kompletní specifikaci požadavků na nový systém výpůjček a rezervací. Nejprve dojde k rozboru využitelnosti mobilních technologií pro tento systém. Následně bude definována přesná sada funkcí, které musí nový systém implementovat.

4.1 Využitelnost mobilních zařízení

V Kap. 2.4 jsme definovali výčet základních požadavků na nový systém, aniž bychom definovali jejich technologický podklad. Nyní se pokusíme spojit tyto požadavky s technologiemi mobilních zařízení tak, abychom maximálně využili výhody, které nám tyto technologie nabízejí.

4.1.1 Možnosti přístupu a autorizace

Je možné si představit několik způsobů technického řešení přístupu do systému za pomoci mobilních zařízení a s tím související autorizace uživatelů.

Jeden terminál pro více uživatelů

V organizaci jsou na předem daných místech rozmístěny tablety, umožňující autorizaci uživatele a následnou rezervaci objektu. Tyto terminály jsou vybaveny technologií NFC. Uživatel, který chce provést rezervaci, přistoupí k tabletu, přiloží svůj telefon vybavený NFC technologií a díky unikátnímu klíči se přihlásí do systému.

Výhody jsou zřejmé – díky NFC šifrování je identifikátor nezjistitelný třetí stranou. V organizaci již může existovat princip ověřování uživatelů na principu přístupových karet a v tom případě lze za předpokladu, že je NFC technologie se standardem karet navzájem kompatibilní, autentizaci sjednotit se stávající infrastrukturou (přístupové karty by hrály roli NFC tagu s údaji o uživateli). Uživatelé si také nemusí pamatovat další heslo.

Autorizace přes NFC má však také své nevýhody – použití kompatibilních přístupových karet nelze paušalizovat, což se neslučuje se základním požadavkem poskytnout co nejuniverzálnější řešení systému. Mobilních zařízení vybavených NFC technologií je stále příliš málo na to, aby bylo možné omezit autorizaci jen na ně.

Pro ověřování by se daly teoreticky použít i čárové nebo QR kódy. Každý uživatel by u sebe nosil visačku s vytištěným kódem a pouze by ji přiložil k integrovanému fotoaparátu tabletu. Toto levné a jednoduché řešení (každý by si svůj unikátní kód mohl svépomocí opakovaně vytisknout) ovšem naráží na nepřekonatelnou bariéru – tyto kódy nelze nijak šifrovat. Pro jednoduchou autorizaci by jistě postačily, ale jejich snadná reprodukovatelnost, čitelnost a nulová možnost zabezpečení toto řešení předem zavrhuje.

Nevýhodou celého modelu je také jeho statičnost. Není žádoucí nutit uživatele, aby kvůli rezervování místnosti ležící v přízemí chodili do pátého patra budovy, protože je tam je umístěný tablet s aplikací umožňující vytvoření rezervace. Pokusíme se tedy spojit mobilní zařízení se systémem ještě úžeji.

Jeden terminál pro jednoho uživatele

V tomto modelu provádí uživatelé rezervace na zařízeních, která mají fyzicky u sebe – tedy na svých osobních telefonech a tabletech. Zde se již jeví autorizace pomocí NFC nebo optických kódů jako zcela zbytečná. Není žádného důvodu proč nevyužít interní úložiště zařízení a neautorizovat se pomocí jména a hesla, přičemž tyto údaje by byly paušálně vyžadovány pouze při prvním startu aplikace nebo při neúspěšném připojení do systému. Tyto údaje by mělo být možné kdykoliv změnit.

Předcházející model se s tímto přístupem přitom nijak nevylučuje – je možné poskytnout další, webové rozhraní pro ty uživatele, kteří patřičné zařízení nevlastní, nebo je z nějakého důvodu odmítají používat. V tomto webovém rozhraní by se uživatelé autorizovali stejným jménem a heslem jako na mobilních zařízeních s tím rozdílem, že by mělo být umožněno se ze systému kdykoliv odhlásit. Toto webové rozhraní je poté možno provozovat kdekoliv jako statický terminál a uživatelé se k němu budou moci připojovat ze svých počítačů přes webový prohlížeč. Portál bude také možné využít pro administrátory systému ke správě objektů, uživatelů a rezervací.

V rámci co největšího uživatelského komfortu by také mělo být možné využít již pravděpodobně existující databázi uživatelů a ověřovat zadaná jména a hesla proti této databázi. Uživatelé by poté nemuseli spravovat duplicitní přihlašovací údaje.

4.1.2 Identifikace objektů

Pro vytvoření rezervace je nejprve třeba přesně určit, k jakému objektu rezervaci vytváříme. Díky mobilním zařízením máme k dispozici víc než jen obvyklou sadu možností (např. fulltextové vyhledávání, databázové dotazy apod.) – můžeme využít NFC technologii nebo optické kódy. Obě technologie dovolí identifikaci řádově urychlit.

V prvním případě jsou objekty fyzicky spárovány s NFC nebo RFID pasivními tagy. Uživatel vytvářející rezervaci pouze přiloží zařízení vybavené NFC k tagu a prakticky okamžitě získá identifikátor objektu, se kterým lze poté dále pracovat. Výhodou je zejména objem dat, který se do tagu vejde – je možné do tagů uložit i další informace o objektu (polohu, inventární číslo apod.) a ty prohlížet libovolnou NFC čtečkou. Nezanedbatelná je také trvanlivost a odolnost pasivních tagů a možnost jejich snadného umístění.

Bohužel, ke všem nevýhodám NFC, které již byly zmíněny, se přidávají i poměrně vysoké náklady za pořízení tagů a jejich nesnadná replikovatelnost. Protože tagy musí být kvůli snadné orientaci umístěny na dobře viditelném místě, jistě by se po čase staly cílem nenechavců.

Řešením je využití optických QR kódů. Do těchto kódů je možné (narozdíl od kódů čárových) uložit obdobný objem dat jako do NFC tagů, ovšem nepojí se s nimi žádné fyzické zařízení. Díky tomu je možné QR kódy libovolně replikovat a měnit jejich velikost, lepit je po zdech, vkládat je do hlaviček dokumentů, nástěnek, popisků dveří, štítků notebooků atp. Pokud je štítek poškozen, je možné při čtení využít autoopravných algoritmů, které umožňují i z velmi poškozeného obrazu získat původní data. Při nenávratném zničení štítku stačí pouze vytisknout na obyčejné tiskárně štítek jiný. Uživatel místo NFC čipu využije prostého fotoaparátu, a tím se použitelnost mobilních zařízení při vytváření rezervace řádově rozšíří. Nevýhodou oproti NFC je pak nižší odolnost nosného materiálu a pomalejší doba čtení – díky obvykle nízké kvalitě integrovaných fotoaparátů a potřeby analyzovat jejich obraz dalšími algoritmy je čtení QR kódů obvykle o něco pomalejší.

Může dojít k tak velkému fyzickému poškození štítku, že již nebude čitelný a nový nebude momentálně k dispozici. Pro tyto případy by měla mobilní aplikace umožnit i klasické vyhledání objektu podle textového řetězce, případně nabídnout možnost vytvořit si seznam často používaných objektů. Tato funkcionalita uživatelům mimo jiné umožní rezervovat si libovolný objekt, aniž by byli nuceni skenovat QR kód.

4.1.3 Vyzvednutí a vrácení objektu

Pro vyzvedávání a vracení objektů můžeme použít již definované postupy pro identifikaci objektů. Uživatel může na svém osobním zařízení objekt buď skenovat, nebo využije jinou metodu pro identifikaci. Poté dojde k samotné akci vyzvednutí nebo vrácení.

4.1.4 Lokalizace objektů

Lokalizace objektů v reálném čase není snadná. U nepohyblivých objektů mimo budovu je možné určit jejich zeměpisnou polohu na základě GPS přijímače a tu poté distribuovat spolu s popisem objektu, zakódovat ji do QR popisu atp. Uživatelé si poté mohou snadno zobrazit polohu objektu na svém mobilním zařízení v integrovaných mapových podkladech.

U pohyblivých objektů jistě můžeme obdobně definovat jejich obvyklou polohu v době, kdy nejsou nikomu vypůjčeny. Jak ovšem zařídit sledování objektů v reálném čase, tedy i v čase vypůjčení uživatelům? Osazení všech objektů GPS lokátory je finančně neúnosné. Bylo by možné sledovat objekty na základě polohy zařízení, ze kterých byly tyto objekty vyzvednuty. To se však rovná sledování polohy majitele zařízení a tím pádem se tento přístup potýká s celou řadou problémů:

- uživatelé musí dát provozovateli systému souhlas se sledováním.
- mobilní zařízení se nesmí vzdálit od sledovaného objektu.
- uživateli stačí vypnout GPS navigaci nebo připojení na Internet a sledování je ztraceno.
- sledování nebude fungovat uvnitř budov.

Tyto problémy činí efektivní sledování objektů velice obtížným.

Řešení uvnitř budov by mohlo být využití pasivních RFID *far–field* tagů, které by byly fyzicky svázány s objekty. To by v kombinaci čteček těchto tagů umístěných na všech rámech dveří a dalších strategických místech umožnilo jakési sledování polohy – čtečky by měly unikátní identifikátor a byly by propojeny s centrálním serverem, který by znal polohu čteček. Při každém průchodu tagu kolem čtečky by tato odeslala na server svůj identifikátor a identifikátor objektu přečtený z RFID tagu. Tím by se dosáhlo alespoň přibližné polohy. Bohužel, nejenže by bylo možné tento systém velmi jednoduše obejít, byl by také finančně nákladný a náročný na

infrastrukturu. Dříve zmíněné možnosti IPS navigace se zdají být pro účely této práce jen obtížně využitelné.

Zvolíme proto jiný přístup. Objekty budou sloučeny do skupin – tříd, které je budou nějakým způsobem popisovat a seskupovat. Způsob sloučení objektů zůstane záměrně nedefinován – může tak být z důvodu geografického (všechny objekty v budově A), technického, na základě podobnosti objektů, na základě stejného dodavatele – konkrétní důvody pro sloučení budou přenechány na administrátorech systému. Údaj o lokaci poté bude moci možné uvést v popisu objektu, nebo v popisu třídy. Většina mobilních zařízení syntaxi geografickýh souřadnic rozpozná a v případě nutnosti nabídne jejich zobrazení v interních mapách.

4.1.5 Shrnutí

Z analýzy využitelnosti mobilních zařízení pro systém správy výpůjček a rezervací vyplynuly pro účely této práce následující obecné požadavky na koncový systém.

Uživatelé budou moci na svém mobilním zařízení využívat aplikaci pro vytváření a editaci rezervací objektů. Z mobilních technologií bude využit fotoaparát jako čtečka QR kódů a přístup k Internetu. Každý objekt bude mít unikátní QR kód, jehož naskenováním budou moci uživatelé objekt jednoznačně identifikovat. Systém nabídne vyhledávání objektů a editovatelný seznam „oblíbených“ objektů pro každého uživatele.

Vedle této aplikace bude spuštěn i webový portál se stejnou funkcionalitou s výjimkou skenování QR kódů. Portál bude sloužit nejen uživatelům pro správu vlastních rezervací a seznamu oblíbených objektů, ale i administrátorům pro správu objektů, uživatelů a jejich rezervací.

Objekty budou slučovány do tříd podle logických pravidel definovaných administrátory systému. Jak objekty, tak třídy objektů poskytnou možnost dodatečného popisu, ve kterém mohou být libovolné informace včetně zeměpisných souřadnic.

4.2 Obecný popis systému

Předmětem tohoto popisu a následné specifikace je systém pro realizaci rezervací movitých a nemovitých objektů pomocí centrálního serveru s webovým rozhraním a mobilních klientů.

4.2.1 Účel systému

Systém slouží k realizaci časové rezervace objektů. Objekty se rozumí jak věci movité (jako například notebook, projektor atp.), tak i věci nemovité. Díky tomuto systému bude možné na Katedře informatiky a výpočetní techniky při Západočeské Univerzitě v Plzni zefektivnit stávající procesy vypůjčování a rezervace těchto objektů, rozšířit oblast použití i na další objekty co do typu a množství a decentralizovat správu objektů, výpůjček a rezervací.

4.2.2 Rozsah systému

Uživatelé se připojují k centrálnímu serveru systému pomocí klientské aplikace nebo prostřednictvím webového rozhraní. Autorizace uživatelů systému probíhá zadáním kombinace unikátního jména a hesla. Součástí systému bude databáze uživatelů. Systém bude implementovat dva způsoby autorizace - oproti lokální databázi systému a pomocí účtu systému Orion.

Po připojení je možné vytvářet nebo editovat rezervace objektů a uskutečňovat jejich výpůjčky. Rezervace je tvořena identifikátorem rezervovaného objektu a časovým rozpětím, po které si uživatel chce objekt rezervovat. Identifikátor bude realizován jako textový popisek nebo jako QR kód fyzicky umístěný na objektu. Objekty budou seskupeny do tříd. Výsledkem činnosti systému je kalendář s rezervacemi, který bude zobrazen pro každého uživatele a pro každý objekt v klientské aplikaci a ve webovém rozhraní.

4.2.3 Omezení návrhu a implementace

Pro správnou funkci celého systému je nutné připojení k internetu. Serverová část bude implementována v programovacím jazyce Java. Systém bude poskytovat webové rozhraní. Klientská mobilní aplikace bude implementována pro platformu Android.

4.2.4 Třídy uživatelů

V systému budou definovány tři třídy uživatelů:

- *Uživatel* – typicky zaměstnanec katedry nebo její spolupracovník, který si

potřebuje půjčovat objekty z inventáře katedry nebo provádět jejich rezervace pro např. výukové účely.

- *Správce rezervací* – zaměstnanec sekretariátu katedry nebo jiná osoba odpovědná za udržování pořádku v rezervacích a výpůjčkách majetku.

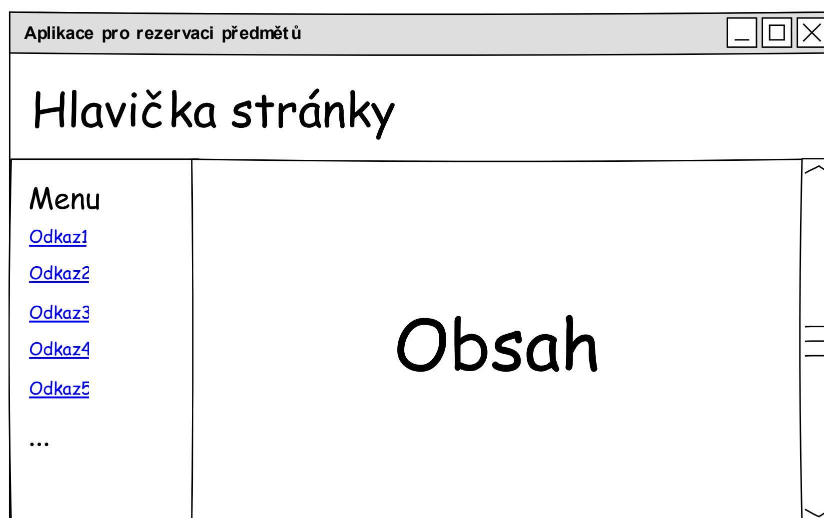
- *Administrátor* – zaměstnanec sekretariátu katedry, správce sítě katedry nebo jiná osoba odpovědná za udržování informačního systému katedry.

4.3 Funkce webového rozhraní

V této tapitole dojde k přesnému vymezení systému a definici funkcí požadovaných od webového rozhraní.

4.3.1 Základní rozvržení webového rozhraní

Rozvržení webového rozhraní se bude skládat z hlavičky s názvem aplikace a hlavního menu umístěného na levé straně obrazovky (Obr. 4.1). V menu se bude zobrazovat konstantní seznam hypertextových odkazů. Při kliknutí na odkaz se požadovaná stránka zobrazí v pravé části vedle menu.



Obrázek 4.1: Základní rozvržení webového rozhraní.

4.3.2 Autorizace uživatelů

Do webového rozhraní bude umožněno přihlášení pomocí uživatelského jména a hesla a následné úplné odhlášení (Diagram 4.1). Bez přihlášení nebude umožněna žádná další interakce se systémem a bude zobrazena stránka s přihlašovacím formulářem. Odkaz pro odhlášení bude přímou součástí hlavního menu aplikace. V případě neaktivity bude uživatel automaticky odhlášen.

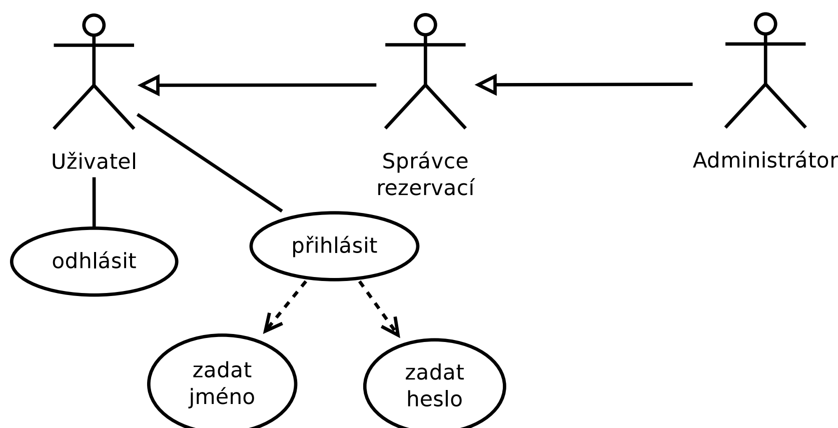


Diagram užití 4.1: Autorizace uživatelů přes webové rozhraní.

4.3.3 Správa uživatelů

Pro všechny role budou viditelné profily ostatních uživatelských účtů. Administrátor bude moci vytvářet, mazat a editovat všechny uživatelské účty v systému (Diagram 4.2).

Uživatelský účet se bude skládat ze jména, příjmení, emailu, definice role v systému, přihlašovacího jména a poskytovatele autorizace účtu. Všechny položky musí být při vytváření nebo editaci profilu vyplněny. Při smazání účtu uživatele musí být odstraněny i všechny s účtem související rezervace.

Editace profilu a změna hesla

Pokud je poskytovatel autorizace účtu nastaven na interní databázi, je při vytváření účtu vyžadováno i zadání hesla. Tímto heslem poté budou uživatelé systému autorizováni. Všechny role budou moci měnit své vlastní heslo, pouze Administrátor bude oprávněn měnit hesla i ostatních účtů (Diagram 4.3).

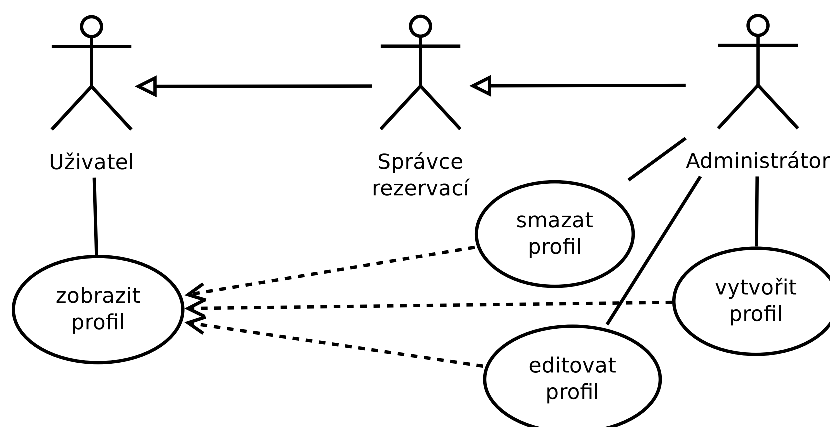


Diagram užití 4.2: Zobrazení a editace uživatelských účtů.

Všechny role také mohou měnit položky svého vlastního profilu s výjimkou poskytovatele autorizace a definice role. Při editaci profilu musí zůstat unikátní přihlašovací jméno uživatele.

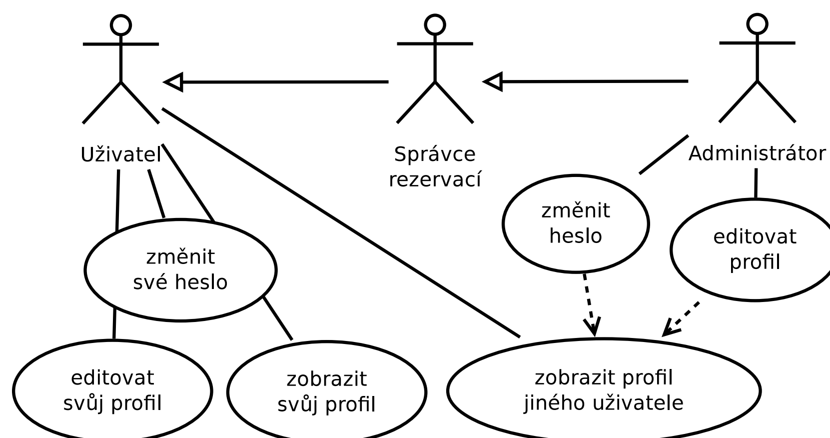


Diagram užití 4.3: Správa uživatelů.

4.3.4 Správa tříd objektů

Administrátor bude moci přidávat, editovat a mazat třídy objektů (Diagram 4.4). Třída objektů se skládá z názvu třídy a popisné informace. Název třídy je povinná položka. Třída bude definovat, zda bude nutné potvrzovat vyzvednutí nebo vracení

všech objektů, které budou přiřazeny do této třídy. Při odstranění třídy objektů musí být automaticky vymazány i všechny objekty této třídy.

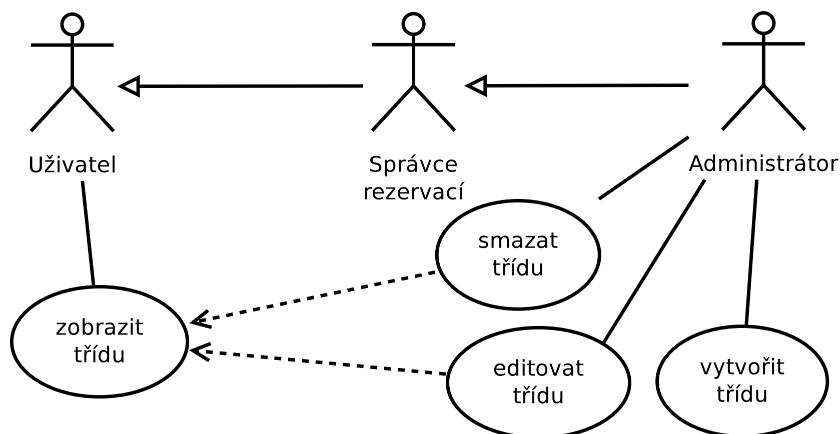


Diagram užití 4.4: Správa tříd objektů.

4.3.5 Správa objektů

Administrátor bude moci přidávat, editovat a mazat objekty (Diagram 4.5). Objekt je definován názvem, popisem a unikátním textovým identifikátorem. Identifikátor nebude možné po vložení objektu do systému v rámci editace měnit. Při odstranění objektu ze systému musí být vymazány i všechny s objektem související rezervace.

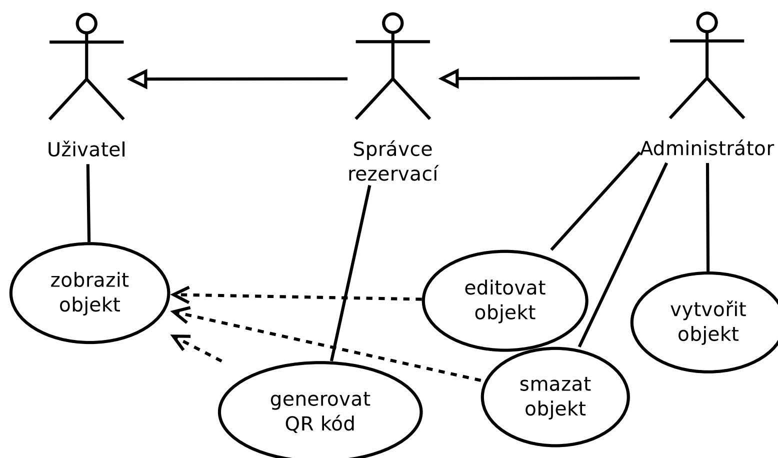


Diagram užití 4.5: Správa objektů.

Všechny objekty bude možné zobrazit v tabulce a tuto tabulku bude možné filtrovat na základě textového vyhledávání nebo tříd objektů (Obrázek 4.2).

Obrázek 4.2: Základní rozvržení správy objektů.

Export identifikátoru do QR kódu

Administrátorům a Správcům rezervací bude umožněn export identifikátoru objektu do podoby QR kódu. Výstupní formát bude PDF dokument se třemi stejnými QR kódy o různých rozměrech, textovou reprezentací identifikátoru objektu a názvem objektu.

4.3.6 Správa oblíbených objektů

Všem rolím bude umožněno udržovat si svůj seznam objektů, se kterými často manipulují. Do tohoto seznamu bude možné jeho vlastníky libovolně přidávat objekty, nebo je následně odebírat (Diagram 4.6). Seznam těchto objektů bude přístupný z hlavního menu webového rozhraní.

4.3.7 Vytváření rezervací

Všem rolím bude umožněno vytvářet vlastní rezervace. Správcům rezervací a Administrátorům bude navíc umožněno vytvářet rezervace za jiné uživatele (Diagram 4.7).

Rezervace je definována počátečním a koncovým termínem rezervace a objektem, ke kterému se rezervace vztahuje. Počáteční ani koncový termín nesmí ležet v minulosti (vztaženo k času vytvoření rezervace). Pokud uživatel zadá počáteční termín ležící v minulosti, systém musí při vytváření rezervace zajistit posunutí

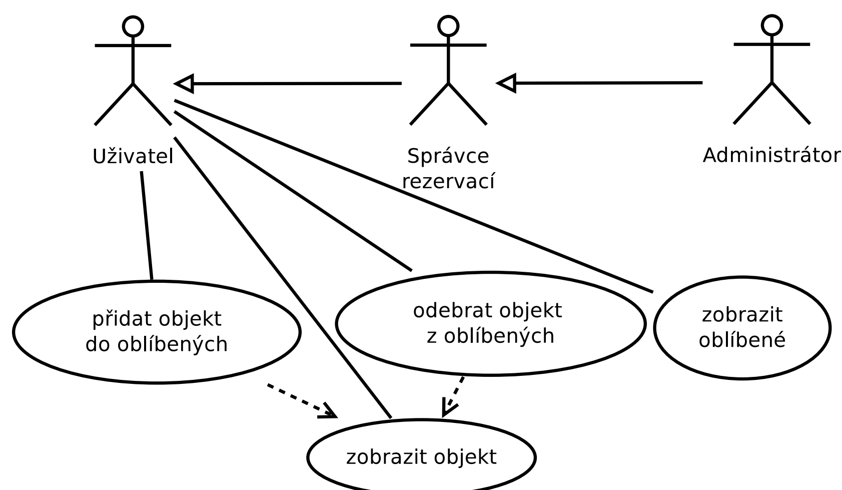


Diagram užití 4.6: Správa oblíbených objektů.

počátku rezervace do přítomnosti. Systém musí odmítnout vytvoření rezervace, kde je koncový termín umístěn v čase před počátečním termínem. Rezervace objektu může být vytvořena pouze v případě, pokud neexistuje jiná platná rezervace na tento objekt ve stejném časovém rámci nebo rámci, který se s úsekem nové rezervace částečně překrývá.

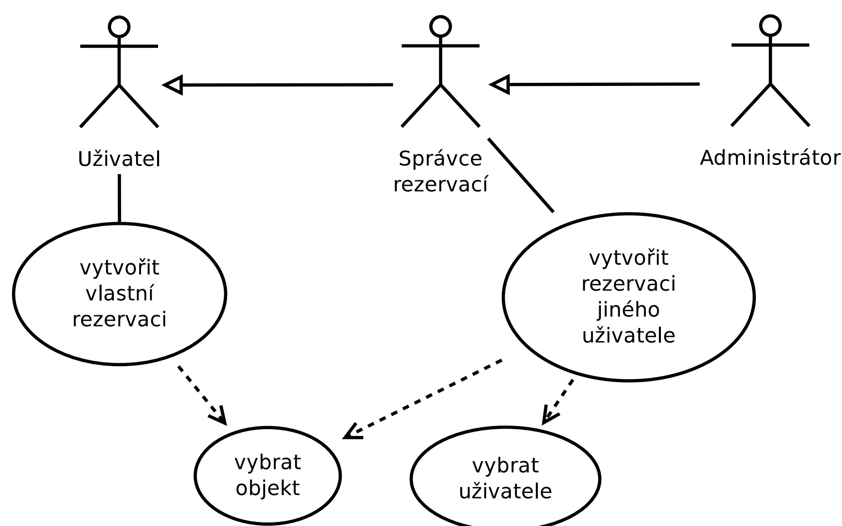


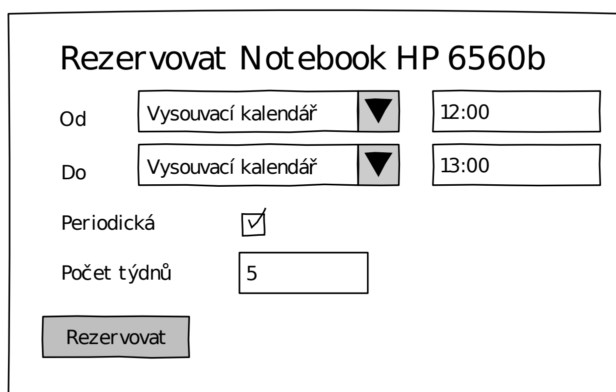
Diagram užití 4.7: Vytváření rezervací.

Na centrálním serveru bude umožněna konfigurace minimální a maximální přípustné délky rezervace. Pokud uživatel zadá při vytváření rezervace kratší časový

úsek než je minimální přípustná délka rezervace, systém musí rezervaci prodloužit na tuto minimální délku. Pokud uživatel zadá časový úsek delší než je maximální délka rezerace, systém musí vytvoření takové rezervace odmítnout.

Periodické rezervace

Rezervace objektů bude možné po pevně daném časovém úseku opakovat. Tento úsek je s ohledem na potřeby katedry stanoven na **jeden týden**. Při vytváření rezervace bude možné zadat počet týdnů, po který bude rezervace opakována (Obrázek 4.3). Termín první rezervace je čas počátku rezervace, který uživatel zadal do formuláře pro vytvoření rezervace.



Obrázek 4.3: Rozvržení vytvoření periodické rezervace.

Maximální počet opakování rezervace bude konfigurovatelný na centrálním serveru a systém nesmí umožnit překročení tohoto limitu. Délka časového úseku rezervace nesmí překročit délku jednoho týdne.

4.3.8 Správa rezervací

Všem rolím bude umožněno editovat a mazat vlastní rezervace. Správcům rezervací a Administrátorům bude navíc umožněno editovat a mazat rezervace jiných uživatelů, nebo měnit uživatele rezervací (Diagram 4.8).

Editovat bude možné čas vyzvednutí a čas vrácení. Pro editaci těchto termínů budou platit stejná pravidla jako pro vytváření nové rezervace. Editovat rezervace ležící v minulosti (oproti času editace) nebude možné.

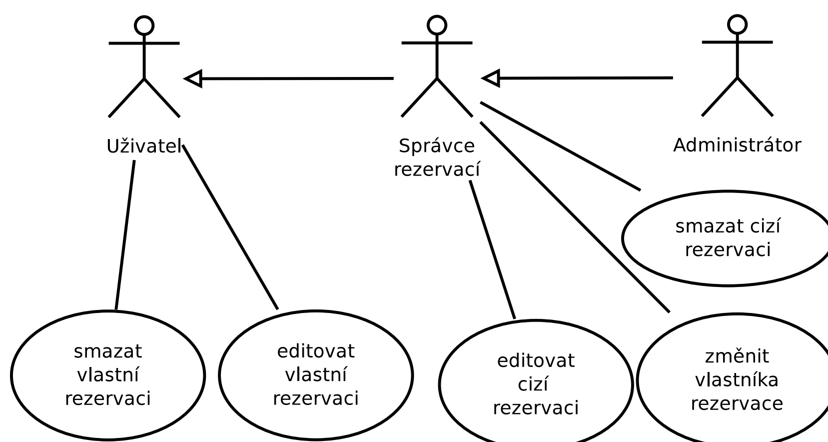


Diagram užití 4.8: Editace rezervací.

Platnost rezervací

Další editovatelnou vlastností rezervací bude tzv. „platnost rezervace". Každá nové vytvořená rezervace bude automaticky vedena jako platná. Uživatům bude umožněno zneplatnit rezervaci v rámci její editace. Systém bude interpretovat neplatné rezervace tak, jako by v daném časovém úseku žádná rezervace nevznikla (bude tedy možné v tomto úseku vytvářet jiné rezervace). Neplatná rezervace bude však stále pro uživatele viditelná a bude možné jí opět učinit platnou za podmínky, že v jejich časovém úseku v okamžiku editace neexistuje jiná platná rezervace na stejný objekt.

Editace periodických rezervací

Při editaci periodických rezervací musí být zachován konstantní průběh rezervace. Pokud tedy dojde ke změně termínu rezervace, tato změna se musí promítnout do všech týdenních úseků rezervace s výjimkou těch úseků, které leží v minulosti oproti času editace. Bude možné zneplatňovat jednotlivé úseky periodické rezervace, aniž by ostatní úseky byly touto editací jakkoliv zasaženy.

Pokud dojde k odstranění periodické rezervace, budou odstraněny i všechny její časové úseky.

4.3.9 Vyzvedávání rezervací

Všem rolím bude umožněno vyzvedávat vlastní rezervace. Administrátoům a Správcům rezervací bude umožněno vyzvedávat rezervace za jiné uživatele (Diagram 4.9). Vyzvednutí rezervace bude systémem povoleno pouze tehdy, pokud objekt svázaný s rezervací náleží třídě, která vynucuje potvrzení vyzvednutí svých objektů. Dále musí být splněny následující podmínky:

- rezervace musí být nevyzvednutá.
- rezervace musí být platná.
- pokud čas počátku rezervace leží v budoucnosti oproti času pokusu o vyzvednutí rezervace, musí být jejich rozdíl menší než minimální délka rezervace objektu.
- pokud čas počátku rezervace leží v minulosti oproti času pokusu o vyzvednutí rezervace, nesmí čas vrácení objektu ležet v minulosti (nelze vyzvednout rezervace, jejichž časový úsek již skončil).

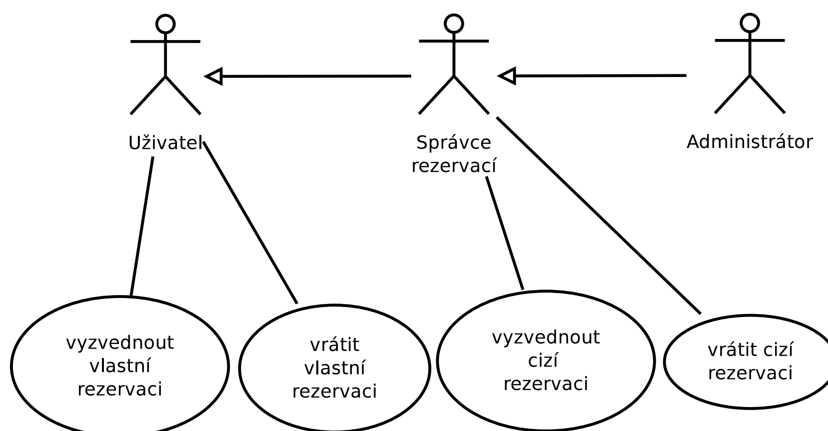


Diagram užití 4.9: Vyzvedávání a vracení rezervací.

4.3.10 Vracení rezervací

Všem rolím bude umožněno vracet vlastní rezervace. Administrátorům a Správcům rezervací bude umožněno vracet rezervace za jiné uživatele (Diagram 4.9). Vrácení rezervace bude systémem povoleno pouze tehdy, pokud objekt svázaný s rezervací náleží třídě, která vynucuje potvrzení vracení svých objektů. Dále musí být splněny následující podmínky:

- pokud rezervovaný objekt náleží třídě vynucující vyzvednutí, rezervace musí být vyzvednutá.
- rezervace musí být platná.

Pokud čas konce rezervace leží v budoucnosti oproti času vrácení rezervace, musí systém změnit čas konce rezervace na tento čas vrácení.

4.3.11 Expirace rezervace

Pokud rezervace vyžaduje potvrzení vyzvednutí a majitel rezervace nevyzvedne rezervaci před časem začátku této rezervace, systém umožní vytvoření jiné platné rezervace ve stejném časovém úseku nebo změnu časového rámce jiné rezervace do rámce původní nevyzvednuté rezervace za předpokladu, že od času začátku doby rezervace uběhla **doba expirace rezervace**.

Doba expirace bude konfigurovatelná na centrálním serveru. Pokud dojde k překročení doby expirace a je vytvořena nebo přesunuta jiná rezervace do stejného časového rámce, musí systém zajistit zneplatnění původní vyexpirované rezervace.

4.3.12 Kalendář

Rezervace budou organizovány do kalendáře rezervací. Kalendář rezervací bude možné zobrazit jak pro objekty tak pro každého uživatele. Editací kalendáře se rozumí jakákoliv editace rezervací zobrazených v kalendáři. Tato editace podléhá dříve popsaným pravidlům pro editaci rezervace.

Kalendář uživatele

Všem rolím bude umožněno zobrazit a editovat svůj kalendář a zobrazit kalendář ostatních uživatelů. Správci rezervací a Administrátoři systému budou moci editovat kalendáře jiných uživatelů (Diagram 4.10). Na kalendář aktuálně přihlášeného uživatele bude možné se dostat přímo z hlavního menu.

Jak je naznačeno na Obr. 4.4, jednotlivé rezervace se budou od sebe barevně odlišovat v závislosti na aktuálním stavu rezervace. Schéma bude následující:

- **žlutě** budou vybarveny všechny validní rezervace, jejichž časový úsek leží

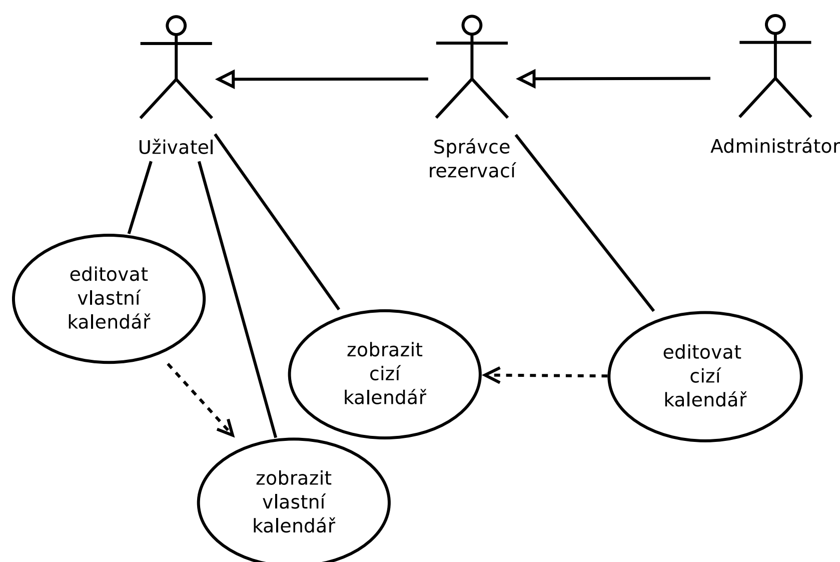


Diagram užití 4.10: Kalendář uživatele.

v budoucnosti a nevyžadují vyzvednutí, nebo probíhají v aktuálním čase a nevyžadují vrácení (rezervace, které jsou aktivní a nevyžadují žádnou akci).

- **modře** budou označeny validní rezervace, jejichž časový úsek leží v budoucnosti a vyžadují vyzvednutí (aktivní rezervace, které vyžadují akci).

- jako **zelené** budou označeny rezervace, které byly vyzvednuty, vyžadují vrácení a jejich koncový čas leží v budoucnosti (právě probíhající rezervace, které budou v budoucnu vyžadovat akci vrácení).

- **červeně** budou znázorněny rezervace, které vyžadují vyzvednutí a nebyly uživatelem včas vyzvednuty, nebo vyžadují vrácení a nebyly uživatelem včas vráceny.

- **šedé** rezervace jsou neplatné rezervace, nebo rezervace které byly úspěšně vráceny nebo nevyžadují potvrzení vrácení a jejich časový úsek leží v minulosti (neaktivní rezervace, již nevyžadují žádnou další akci).

Krátké popisky popisující aktuální stav rezervace budou zobrazeny i při zobrazení profilu rezervace. Ten by mělo být možné zobrazit přímo z kalendáře.

Obrázek 4.4: Rozvržení kalendáře uživatele.

Kalendář objektu

Kalendář objektu bude mít obdobné rozvržení jako kalendář uživatele. Všem rolím bude umožněno přidávat a editovat své rezervace. Správci rezervací a Administrátoři systému budou v kalendáři moci editovat rezervace všech uživatelů nebo za ně jejich rezervace přidávat a mazat.

Barevné rozlišení bude jiné než v případě kalendáře uživatele. **Zelené** rezervace budou patřit uživateli, který prohlíží kalendář objektu. Všechny ostatní rezervace budou **šedé**.

Export kalendáře

Kalendář každého uživatele nebo objektu bude možné exportovat do formátu iCal. To bude možné buď jednorázově exportem do textového souboru, nebo pomocí veřejně dostupné URL unikátní pro každého uživatele a objekt. Tyto URL budou zobrazeny v profilech uživatelů a objektů.

4.4 Funkce klientské aplikace

Klientská aplikace narozdíl od webového rozhraní nenabídne žádné možnosti administrace uživatelů, rezervací, objektů nebo tříd objektů. Všechny případy užití systému dosud definované pro role Správce rezervací a Administrátor nebudou v mobilní aplikaci dostupné. Tyto dvě role se tedy připojují k systému přes mobilní aplikaci v roli Uživatele.

Přibližné rozvržení jednotlivých obrazovek a diagram průchodu mobilní aplikací jsou uvedeny v Příloze A.

4.4.1 Hlavní obrazovka

Po startu aplikace a úspěšném přihlášení k serveru bude zobrazena hlavní obrazovka s navigačními tlačítky na obrazovky (Obr. A.2) obsahující základní funkce aplikace:

- *Můj kalendář* – zobrazení kalendáře uživatele.
- *Oblíbené* – seznam oblíbených objektů.
- *Vyzvedni* – funkce pro vyzvednutí rezervace.
- *Vrat'* – funkce pro vrácení rezervace.
- *Prohlížet* – prohlížeč objektů.
- *Nastavení* – globální nastavení připojovacích direktiv.

Na rozdíl od webového rozhraní bude mobilní aplikace koncipována pro převážné používání jedním uživatelem. Zadání jména a hesla při každém startu aplikace nesmí být vyžadováno.

4.4.2 Nastavení

Aplikace bude obsahovat obrazovku s globálním nastavením připojení k centrálnímu serveru, jehož součástí budou i položky pro jméno a heslo uživatele.

Aplikace bude rozpoznávat změny nastavení a po jejich uložení se pokusí připojit na server. V případě neúspěšného připojení o tom musí notifikovat uživatele a současně zakázat navigaci na všechny obrazovky aplikace s výjimkou nastavení připojení.

4.4.3 Oblíbené položky

Aplikace poskytne obrazovku se zobrazením všech oblíbených objektů uživatele, obdobně jako na webovém rozhraní. Po vybrání jednoho z objektů dojde k automatickému přepnutí na profil objektu.

4.4.4 Profil objektu

Na obrazovce s profilem objektu se budou zobrazovat všechny informace vztažené k objektu – jméno, identifikátor a popis objektu, název třídy objektu a informace, zdali je vyžadováno vyzvednutí a vracení objektu. Dále budou dostupné následující akce:

- *Přidat do oblíbených* – zobrazeno v případě, pokud objekt nebude v seznamu oblíbených objektů.
- *Odebrat z oblíbených* – zobrazeno, pokud objekt v tomto seznamu bude přítomen.
- *Kalendář objektu* – navigační tlačítko na kalendář objektu.
- *Vyzvedni* – funkce pro vyzvednutí objektu (viz dále).

4.4.5 Prohlížeč objektů

Aplikace poskytne obrazovku se zobrazením seznamu objektů. Bude možné zobrazit objekty všechny oblíbené položky přihlášeného uživatele a objekty vyhledané podle textového vstupu od uživatele nebo podle vyfotografovaného QR kódu (Obr. A.5). Po vybrání jednoho z objektů ze seznamu bude uživatel automaticky přesměrován na obrazovku s profilem objektu.

Vyhledávání objektů

Vyhledávání objektů bude možné na základě jména objektu, popisu objektu, jména třídy nebo všech těchto kritérií najednou. Pokud bude některý objekt vyhovovat více kritériím, musí se v seznamu objevit pouze jednou.

Pokud uživatel zvolí vyhledávání podle QR kódu, aplikace automaticky aktivuje fotoaparát zařízení a spustí detekci QR kódu. Po úspěšném rozpoznání se zobrazí objekt, který odpovídá vyfotografovanému identifikátoru.

4.4.6 Kalendář

Aplikace bude zobrazovat obdobný formát kalendáře jako v případě webového rozhraní. Přihlášenému uživateli bude přímo z hlavní obrazovky dostupný jeho osobní kalendář rezervací.

Rozvržení kalendáře

Rozvržení kalendáře musí být optimalizováno pro zobrazení na malých displejích a musí maximálně využívat dotykového ovládání.

Kalendář bude reprezentován dvěma obrazovkami. Nejprve přijde uživatel do styku s měsíčním kalendářem (Obr. A.4). Ten musí umožňovat přechod mezi jednotlivými měsíci a zobrazovat počet rezervací na každý den v měsíci (v případě, kdy nebude existovat žádná rezervace, nebude tato informace zobrazena). Aktuální den bude zvýrazněn.

Při rozkliku dne v měsíčním kalendáři se zobrazí posunovatelný denní kalendář s jednotlivými rezervacemi (Obr. A.3). Vizualizace a barevná schémata rezervací budou shodná s webovým rozhraním. Kalendář musí podporovat stejné operace s rezervacemi, jaké jsou dostupné pro roli Uživatele ve webovém rozhraní systému.

Kalendář objektu

Aplikace také umožní vizualizaci kalendáře objektu, opět se shodným barevným schématem a možnými akcemi a restrikcemi definovanými pro roli *Uživatele* v rámci webového rozhraní systému. Bude tedy možné vytvářet a měnit vlastní rezervace a prohlížet rezervace jiných uživatelů. Tento kalendář bude dostupný z profilu každého objektu.

4.4.7 Správa rezervací

Obrazovky pro vytváření a editaci rezervací budou dostupné z kalendáře uživatele i objektu a nabídnou stejnou sadu položek a stejnou funkcionalitu jako je definována pro webové rozhraní (Obr. A.6). V obou obrazovkách bude maximálně využito prostředků dotykového displeje a grafických uživatelských komponent. Na editační obrazovce bude nabídnuta možnost rezervaci úplně smazat ze systému. Dále bude v případě editace periodické rezervace nabídnuta možnost měnit platnost všech časových úseků rezervace, aniž by je bylo nutné dohledávat v kalendáři. Změna

platnosti časového úseku rezervace bude realizována jako výběr ze seznamu všech časových úseků rezervace.

Jakékoliv změny uskutečněné v rámci správy rezervací se musí okamžitě po úspěšném odeslání promítnout do všech dotčených kalendářů. V případě, že nebude možné změnit rezervaci z důvodu blokace této změny jinou rezervací, je nutné zobrazit zprávu se jménem majitele této blokující rezervace.

4.4.8 Funkce „Vyzvedni"

Tato funkce bude zajišťovat vyzvedávání objektů na základě existujících rezervací nebo případné vytváření nových rezervací. Navigační tlačítka zpřístupňující tuto funkci budou dostupná přímo z hlavní obrazovky a z obrazovky profilu objektu.

V případě, že uživatel zvolí přístup z hlavní obrazovky aplikace, musí být nejprve zajištěna selekce objektu určenému k vyzvednutí. Přístup bude obdobný jako v případě prohlížení objektů – bude možné vybrat objekt ze seznamu oblíbených objektů, ze seznamu objektů vyhledaných podle určitých kritérií nebo bude možné objekt identifikovat skenováním QR kódu. V případě přístupu z profilu objektu je tento identifikován samotným profilem. Tato funkce nebude dostupná pro objekty, které nevyžadují potvrzení při vyzvednutí.

Po identifikaci objektu musí systém vyhledat rezervaci vytvořenou na tento objekt a splňující podmínky vyzvednutelnosti uvedené v požadavcích na webové rozhraní.V případě nalezení takové rezervace systém zajistí její vyzvednutí a informuje o tom uživatele. Pokud taková rezervace není nalezena, bude umožněno vytvoření rezervace nové, jejíž počátek bude v aktuálním čase vyzvednutí. Koncový čas bude zvolen uživatelem. Rezervace bude automaticky označena jako neperiodická, validní a vyzvednutá. Pro vytvoření této rezervace budou platit stejné podmínky jako pro vytváření všech ostatních rezervací (minimální a maximální délka atp.) a v případě nesplnění těchto podmínek o tom musí být uživatel informován.

4.4.9 Funkce „Vrať"

Tato funkce bude zajišťovat vracení objektů na základě existujících nevrácených rezervací. Bude dostupná z hlavní obrazovky aplikace.

Uživateli se bude zobrazovat seznam všech nevrácených rezervací, a to jak aktuálně běžících, tak rezervací nevrácených v minulosti. Nebudou se zobrazovat rezervace, které potvrzení vrácení nevyžadují. Po vybrání jedné z rezervací dojde k jejímu vrácení a návratu na hlavní obrazovku aplikace.

4.5 Mimofunkční požadavky

4.5.1 Komunikace se serverem

Komunikace s centrálním serverem bude zabezpečena HTTPS certifikátem. Bude žádoucí, aby autorizace uživatelů mobilních aplikací a s tím související přenos citlivých údajů probíhal co možná nejméně. Jakékoliv chyby komunikace mezi mobilní aplikací a serverem se musí zobrazit uživateli. Rychlost a odezvy komunikace musí korespondovat s pohodlným používáním mobilní aplikace a webového rozhraní (maximálně jednotky vteřin).

4.5.2 Požadavky na mobilní aplikaci

Všechny obrazovky bude možné rotovat podle uživatelských preferencí. V případě překrytí jinou aplikací musí obrazovky korektně obnovit svůj stav i za cenu opakovaného stažení dat z centrálního serveru. Aplikace musí být maximálně ovladatelná pomocí dotykového displeje a využívat komponent grafického uživatelského rozhraní, které jsou již zabudovány v systému. Nevyžaduje se přenositelnost aplikace z Androidu na jiné mobilní platformy.

4.5.3 Požadavky na výkon systému

Na centrálním serveru musí být možné uložit řádově desítky uživatelů a objektů bez ztráty rychlosti přístupu. Server musí zvládnout narůstající počet rezervací v čase bez znatelné ztráty výkonu. Systém musí bez problémů obsloužit současné přihlášení jednotek až desítek uživatelů.

5 Možnosti návrhu webových aplikací

V této kapitole budou rozebrány možnosti návrhu a implementace dříve specifikovaného systému pro správu rezervací a uvedeny některé využitelné technologie založené na platformě Java. Čtenář se seznámí s obecnými architektonickými principy uplatnitelnými při vytváření informačních systémů a distribuovaných webových aplikací.

5.1 Vícevrstvá architektura

Vícevrstvá architektura je architektonický vzor, ve kterém se model aplikace skládá z více vzájemně spolupracujících vrstev. Sousedící vrstvy komunikují přes předem definovaná rozhraní a díky tomu mohou být zaměňovány, aniž by to mělo dopad na funkčnost celé aplikace nebo bylo nutné měnit ostatní vrstvy. Přenos dat a definování vzájemného rozhraní mezi vrstvami je součástí návrhu vícevrstvé architektury.

Pravděpodobně nejznámějším zástupcem uplatnění vícevrstvé architektury je referenční OSI (*Open Systems Interconnection*) model. Jeho původním účelem byla snaha o standardizaci komunikace uvnitř počítačových sítí [18], dnes se však používají jeho jednodušší odvozeniny (např. protokol TCP/IP) a původní model má spíše metodický význam. Je však příkladem, že vícevrstvá architektura se nemusí omezovat pouze na čistě softwarovou implementaci.

Model se skládá ze sedmi vrstev. Nejvyšší vrstva v modelu zajišťuje přístup aplikací k síťovému komunikačnímu systému. Pokud klientská aplikace vytvoří požadavek na přístup do sítě, je tento předáván sestupně v zásobníku vstev až do vrstvy nejnižší, která zajišťuje přímý přístup k fyzickému médiu. Každá vrstva má přesně vymezenou sadu úkolů jako je šifrování, navazování spojení, synchronizace, směrování atp. a má jednoznačně definováno rozhraní mezi sousedními vrstvami v zásobníku.

Na tom lze názorně demonstrovat výhody vícerstvých architektur. Díky tomu, že se každá vrstva omezuje na předem definovanou sadu úkolů a každá pracuje s jinou úrovní abstrakce, zbývá jen malý krok k vytvoření standardizovaných protokolů pro každou vrstvu. Jednotlivé implementace těchto protokolů bude poté možné libovolně vyměňovat bez ovlivnění funkcionality ostatních vrstev. Pokud se ukáže, že současná implementace je již příliš složitá, je možné vrstvy dále dělit.

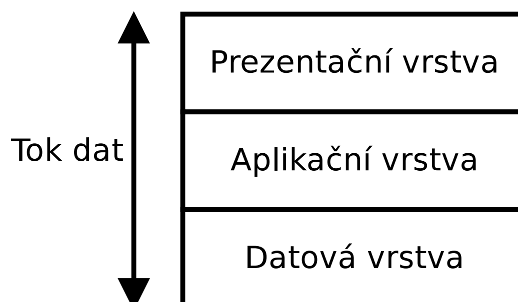
Tyto lze dokonce i slučovat, jak se také stalo ve výše zmíněné od OSI modelu odvozené architektuře TCP/IP, která využívá pouze čtyř vrstev.

5.1.1 Zásady pro návrh vícevrstvého modelu

Architekt aplikace by měl navrhnout novou vrstvu všude tam, kde je zapotřebí jiný stupeň abstrakce. Každá vrstva by měla zajišťovat přesně vymezené funkce zvolené tak, aby pro jejich realizaci mohly být vytvořeny standardizovaná rozhraní nebo protokoly. Počet vrstev by měl být tak velký, aby vzájemně odlišné funkce nemusely být zařazovány do stejné vrstvy, a současně s tím tak malý, aby celá architektura zůstala dostatečně přehledná. Možné budoucí komplexní přepracování vrstvy nesmí zásadně ovlivnit sousední vrstvy. Rozhraní mezi vrstvami by měla být zvolena tak, aby byl minimalizován tok dat.

5.1.2 Třívrstvá architektura

Třívrstvá architektura je obecný typ vícevrstvé architektury. Je s oblibou využívána v kombinaci s modelem distribuované aplikace klient-server. Rozděluje aplikaci do tří separátních logických vrstev, které nemusí nutně běžet na stejných zařízeních nebo operačních systémech. Rozdělení architektury je zjednodušeně znázorněno na Obr. 5.1.

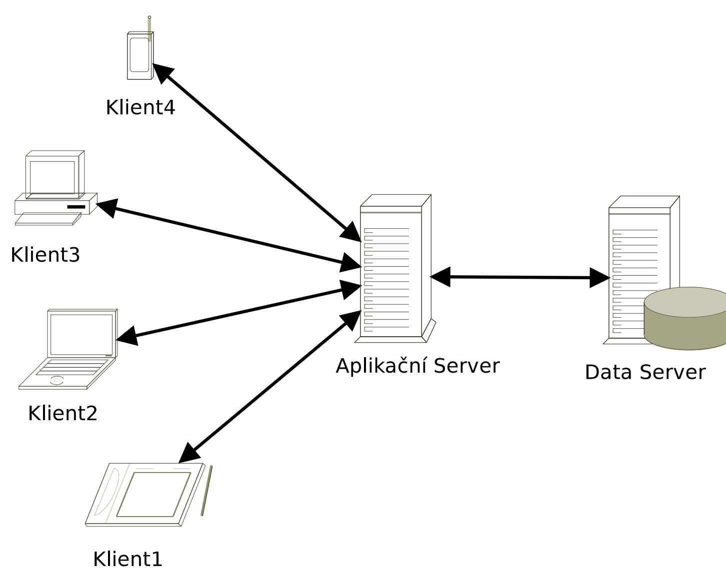


Obrázek 5.1: Základní schéma třívrstvé aplikace.

Prezentační vrstva zajištuje vizualizaci dat pro uživatele a interakci s uživatelem [20]. Aplikační (doménová) vrstva obsahuje samotné jádro aplikace, funkce pro výpočty nebo zpracování dat. Datová vrstva se poté stará o persistenci dat. Je nutné zmínit, že v čistě třívrstvé architektuře se tok dat děje vždy jen mezi sousedícími vrstvami, tzn. při přístupu z prezentační vrstvy nemůže být prostřední vrstva překročena a opačně. Úkolem třívrstvé architektury není definování rozhraní mezi

vrstvami nebo technologického základu pro třívrstvé aplikace, klade si za cíl pouze rozvržení logického modelu aplikace.

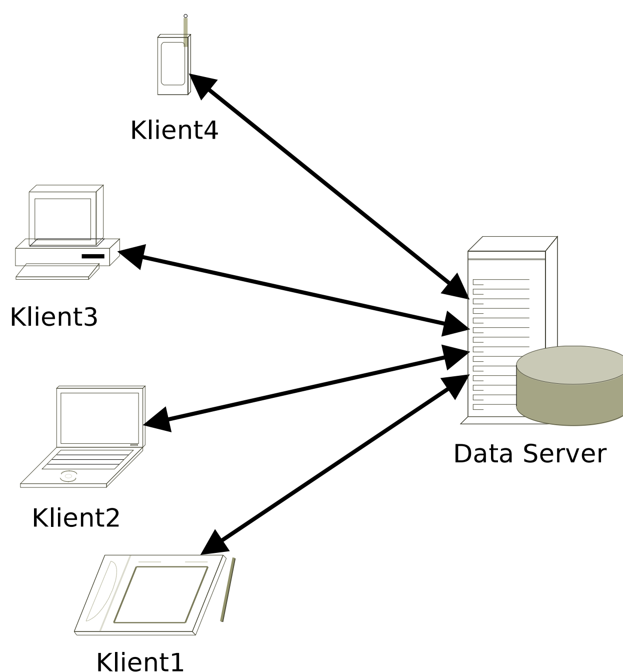
Na Obrázku 5.2 je znázorněno typické rozvržení třívrstvé aplikace. Datová vrstva je zde reprezentována relační databází na vzdáleném serveru. K této databázi se připojuje aplikační server, zajišťující výpočetní jádro systému. K aplikačnímu serveru se připojují jednotliví klienti – na těchto zařízeních jsou data vizualizována a je umožněno uživatelům těchto klientů s daty pracovat. Díky existenci aplikačního serveru tito klienti neobsahují žádnou další aplikační logiku a hrají roli tzv. *tenkých klientů*.



Obrázek 5.2: Příklad třívrstvé architektury (*tenký klient*).

Nicméně díky obecnosti třívrstvé architektury je možné si představit celou řadu jiných scénářů. Kupříkladu relační databáze může fungovat přímo na aplikačním serveru nebo může úplně chybět a roli datové vrstvy sehraje např. implementace transformující data do XML souborů. Na Obr. 5.3 je pak znázorněna situace, kde aplikační server zcela chybí. Aplikační vrstva může být poté přítomna jako součást relační databáze v podobě uložených procedur a tzv. *triggerů* (viz dále), nebo mohou jednotliví klienti s sebou nést i aplikační logiku – stanou se tzv. *tlustými klienty*. Je možné tyto postupy i kombinovat. Je možné (a dnes již celkem obvyklé), aby část aplikační logiky ležela na aplikačním serveru a část kódu se spouštěla na jednotlivých klientech.

Využití třívrstvé architektury je tedy velice flexibilní. Při návrhu systému si softwarový architekt musí velmi dobře rozmyslet celkové rozvržení jednotlivých logických vrstev s ohledem na výkon, bezpečnost, udržitelnost, technické vlastnosti a možnosti klientských zařízení a další rozšiřitelnost navrhovaného systému. Je také



Obrázek 5.3: Příklad třívrstvé architektury (*tlustý klient*).

nutné vždy jednoznačně definovat komunikační rozhraní mezi vrstvami podle zásad návrhu vícevrstvých aplikací tak, aby bylo možné jednotlivé vrstvy zaměnit nebo je standardizovat.

Nyní si detailněji rozebereme jednotlivé části a další architektonické vzory třívrstvé architektury s ohledem na možné využití při implementaci distibuovaných webových aplikací.

5.2 Aplikační vrstva

Aplikační (nebo také doménová) vrstva je taková vrstva, do které by mělo být soustředěno maximum výkonného kódu a vlastní logiky aplikace. Existuje celá řada obecných architektonických vzorů uplatnitelných při návrhu této vrstvy, nicméně zejména u webových aplikací se vrstva obvykle rozdělí na *doménový model*, ve kterém jsou definovány základní entity aplikace a jejich vzájemné vazby, a *servisní vrstvu*, která s tímto modelem manipuluje nebo vytváří jeho rozhraní [20].

5.2.1 Doménový model

Doménový model vytváří jakousi sít' objektů navzájem spojených vazbami, kde každý objekt obvykle reprezentuje nějakou entitu v reálném světě. Návrhem doménového modelu by měl začínat jakýkoliv návrh implementace aplikace a jeho programové realizaci obvykle předchází vytvoření analytického doménového modelu za využití jazyka UML (*Unified Modeling Language*).

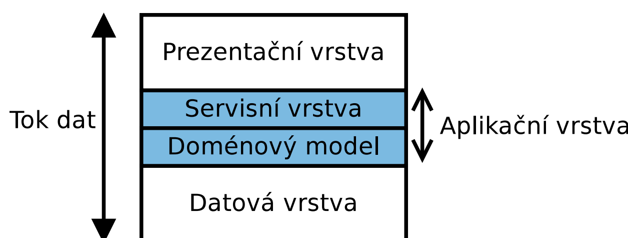
Objekty doménového modelu mají sadu atributů definovanou tak, aby se co nejvíce přiblížily svému reálnému vzoru. Každý objekt doménového modelu by však měl být jen tak velký, jak je to bezpodmínečně nutné. V případě příliš komplexního objektu je vhodné ho rozdělit na menší objekty a tyto propojit vazbami. Kvůli snadné rozšiřitelnosti a udržitelnosti je důležité mít neustále možnost během životního cyklu aplikace snadno doménový model měnit a testovat. Dále je nutné udržovat co nejméně vazeb modelu na ostatní části systému (což je mimo jiné cílem mnoha návrhových vzorů).

Ve světě Javy jsou obvykle jednotlivé doménové objekty realizovány pomocí POJO (*Plain Old Java Object*) objektů. Definice POJO vznikla jako reakce na nepříliš flexibilní EJB2 (*Enterprise Java Bean*), které se váží na použitý framework a nesplňují tak požadavek na co nejnižší počet vazeb modelu na okolní programové prostředí. Pro POJO neexistuje žádný uchopitelný standard, nicméně lze je definovat jako instance každé *Java Beany*, která se snaží minimalizovat své vazby vůči okolí at' už po stránce anotací, dědičnosti nebo implementace rozhraní. To samozřejmě není (například v případě spolupráce aplikace s jinými frameworky) téměř nikdy možné, nicméně zejména podmínka neexistující dědičnosti od nějaké komponenty užitého frameworku by měla být splněna.

Pod pojmem *Java Bean* se poté chápe taková třída, k jejímž členským proměnným se přistupuje pouze pomocí veřejných *getterů* a *setterů* a tyto dodržují jisté jmenné konvence. Pro přístup k jednotlivým členským proměnným se díky jednoduchosti POJO může využít pokročilých programovacích technik, jako je aspektové programování a reflexe. Obdobná situace je i v jiných programových prostředích (např. v *.NET Frameworku*), i když syntax kódu a užitá terminologie se samozřejmě liší.

5.2.2 Servisní vrstva

Jak ukazuje Obr. 5.4, servisní vrstva je v hierarchii vrstev umístěna na vyšší úrovni abstrakce než je doménový model a současně vytváří programové rozhraní (*Application Programming Interface*, zkratkou API) pro prezentační vrstvu. Jinak než skrze servisní vrstvu nelze s doménovým modelem pracovat.



Obrázek 5.4: Základní rozdělení aplikační vrstvy.

Servisní vrstva je díky tomu, že tvoří „úzké hrdlo“ v toku dat, dobrým místem nejen pro umístění aplikační logiky, ale i zabezpečení nebo kontroly dat. Podle [20] existují dva přístupy, jak vytvořit servisní vrstvu. První z nich je tzv. **Doménová Fasáda**. Ta principiálně deleguje aplikační logiku do doménového modelu a tvoří pouze zmiňované rozhraní mezi prezentační a aplikační vrstvou a celou servisní vrstvu tvoří pouze množina *fasád* nad doménovým modelem. Fasády obsahují pouze ty operace, které jsou poskytnuty klientovi nad doménovým modelem a samotná servisní vrstva je tak velice tenká.

Druhý princip, tzv. **Operation Script**, funguje přesně opačně. Doménový model neimplementuje žádnou aplikační logiku a tato je celá implementována přímo v servisní vrstvě. Rozhraní dostupné klientovi tedy nedeleguje práci do doménového modelu, ale s doménovým modelem přímo pracují. V tomto případě se dá již hovořit o robustní servisní vrstvě a třídy hrající roli servis bývají programově velmi rozsáhlé.

Oba přístupy mají své klady a zápory. V případě doménové fasády se dá jistě hovořit o lepší dekompozici aplikační logiky a tím pádem rozšiřetelnější implementaci. S tím se ovšem také pojí netriviální návrh aplikace. Navíc díky tomu, že je aplikační logika rozeseta do mnoha míst v doménovém modelu, může se s rostoucím počtem doménových objektů tvořit duplicitní kód. U druhého přístupu je aplikační kód pouze na několika málo místech a to umožňuje snažší refaktorizaci kódu, na druhou stranu to znesnadňuje budoucí rozšiřitelnost doménového modelu.

5.3 Vrstva persistence dat

V předchozím textu jsme zavedli základní terminologii vrstev a třívrstvého architektonického vzoru. Nyní je nutné detailněji rozebrat otázku zachování persistence dat a programového přístupu k těmto datům (tedy nejnižší vrstvy třívrstvého modelu).

5.3.1 Systém řízení báze dat

Řešení problému uchovávání dat spočívá obvykle v nasazení některé z implementací systému řízení báze dat (SŘBD nebo anglicky DBMS – *Database management system*). Jejich hlavním účelem je poskytnutí řízení (vkládání, editace, mazání) nad bází dat (neboli *databází*), definovat strukturu uložení těchto dat a vytvořit rozhraní mezi aplikačními programy a uloženými daty [19]. Pojmem „databázový systém" poté obvykle označuje spojení SŘBD se samotnou bází dat.

Použití některého z databázových systémů přináší celou řadu výhod, jako je udržování strukturovanosti dat, podpora transakcí, možnost agregace dat, řízení přístupových práv atd. Databázové systémy se obvykle snaží maximálně využít operačního a souborového systému tak, aby bylo získávání a ukládání dat co nejvíce efektivní. Převážná většina dnes používaných SŘBD pak pro strukturování dat v databázi využívá tzv. *relační model*.

Relační model

Relační model sdružuje data do tabulek, které obsahují n-tice jednotlivých entit (řádků). Tabulka je struktura entit s pevně stanovenými atributy (sloupci). Každý sloupec má jasně definován jednoznačný název, typ a možný rozsah vkládaných dat. Pokud jsou v různých tabulkách sloupce stejného typu, pak tyto sloupce mohou vytvářet vazby mezi jednotlivými tabulkami. Tabulky se poté naplňují konkrétními daty. Kolekce více tabulek, jejich funkčních vztahů, indexů a dalších součástí tvoří samotnou relační databázi.

Relační model umožňuje přirozenou reprezentaci zpracovávaných dat a je v něm možné při návrhu databáze snadno a poměrně intuitivně definovat jednotlivé vazby mezi tabulkami. Model také klade velký důraz na zachování integrity dat. Model dále pracuje s termíny jako je *referenční integrita*, *cizí klíč*, *primární klíč*, *normální forma* apod. Relační databázový systémy obvykle pro kontrolu databáze a nastavování dalších direktiv využívají strukturovaný dotazovací jazyk SQL (*Structured Query Language*).

Problematika databází je velmi obsáhlá a pro účely této práce postačí, pokud se nadále budeme zabývat jen některými vlastnostmi relačních databázových systémů uplatnitelných v prostředí webových distribuovaných aplikací.

Uložené procedury a triggery

Jak již bylo zmíněno v předchozím textu, databázové systémy umožňují částečně převzít roli aplikační vrstvy díky uloženým procedurám a tzv. *triggerům*. To jsou v podstatě bloky kódu napsaného v některém programovacím jazyce (např. PL/SQL) vytvořeného speciálně pro účel běhu v prostředí relační databáze. Lze je kupříkladu využít pro zachování referenční integrity dat. Výhodou je zpravidla mnohem větší rychlost zpracování než pokud by tyto bloky kódu měly být řízeny vzdáleně klientskou aplikací.

Kromě velmi specifických případů však nelze použití procedur a *triggerů* příliš doporučit. Jsou jen velmi obtížně udržovatelné, na velmi nízké úrovni abstrakce, není možné je rozumně debuggovat a jejich podpora se liší v závislosti na použité implementaci databázového systému.

Nevýhody relačních databází

Ačkoliv existují standardy pro jazyk SQL, mezi jednotlivými implementacemi relačních databází jsou značné rozdíly v jejich podpoře. Obvykle nejsou implementovány všechny požadavky standardu a naopak, každá implementace obsahuje nejrůznější prvky, které nejsou ve standardech obsaženy. Přenositelnost SQL sekvencí mezi jednotlivými databázemi je proto omezená. Současně není možné jednoduše přenášet data mezi různými databázemi.

Relační databázové systémy jsou dobré pro řízení velkého množství dat, ale poskytují nízkou podporu pro manipulaci s nimi. Jsou založeny na dvourozměrných tabulkách a vztahy mezi daty jsou vyjadřovány porovnáváním hodnot v nich uložených. SQL sice umožňuje tabulky za běhu propojit, nicméně tento přístup je velmi neintuitivní a nepřehledný. Relační model databází je sice jednoduchý a přitom velmi robustní a flexibilní, je ale také naprosto rozdílný od objektového modelu. Relační databáze nejsou navrhovány pro ukládání objektů a vytvoření rozhraní pro ukládání těchto objektů v relační databázi je netriviální úkol.

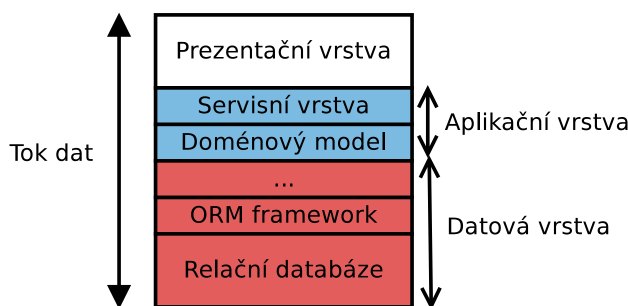
Alespoň částečné řešení obou výše nastíněných problémů tkví v použití buď objektově orientovaných databází (které ovšem nejsou příliš využívané), nebo v nasazení tzv. *objektově-relačního mapování*.

5.3.2 Objektově-relační mapování

Jako objektově-relační mapování (*Object-Relational Mapping*, zkratkou ORM) se označuje sada programovacích technik, které se snaží zajistit automatickou kon-

verzi dat mezi relační databází a doménovým modelem a následnou synchronizaci doménových objektů v aplikaci a jejich reprezentaci v databázovém systému tak, aby byla zajištěna persistence dat (Obr. 5.5). Díky ORM je možné přirozeně pracovat s doménovými objekty, aniž by se programátor staral jak zajistit jejich perzistenci.

Ve [20] jsou rozebrány některé základní návrhové vzory, které umožní svépomocí implementovat ORM. Nicméně protože zajištění synchronizace mezi databází a doménovým modelem je netriviální problém zejména v kontextu paralelního přístupu a výkonnosti celé aplikace, využívá se dnes obvykle již ustálených a standardizovaných frameworků, které ORM mapování zajistí.



Obrázek 5.5: Využití objektově-relačního mapování.

Některé implementace ORM frameworků navíc programátora odstíní od poněkud nepohodlného SQL jazyka a poskytnou mu jiné možnosti přístupu do databáze, které obvyklé implementace databázových systémů nenabízí. Dále také umožňují částečně smazat rozdíly mezi jednotlivými implementacemi databázových systémů – je možné konfigurovat SQL dialekt, se kterým framework komunikuje s databází a tím je lze při změně databázového systému místo přepisování celé datové vrstvy pouze překonfigurovat ORM framework.

I přes nesporné výhody je však nasazení těchto frameworků spíše počátek cesty za bezproblémovým mapováním mezi doménovým modelem a relační databází a v některých případech se dokonce vyplatí jejich nasazení úplně vyhnout.

Java Persistence API

Specifikací pro ORM frameworky na platformě Java je tzv. *Java Persistence API* (zkratkou JPA). Původně existovalo několik různých ORM frameworků, které vznikly jako reakce na příliš komplikovaný tehdejší model EJB, který se navíc dal použít pouze ve světě rozsáhlých Java EE aplikací [21]. V rámci standardizace těchto ORM frameworků poté dodatečné vznikla specifikace JPA, jejíž podporu

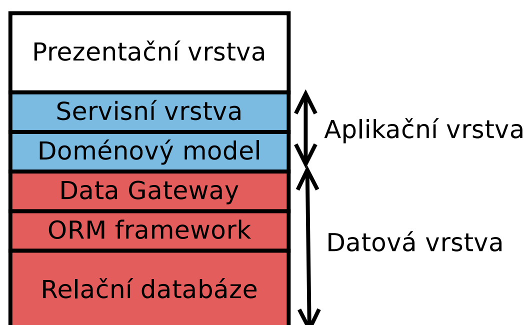
následně stávající frameworky převzaly. Nejznámějšími implementacemi JPA je Hibernate a EclipseLink. Při dodržení konvencí předepsaných JPA je možné tyto dva frameworky na ORM vrstvě navzájem zaměnit bez změny implementace.

JPA mimo jiné definuje i dotazovací jazyk JPQL (*Java Persistence Query Language*). Ten je syntaxí podobný SQL, nicméně namísto tabulek se pohybuje na úrovni doménových objektů. Některé frameworky jazyk dále rozšiřují, například Hibernate má vlastní jazyk HQL (*Hibernate Query Language*), jemuž je JPQL podmnožinou.

5.3.3 Data Gateway

I v okamžiku využití ORM frameworků se nelze vyhnout kódu, který přímo závisí na logice relačních databází. Zejména pro CRUD (*Create-Read-Update-Delete*) operace je stále nutné využít ať už generické SQL nebo například výše zmíněný jazyk JPQL. Je naprosto nežádoucí „míchat“ tento specifický kód do aplikační vrstvy už jen s ohledem na to, že některá z budoucích odvozenin aplikace může využívat místo relačních databází například XML soubory, s nimiž se ovšem pracuje diametrálně odlišně.

Obvyklým postupem je tedy vytvoření programového rozhraní pro přístup k persistentní vrstvě. Implementace tohoto rozhraní bude poté obsahovat všechen kód specifický pro technologii použitou pro persistenci dat. Takové implementace poté tvoří jakousi bránu (*gateway*) mezi architektonickými věty aplikační a persistentní vrstvy (Obr. 5.6).



Obrázek 5.6: Rozhraní mezi aplikační a persistentní vrstvou.

Není žádoucí do těchto objektů vkládat aplikační logiku – rozhraní by mělo být co nejjednodušší a zajišťovat pouze základní CRUD operace. Jakákoliv komplexní logika by měla být součástí klienta tohoho rozhraní. Implementace se poté často neprogramují ručně, ale využívá se spíše různých možností dědičnosti nebo gene-

rátorů kódu. Je dokonce možné vytvářet implementace rozhraní přímo za běhu aplikace pomocí *aspektově orientovaného programování* (viz dále).

Data Access Object

Jako *Data Access Object* (*objekt zpřístupňující data*, zkratkou DAO) se tradičně označuje uplatnění návrhového vzoru *Data Gateway* v prostředí Java aplikací. Rozhraní DAO navenek poskytuje metody pro práci s doménovými objekty – instancemi POJO nebo EJB, a umožňuje jejich persistenci. Vnitřně poté zajišťují „překlad“ těchto objektů do tvaru, kterému rozumí relační databáze. To může být zajištěno mnoha cestami – parametrizací generických SQL dotazů nebo například využitím již zmiňovaných ORM frameworků.

Seznámili jsme se s obvyklými architektonickými návrhovými vzory pro spojení datové a aplikační vrstvy. Nyní je nutné vyřešit poslední prázdné místo v rozboru třívrstvé architektury – prezentační vrstvu.

5.4 Prezentační vrstva

Úkolem této kapitoly je ukázat obvyklá architektonická řešení rozhraní mezi aplikační a prezentační vrstvou specifická pro webové aplikace a představit některé technologie, které tyto obecné vzory implementují.

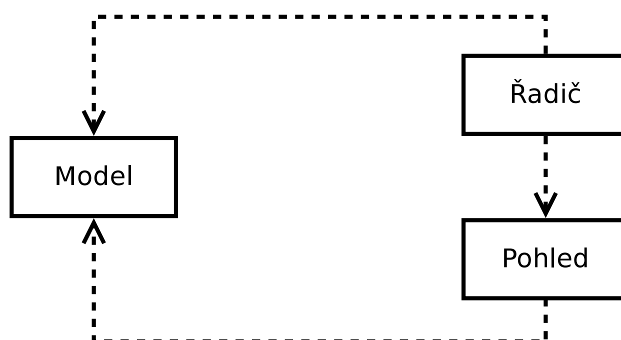
5.4.1 Architektura MVC *(Model-View-Controller)*

MVC je architektonický vzor, který odděluje doménový model aplikace od uživatelského rozhraní. Vznikl spolu s prvními návrhy grafického uživatelského rozhraní a dnes se uplatuje zejména v rámci distribuovaných webových aplikací. Popis jednotlivých komponent je následující:

- *Model* – v klasickém třívrstvém modelu obvykle odpovídá doménovému modelu v aplikační vrstvě. Nicméně frameworky implementující MVC obvykle tyto komponenty neumí navzájem synchronizovat a je tedy na programátorovi, aby synchronizaci zajistil (například přes servisní vrstvu).
- *Pohled (View)* –převádí data reprezentovaná modelem do podoby vhodné k zobrazení uživalům.

- *Řadič (Controler)* – reaguje na události od uživatele a zajišťuje změny v modelu nebo v pohledu.

Ačkoliv by se mohlo zdát, že MVC je vlastně třívrstvá architektura (a oba architektonické vzory opravdu bývají často zaměňovány a jsou si na první pohled podobné), není to vůbec pravda. Topologie MVC totiž není lineární, ale tvoří jakýsi trojúhelník. Na rozdíl od obecné třívrstvé architektury také definuje základní tok dat a pohybuje se na rozhraní aplikační a prezentační vrstvy, způsoby zajištění persistence doménového modelu nejsou architekturou nijak pokryty a obecně nelze zaměňovat datovou vrstvu s MVC *Modelem*. Na Obrázku 5.7 je zobrazeno základní schéma MVC architektury.



Obrázek 5.7: Základní schéma MVC architektury.

Důležitý je zejména fakt, že *Řadič* a *Pohled* závisejí na *Modelu*, ale *Model* nezávisí ani na jedné z těchto dvou komponent. To (mimo jiné) umožňuje testovat doménový model aplikace nezávisle na vizuální prezentaci dat. Dále můžeme podle [20] rozdělit MVC architekturu na dva základní typy.

Pasivní MVC

V pasivním modelu je to pouze *Řadič*, který je oprávněn manipulovat s *Modelem*. *Řadič* modifikuje *Model* a poté informuje *Pohled*, že se *Model* změnil. Tento se poté překreslí, aby zobrazoval aktuální data. *Model* je v pasivním MVC naprosto nezávislý a proto nemusí nijak notifikovat změnu sám sebe – tuto činnost obstará *Řadič*.

Příkladem pasivního MVC může být klasická webová prezentace přes HTTP protokol. Bez využití dalších technologií neexistuje cesta, jak jednoduše přijímat asynchronní aktualizace dat ze serveru. Webový prohlížeč zobrazí *Pohled* a umí zachytávat vstupy od uživatele, neumí však detekovat žádné změny modelu na serveru.

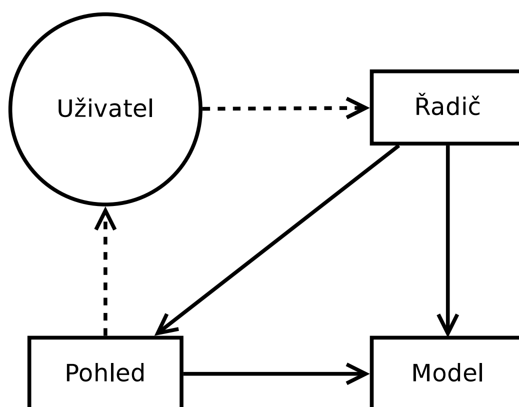
Aktivní MVC

V aktivním MVC může *Model* měnit svůj stav bez jakéhokoliv zapojení *Řadiče*. To se může stát v případě, kdy jsou data měněna z nějakých dalších zdrojů (například jiným uživatelem, přímo v databázi, opakující se paralelní úlohou atp.) a změny se musí okamžitě projevit do *Pohledu*. *Model* tedy notifikuje změnu sama sebe, *Pohled* tuto změnu zachytí a dojde k překreslení okna grafického uživatelského rozhraní. Zapojení *Řadiče* se nicméně nijak nevylučuje. Může to být právě *Řadič*, který *Model* změní, nicméně aktualizaci *Pohledu* nebude sám zajišťovat a nechá ji plně v kompetenci *Modelu*.

Tok událostí v MVC

Pro úplné pochopení MVC architektury jsou na Obrázku 5.8 znázorněny typické interakce mezi uživatelem a aplikací využívající MVC. Tok událostí v aplikaci vypadá následovně:

- uživatel vykoná nějakou akci v uživatelském rozhraní.
- událost je zachycena *Řadičem*.
- *Řadič* provede rozhodnutí o tom, jak na událost zareagovat a změní hodnoty v modelu nebo přímo ovlivní *Pohled*.
- *Pohled* se aktualizuje a uživateli zobrazí změny v *Modelu* (nebo se zobrazí zcela jiný *Pohled*).

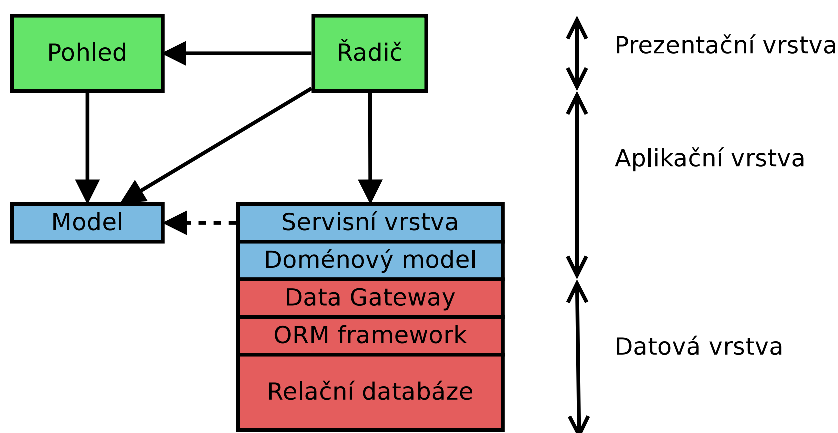


Obrázek 5.8: Scénář interakce v MVC architektuře.

V celém modelu je to tedy *Řadič*, který hraje dominantní úlohu a určuje, jak se bude reagovat na události vzniklé na straně uživatele – odtud anglický název *Controller*.

Spojení MVC a servisní vrstvy

Nabízí se otázka, jak vyřešit již dříve zmíněný problém synchronizace *Modelu* s doménovým modelem aplikace. V Kap. 5.2 jsme definovali servisní vrstvu, která bude vytvářet programové rozhraní mezi prezentační vrstvou a doménovým modelem aplikace. Na obrázku 5.9 je znázorněno jedno z možných spojení MVC se servisní vrstvou (někdy nazývané jako MVCS - *Model View Controller Service*). Jak již bylo dříve zmíněno, servisní vrstva může s doménovým modelem manipulovat přímo podle vzoru *Operation Script*, nebo může tvořit pouze doménovou fasádu a delegovat logiku do doménového modelu.



Obrázek 5.9: MVC architektura ve spojení se servisní vrstvou.

Řadič tedy může přímo měnit *Model* a na žádost uživatele může provést například uložení dat z *Modelu* do databáze prostřednictvím servisní vrstvy. Servisní vrstva může mít přímou referenci na *Model* nebo může být aktualizace provedena přes *Řadič*.

5.4.2 MVC v prostředí webu

Rozlišovat rozdíl mezi *Pohledem* a *Řadičem* není u desktopových aplikací až tak důležité a některé frameworky explicitně logické rozdělení příslušné části prezentační vrstvy do těchto MVC komponent nevyžadují (například stále široce používaný Swing toolkit).

Nicméně zejména u webových aplikací je situace zcela jiná a MVC architektura je zde naprosto přirozeně využita pro dekompozici komponent starajících se o generování HTML kódu od částí systému obsluhujících HTTP požadavky:

- *Model* je identický s *Modelem* v desktopových technologiích (obsahuje data a doménovou logiku).
- *Pohled* je ta část serverového kódu, která se stará o generování HTML kódu. Může však prezentovat data i jinak, například ve formátu XML, JSON (*JavaScript Object Notation*), PDF (*Portable Document Format*) atp.
- *Řadič* se v prostředí webu nejčastěji skládá ze dvou hlavních částí. První je tzv. *Front Controller*, který zachytává všechny HTTP požadavky. Ty následně zpracuje a přepošle dalším *Řadičům*, které jsou obvykle identifikovány podle URI. Konkrétní *Řadič* potom typicky přijme data původně pocházející z HTTP požadavku, uloží je do *Modelu* a ten prováže s konkrétním *Pohledem*.

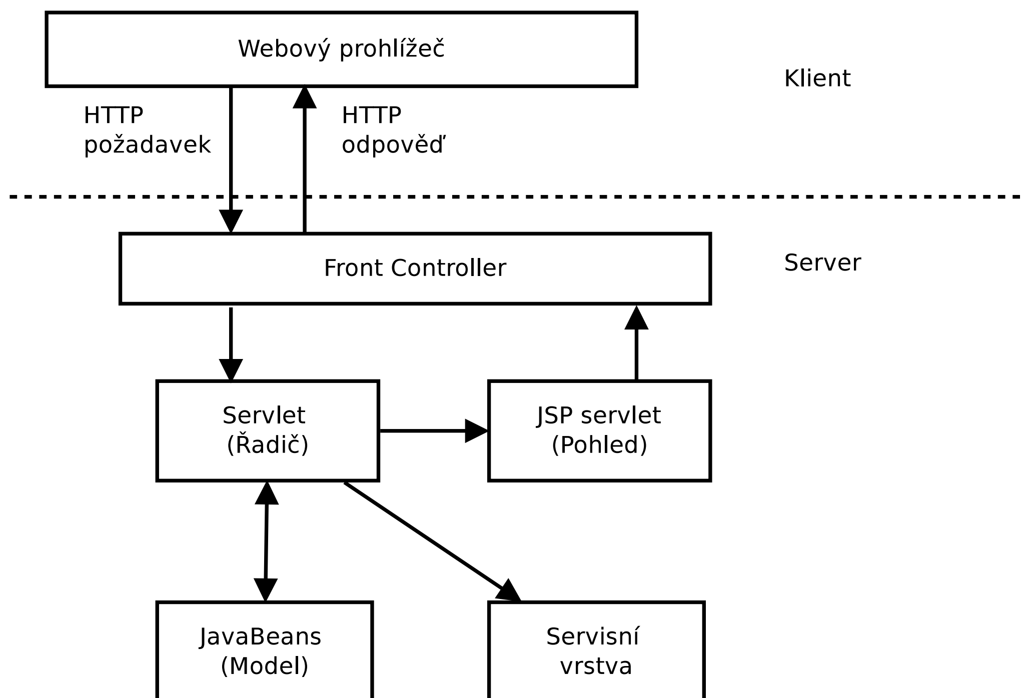
Situace se komplikuje v případě, kdy je využito technologií jako je AJAX (*Asynchronous JavaScript and XML*). V těchto situacích je značná část prezentační logiky (tedy *Pohledu*) přesunuta na klienta (jak velká část záleží na konkrétní implementaci). Nicméně lze říci, že i v těchto případech je architektura MVC využitelná.

JavaServer Pages

Jako nejtypičtějšího představitele uplatnění MVC ve světě webových Java aplikací můžeme jmenovat technologii tzv. *servletů* a s tím související *JavaServer Pages* (JSP). Servlet je velmi obecně řečeno třída odpovědná za zpracování HTTP požadavku a vytvoření odpovídající odpovědi [22]. Servletu bývá v průměrné webové aplikaci celá řada a jejich organizaci a mapování na jednotlivá URL má na starosti tzv. *servletové* nebo také *webové kontejnery*, což jsou vlastně obecně známé „Java servery" jako je například Apache Tomcat nebo GlassFish. Tyto kontejnery obvykle vytvoří jednu nebo více instancí od každého servletu a pokud na server přijde HTTP požadavek, je vyvolána odpovídající metoda instance servletu namapovaného na URL tohoto požadavku.

JSP servlet je poté velmi specifický typ servletu. Jeho programový zápis totiž není realizován v Javě, ale ve struktuře velmi blízké HTML. Je možné do značek HTML „míchat" další tagy ze standardních JSP knihoven, vytvořit si knihovnu vlastní nebo dále používat dalších šablonovacích nástrojů. Lze také omezeně využít

Java kód. Takto zapsaný servlet je poté interně v servletovém kontejneru automaticky konvertován na Java zdrojový kód a následně se chová jako každá jiná třída. Nicméně jak ukazuje Obr. 5.10, cílem JSP servletů je jediná věc – formátovat data obdržená ve od *Řadiče* do podoby HTML kódu (schéma není zcela přesné, obvykle nejsou data do JSP servletu předána přímo přes *Řadič*, ale přes nadřazený *Front Controller*).



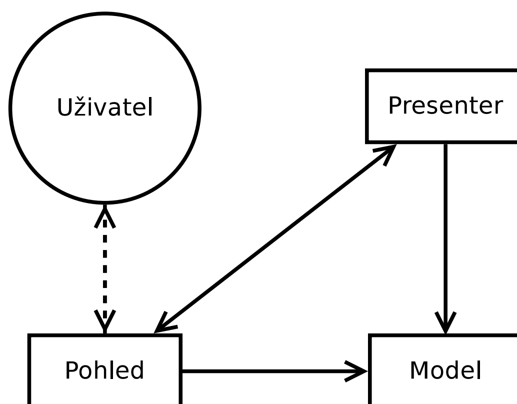
Obrázek 5.10: Základní architektura JSP.

Řadič poté obvykle bývá realizovaný jako klasický programově implementovaný servlet. V servletu *Řadiče* může být soustředěna veškerá logika související s přípravou dat k zobrazení nebo volání nižších vrstev pro načtení dat do *Modelu*. *Řadič* také může provádět přesměrování a další věci, které nutně nesouvisí s aplikační logikou – ta by měla ležet vždy v aplikační vrstvě.

Původní čisté JSP se dnes již u důvodu velké pracnosti vývoje moc nepoužívá a v praxi tuto technologii nahradily frameworky, které možnosti JSP dále rozšiřují. Je však nutné pochopit principy fungování JSP a MVC architektury obecně, protože z nich většina moderních frameworků přímo vychází.

5.4.3 Architektura MVP (*Model-View-Presenter*)

Architektonický vzor MVP je principiálně podobný MVC a oba vzory bývají dost často navzájem zaměňovány. Nicméně jednotlivé komponenty hrají v MVP trochu jinou roli (Obr. 5.11).



Obrázek 5.11: Scénář interakce v MVP achitektuře.

Uživatelský vstup a výstup tentokrát plně kontroluje *Pohled* skrze ovládací prvky uživatelského rozhraní. Primární motivací pro oddělení *Pohledu* a *Presenteru* (budeme nadále používat anglické slovo „presenter“, protože pro ně není žádný vhodný český ekvivalent) už není nutnost ošetření událostí od uživatele a kontroly vstupu, důvody jsou čistě architektonické (lepší udržovatelnost kódu). *Pohled* má typicky přímou vazbu na *Presenter* a ten obvykle přímo pracuje s *Pohledem*, takže i tato vazba je silnější než v případě MVC. *Pohled* oproti MVC navíc zpracovává uživatelský vstup a je zde tedy dominantní komponentou (typicky v reakci na událost od uživatele zavolá nějakou metodu v *Presenteru*).

Presenter může obsahovat prezentační a aplikační logiku (nebo deleguje aplikační logiku do dříve popsané servisní vrstvy). Manipuluje s *Modelem*, což (například pomocí systému posluchačů a notifikací) zajistí aktualizaci *Pohledu*, nebo ovlivňuje *Pohled* přímo (záleží na implementaci a možnostech jazyka).

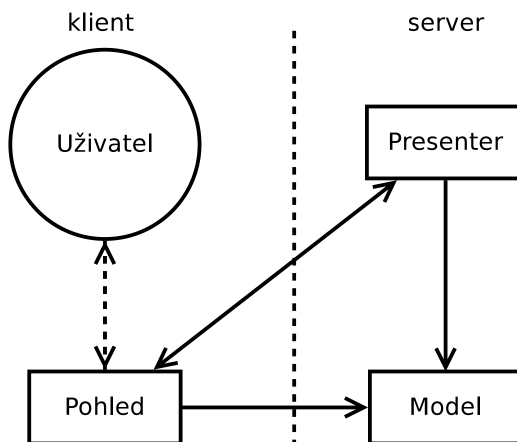
Popsaný architektonický vzor se také označuje jako *Supervising Presenter*. Existují další modifikace MVP, například *Passive View*, kdy je naopak zvýšena odpovědnost *Presenteru* a *Pohled* úplně ztrácí vazbu na *Model*. Jejich popis však přesahuje účel tohoto textu. Důvod, proč zde uvádíme tento architektonický vzor je zejména fakt, že se MVP velice často a zcela nesprávně zaměňuje s MVC. Příkladem využití MVP může být toolkit WinForms, který je součástí *.NET Frameworku* od společnosti Microsoft.

MVP v prostředí webu

MVC se dobře hodí pro klasické webové aplikace, kde jsou statické HTML stránky generovány přímo na serveru. Nicméně v případě moderních AJAX aplikací, kdy se část prezentační vrstvy přesouvá do klientského prohlížeče, začíná být MVC nevyhovující. Oproti tomu MVP architektura je pro tento účel jako stvořená.

Jak je vidět na Obrázku 5.12, komponenta *Pohledu* (obvykle implementovaná pomocí JavaScriptu) se přesouvá do klientského prohlížeče a místo programového rozhraní a volání metod budou tyto komunikovat například pomocí HTTP protokolu. Při vyplňování formulářů je možné validovat data přímo na straně klienta. Pokud tedy uživatel klikne na tlačítko, které má zobrazit nějaké položky z databáze, je tento požadavek zpracován na straně webového prohlížeče a na pozadí se odešle žádost do *Presenteru*. Ten může ze servisní vrstvy načíst patřičná data za databáze a uložit do *Modelu*. Poté dojde opět pomocí HTTP protokolu k aktualizování *Pohledu* tak, aby zobrazil aktuální data.

Z toho vyplývá také potenciál snížit zátěž na servery a síť obecně. Jelikož není potřeba při každém požadavku sestavit celý HTML dokument, ale pouze zobrazit aktualizovaný *Model*, je množství vyměňovaných dat výrazně nižší. Nicméně tento přístup naopak může zvýšit počet vyměňovaných HTTP požadavků, a třebaže přenášejí nižší množství dat, při nevhodné implementaci zátěž neklesne.



Obrázek 5.12: Scénář interakce ve webové MVP aplikaci.

5.4.4 JavaServer Faces

Již dříve jsme si ukázali JSP jako jedno z ukázkových nasazení architektury MVC ve webové aplikaci. Nyní si detailněji rozebereme technologii *JavaServer Faces*

(zkratkou JSF), která i přes to, že z JSP přímo vychází, se naopak blíží spíše MVP architektuře.

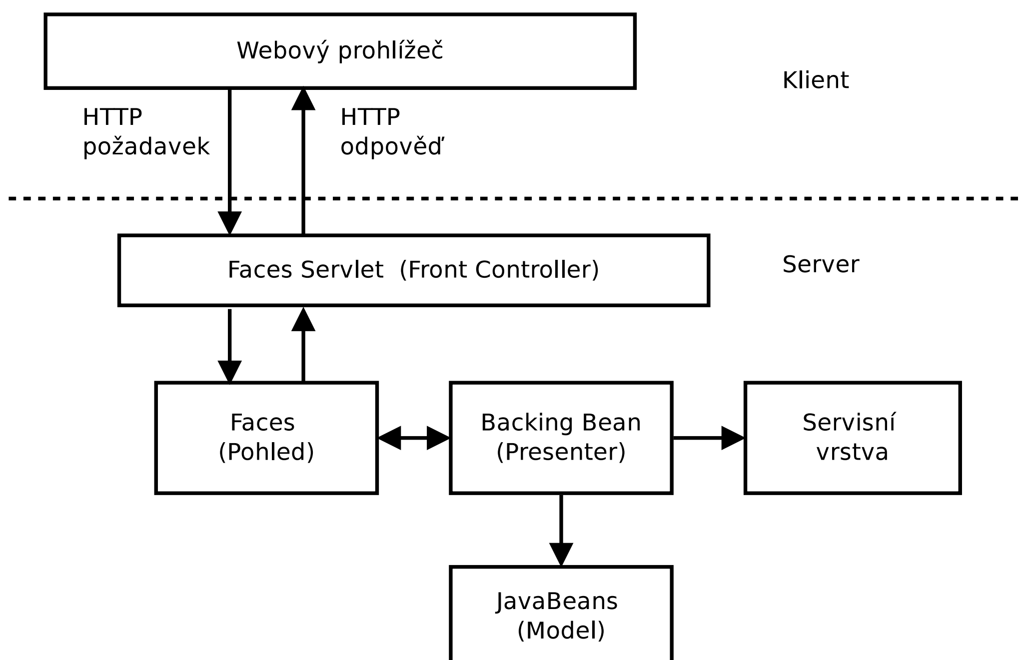
Myšlenkou je opět oddělení definice uživatelského rozhraní (tzv. *Faces*) od vlastní aplikační a prezentační logiky. I zde se tak děje pomocí souborů podobných HTML, ve kterých je definováno vlastní uživatelské rozhraní. Při psaní těchto XML souborů se používají speciální XML tagy, které se importují z tzv. *Tag Library Description* (TLD) knihoven. I zde je využito *Front Controlleru* a vykreslování HTML stránky se odehrává na serveru [23].

První rozdíl přichází v okamžiku, kdy si uvědomíme, že konfigurace grafického rozhraní je komponentově orientovaná. Každý tag má interně v rámci své knihovny přiřazeno několik Java tříd. Soubor těchto tříd se nazývá *Komponenta*. Tato programová reprezentace vytváří aplikační logiku tagu (jako konverzi hodnot na text, validaci a přípravu pro „vykreslení" informace). Ke každé komponentě je vytvořen její programový *Renderer*, který zapíše do výstupu vlastní HTML kód a data z reprezentace tagu. Je tedy možné vytvořit celé uživatelské rozhraní aniž bychom použili jediný HMTL tag. V JSF také odpadá nutnost využití šablonovacích frameworků (součástí je framework *Facelets*, který umožňuje šablonování).

Jak je vidět na Obrázku 5.13, JSF zavádí také pojem *backing* nebo také *managed beans*. Přestože jsou tyto „beany" často realizovány jako POJO objekty, nemají vůbec nic společného s doménovým modelem aplikace. Reprezentují ty třídy, jejichž instance by měly být dynamicky vytvářeny za běhu aplikace spolu s generováním HTML stránek, přičemž je možné určit jejich „rozsah" (*scope*) neboli kontext ve kterém budou konkrétní instance referencovatelné (je možné určit rozsah v rámci jednoho HTTP požadavku, uživatelské relace, v rámci celého běhu aplikace atp.). Jednotlivé členské proměnné a metody těchto objektů jsou poté za pomoci tzv. *bindování* možné referencovat přímo z XML souborů jednotlivých *Pohledů* a na události vyvolané uživatelem (například odeslání vyplněného formuláře) volat přímo metody *backing beany*.

Jak je tedy patrné, architektura JSF je v přímé shodě s architektonickým vzorem MVP – dominantní komponentou zde není *Řadič* ani *Presenter*, ale je to právě *Pohled*, který je při konstrukci HTML stránky vytvořen jako první a jednotlivé *backing beany* hrají roli zpřístupnění *Modelu* a dodatečné prezentační logiky. *Backing beany* také mohou přímo ovlivňovat *Pohled*, protože jednotlivé komponenty jsou referencovatelné jako klasické Java objekty přímo z aplikačního kódu, což je další obrovská výhoda oproti JSP (i když trochu navádí k vytváření různých formátovacích „zkratek" v *Presenteru*, což je principiálně špatně).

JSF nicméně zachází ještě dál. Některé TLD knihovny (např. MyFaces nebo Primefaces) obsahují komponenty, které mohou s *backing beanou* přímo komunikovat pomocí AJAXu. Renderery těchto komponent při vytváření (z principu



Obrázek 5.13: Základní architektura JSF.

statického) HTML kódu jednoduše přidají do stránky JavaScript, který umožní obousměrnou komunikaci mezi serverem a webových prohlížečem klienta. *Pohled* se pak částečně přesouvá do webového prohlížeče a programátorům je umožněno vytvářet velice rychle robustní interaktivní webové grafické rozhraní, aniž by napsali jedinou řádku HTML nebo JavaScriptového kódu.

5.4.5 Shrnutí

V předchozích kapitolách jsme se seznámili s architektonickými postupy uplatnitelnými při návrhu rozsáhlých webových aplikací a distribuovaných informačních systémů a nejpoužívanějšími frameworky, které tyto postupy implementují pro platformu postavenou na programovacím jazyce Java. Definovali jsme základní architektonické pojmy jako je *třívrstvá architektura*, *doménový model* a bude žádoucí se těchto termínů a postupů držet i nadále.

Nicméně dosud jsme se zabývali architekturou, která sledovala jediný cíl – poskytnutí webového rozhraní čitelného a použitelného pro lidského uživatele. Nyní je třeba si položit otázku, jak mohou s webovými aplikacemi komunikovat jiné systémy nebo klienti, kteří neumožňují zobrazovat webové stránky nebo je z nějakého důvodu nežádoucí či nepraktické těmto klientům webové rozhraní poskytovat.

5.5 Architektura orientovaná na služby

Architektura orientovaná na služby (*Service-oriented architecture*, zkratkou SOA) je obecný architektonický vzor založený na spolupráci navzájem nezávislých služeb [24]. Motivací pro vytvoření SOA byla rostoucí náročnost na udržování spolupráce různých vnitropodnikových systémů. Díky jejich platformní závislosti a navzájem nekompatibilních programových rozhraní bylo jen velmi obtížně zajistit vzájemnou komunikaci těchto systémů.

5.5.1 Služba v SOA

Služba je určitá část funkčnosti aplikace, která je zpřístupněná pomocí definovaného rozhraní. Každý jednotlivý informační systém může poskytovat sadu služeb svému okolí. Za službou v SOA stojí většinou poměrně velké množství programového kódu a jakkoliv je princip volání služeb podobný volání metod, probíhá zde na vyšší úrovni granularity.

Rozhraní služeb také není závislé na implementaci hostitelského systému. Hranice systému se od programových rozhraní posunují dále od klasického programového API k jasně definovanému rozhraní, ve kterém daná služba data produkuje či přijímá bez ohledu na to, v jaké technologii je tato služba implementována.

Ve většině případů se pak pro komunikaci mezi službami díky vysoké prostupnosti sítí, platformní nezávislosti a přenositelnosti využívá HTTP protokol. Přenášený formát dat může být v podstatě libovolný, pokud všichni aktéři komunikace tento formát podporují. Nicméně zdaleka nejvyužívanější jsou formáty postavené (zejména z důvodu transparentnosti a multiplatformnosti) na bázi XML. Takové služby se poté zpravidla označují jako služby webové (*Web Services*). Takové služby se pak obvykle jednoznačně identifikují na základě URL adresy.

5.5.2 SOA a třívrstvá architektura

Vícevrstvá architektura se SOA spolu navzájem velice úzce souvisí. Jednotlivé systémy jsou obvykle distribuované, vícevrstvé aplikace s prezentační, aplikační a perzistentní vrstvou. Nicméně veškerá funkcionalita takové aplikace je vystavena prostřednictvím služeb, nebo je vytvořeno rozhraní, které tyto služby navenek poskytuje.

V SOA aplikacích se typicky neudržuje stav konverzace s klientem, komunikace obvykle probíhá způsobem dotaz-odpověď. Tím se nápadně blíží jednodu-

chému HTTP požadavku s tím rozdílem, že URL požadavku tentokrát neidentifikuje HTML stránku, ale přímo konkrétní webovou službu.

Je tedy zcela jistě možné (v případě předchozí dobré dekompozice problému) rozhraní webových služeb implementovat za využití architektonického vzoru MVC na prezentační vrstvě, kdy jediný rozdíl nastane v *Pohledu*, který místo zobrazitelného HTML bude generovat XML dokument o předem daném schématu.

5.5.3 Výhody SOA

Díky bezstavosti a slabému provázání komponent je možné služby skládat do větších celků, což ideálně splňuje potřebu podniků pokrýt a integrovat své vnitropodnikové procesy pomocí informačního systému. Jednotlivé komponenty je možné při výpadku velice rychle nahradit nebo zavést centrální registr služeb, který umožní vyhledávání aktivních služeb a získání jejich popisu (mechanismus *Universal Description, Discovery and Integration*, zkratkou UDDI).

5.5.4 Protokoly komunikace v SOA

Již dříve jsme si řekli, že data jsou v SOA architektuře obvykle přenášeny pomocí XML dokumentů. Tyto lze ale dále strukturovat podle předem definovaných schémat a pravidel a vytvářet komunikační protokoly, který jsou na XML formátu založené.

Simple Object Access Protocol (SOAP)

SOAP je standardizovaný protokol pro výměnu dat mezi webovými službami [25], který je obvykle použit spolu s HTTP protokolem. Zprávu zde reprezentuje jednoduchý XML dokument, který má kořenový element *Envelope*. V této obálce jsou pak uzavřeny dva elementy *Header* (hlavička) a *Body* (tělo).

Hlavička se zpravidla používá pro přenos pomocných informací pro zpracování zprávy – například autorizaci klienta. V těle zprávy se přenáší identifikátor volané služby a parametry zprávy, resp. návratové hodnoty služby. SOAP používá jmenné prostory pro identifikování jednotlivých částí XML zprávy. Je třeba zdůraznit, že SOAP protokol je orientován procedurálně a funguje na principu RPC (*Remote Procedure Call*). Komunikace může být v SOAP stavová.

Typický SOAP požadavek se zasílá v těle HTTP požadavku. Používá se přitom metoda POST, která podle [26] dovoluje posílat data v těle HTTP požadavku. Nicméně ten může mít i další metody, které je možné využít. Z toho by mělo být patrné, že klasická SOAP komunikace vytváří vlastní protokol nad klasickým HTTP a nesnaží se jej nijak dále využít. HTTP protokol je přitom dostatečně robustní na to, aby bylo možné jej rozšířit až do úrovně protokolu určeného pro webové služby.

Representational State Transfer (REST)

REST je architektura rozhraní navržená pro distribuované prostředí. REST je narozdíl od SOAP orientován datově, nikoli procedurálně: rozhraní webových REST služeb je použitelné pro jednotný a snadný přístup k tzv. *zdrojům* (datům nebo stavům aplikace, pokud je lze popsat konkrétními daty). Všechny zdroje jsou jednoznačně identifikovatelné pomocí URI a jsou reprezentovány (a přenášeny) pomocí různých datových formátů (XML, JSON nebo jiného formátu, který je možné odeslat pomocí HTTP).

REST definuje základní CRUD(*Create-Read-Update-Delete*) metody pro manipulaci s těmito zdroji (metody *mění stav* zdroje). Tyto metody jsou poté realizovány pomocí odpovídajících HTTP metod (*GET*, *POST*, *PUT*, *DELETE*). Je tedy patrné, že zatímco SOAP používal HTTP pouze jako jeden z možných komunikačních kanálů, REST na architektuře HTTP přímo staví a dále ji rozšiřuje.

Narozdíl od SOAP neexistují žádné specifikace kladené na strukturu dat. REST je také koncipován jako bezestavový. Jednotlivé implementace REST rozhraní (tedy webové služby) se poté při splnění těchto podmínek obvykle označují jako *RESTful*.

SOAP definuje vlastní způsoby autorizace, REST využívá HTTP autorizace. Autorizační tokeny ale musí být odeslány s každým požadavkem, jinak není splněna podmínka bezestavosti (webová služba si o klientovi nesmí nic „pamatovat“).

Srovnání REST a SOAP

Učinit objektivní srovnání těchto dvou protokolů není zcela možné už jen kvůli tomu, že se oba od sebe odlišují samotnou svou koncepcí. Ač je to dnes populární názor, REST není vždy tou správnou volbou. Sice je implementačně mnohem méně náročný a pouze rozšiřuje standardní HTTP protokol, díky jeho „datacentrismu“ a omezené množině CRUD operací může být implementačně příliš svazující. Také bezestavost komunikace se může ukázat jako velký problém. Jednoduchost REST

rozhraní může být výhoda a slabina zároveň, vždy záleží na případu implementace. Obecně lze říci že REST je velmi silný ve spojení s technologiemi jako je AJAX nebo ke vzdálené správě databáze (k čemuž vybízí už jen základní CRUD matice operací).

SOAP je robustnější a díky stavové komunikaci a procedurální orientaci je mnohem vhodnější k realizaci komunikace mezi dvěma netriviálními informačními systémy, což je ovšem vykoupeno potřebou mnohem rozsáhlejší implementace. Navíc funguje stejně dobře i na jiných sít'ových vrstvách a není třeba se omezovat na pouhý HTTP protokol.

5.6 Technologické nástroje pro vývoj

Všechny dříve zmíněné architektonické vzory je možné implementovat „manuálně" a svépomocí spravovat a udržovat jednotlivé vazby mezi doménovými objekty, mezi vrstvami, komponentami či užitými frameworky. Nicméně takto postavená aplikace je jen obtížně znovupoužitelná a není nutné vlastním kódem řešit problémy, které již někdo vyřešil před námi. V této kapitole si ukážeme postupy, které vedou ke zjednodušení vlastního aplikačního kódu a představíme si framework, který tyto postupy implementuje.

5.6.1 Aspektově orientované programování

Principem *aspektově orientovaného programování* (zkratkou AOP) je oddělení některých částí kódu, tzv. *průřezových koncernů* (*cross-cutting concerns*), od vlastní aplikační logiky. Jako průřezový koncern můžeme definovat takový kód, který se dotýká mnoha různých částí aplikace (příkladem mohou být například transakční a bezpečnostní algoritmy nebo logování) a není jednoduché ho dekomponovat. AOP se snaží tyto koncerny organizovat do tzv. *aspektů*.

AOP není náhrada jiných programovacích technik jako je např. *objektově orientované programování*, slouží spíše jako architektonický doplněk k již zavedeným technikám a uplatní se zejména tam, kde tyto techniky ve snaze dobře dekomponovat kód již nestačí nebo je jejich uplatnění nešikovné.

Aspekty upravují chování programu při spouštění metod, přístupu k atributům třídy nebo při vyhození výjimky. Tato místa se označují jako přípojné body (*joinpoints*). Samotná úprava chování aplikačního kódu (realizovaná opět jako prostý blok kódu) se poté označuje jako *advice*. V Javě se pak označení přípojných bodů v aplikačním kódu obvykle řeší pomocí *anotací*.

Java Anotace

Anotace jsou vlastně jenom textové značky v kódu o určitém předepsaném formátu (anotací je i notoricky známé `@Override`), jejichž cílem je nějakému programovému primitivu přiřadit příznak nebo metadata. Anotace je možné vyhodnocovat při překladu nebo při samotném běhu programu (to ovšem vyžaduje speciální *classloader*). Ve spojení s AOP mohou identifikovat ta programová primitiva, kterým se má přiřadit aspekt (ten je identifikovaný samotnou anotací). I bez využití AOP je možné vytvářet anotace vlastní pro účely např. dokumentace, statického generování kódu atp.

Anotacím je možné vytvářet parametry a těm při použití anotace předávat hodnoty, což jejich možnosti dále rozšiřuje (je možné tyto hodnoty předávat aspektům, které jsou anotacemi identifikovány).

5.6.2 Spring Framework

Cílem této kapitoly je představit framework, který umožňuje spojit většinu uvedených technologií a postupů do jednoho kompaktního celku – Spring Framework. Tento velice populární modulární aplikační framework umožňuje velice rychle vytvářet rozsáhlé aplikace za využití principu *Inversion of Control*, kdy je framework za běhu aplikace zodpovědný za vytvoření a provázání objektů implementované aplikace a na programátorovi závisí pouze samotná implementace aplikační logiky [27]. To je možné díky pokročilým programovacím technikám, jako je například *reflexe*. Nicméně hlavní část Spring Frameworku stojí právě na anotacích a AOP.

Jednotlivé komponenty aplikační logiky, jako jsou např. POJO objekty, se doslova „zaregistrují“ do aplikačního frameworku a tento s těmito komponentami dále manipuluje. Je možné si představit například implementaci návrhového vzoru MVC, kdy programátor implementuje všechny tři komponenty *Pohledu*, *Modelu* a *Řadiče*, aniž by tyto na sebe měly navzájem jakoukoliv programovou vazbu nebo dědily nějakou generickou implementaci. Aplikační framework se pak za běhu aplikace postará o jejich vzájemné provázání. Za využití reflexe a AOP Spring Framework dále *injektuje* vlastní aplikační kód (neboli *aspekty*) do zdrojového kódu vytvářené aplikace, a to buď na základě XML konfigurace, nebo pomocí již zmíněných anotací.

Mezi části Spring Frameworku patří komponenty pro ORM mapování, testovací nástroje, integraci uživatelských komponent, modulárně orientovaný vývoj, webové aplikace a webové služby, bezpečnost na základě rolí a řada dalších. Podrobný popis těchto komponent jde dalece nad rámec této práce a v případě nutnosti budou jednotlivé komponenty v dalším textu stručně popsány.

5.7 Shrnutí

Nyní již máme obecný přehled o tom, jaké architektonické vzory můžeme využít pro programování rozsáhlých distribuovaných systémů a známe široce využívané technologie stavějící na Java platformě, které tyto postupy implementují. V další kapitole si ukážeme konkrétní systém, který byl implementován s cílem pokrýt požadavky na systém správy rezervací, které byly uvedeny v Kapitole 4.

6 RAT - Reserve A Thing!

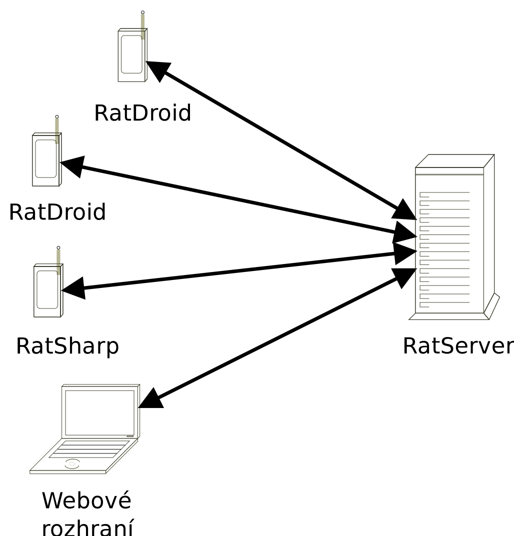
V této kapitole se seznámíme s informačním systémem RAT a detailněji rozebereme implementační detaily všech jeho částí zejména s ohledem na obecné postupy a technologie, které byly uvedeny v Kap. 5. Na závěr se seznámíme s uskutečněným testováním a nasazením tohoto systému.

6.1 Informační systém RAT

RAT je název informačního systému, který byl implementován za cílem pokrýt požadavky na systém správy rezervací definované v Kap. 4. Sestává se z centrálního serveru *RatServer* a mobilní aplikace *RatDroid*, která byla implementována pro mobilní platformu Android. Nad rámec specifikace byla vytvořena i aplikace pro platformu Windows Phone s názvem *RatSharp*.

6.1.1 Základní rozvržení systému

Jednotlivé mobilní aplikace hrají roli klientů centrálního serveru. Server ve shodě se specifikací poskytuje také webové rozhraní. Jednotlivé klientské mobilní aplikace využívají webových služeb, které centrální server poskytuje (Obr. 6.1).

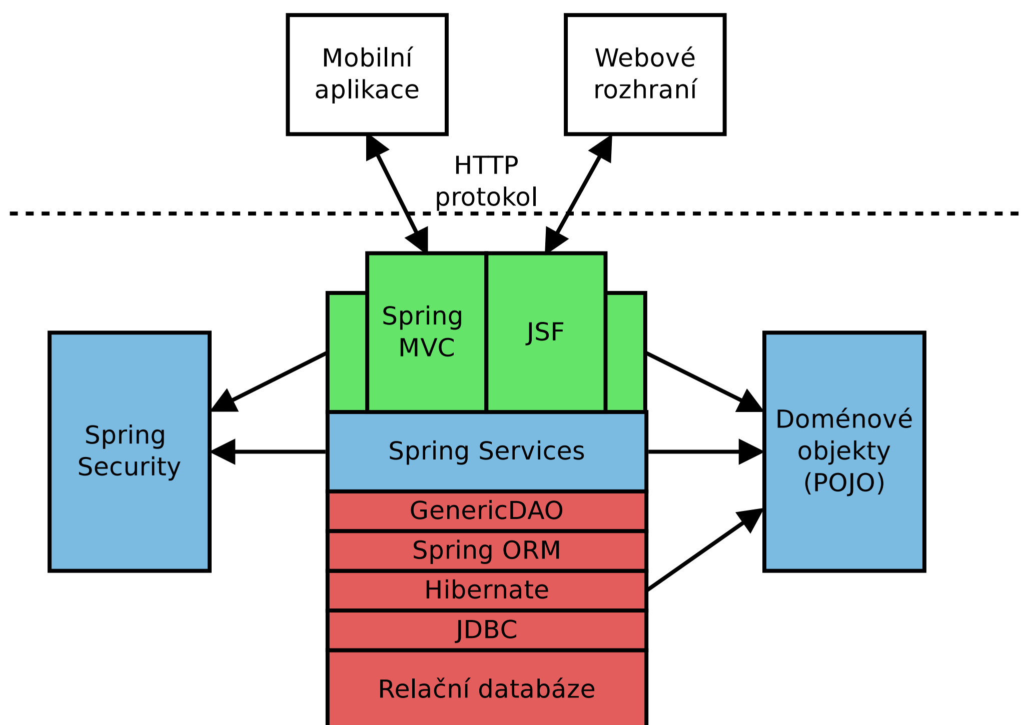


Obrázek 6.1: Informační systém RAT.

6.2 RAT server

6.2.1 Základní architektura serveru

Na Obrázku 6.2 je znázorněno zjednodušené schéma serverové aplikace, včetně rozpisu největších užitých frameworků. V dalším textu se budeme podrobně věnovat jednotlivým částem systému, přiřazovat jim jednotlivé Java balíky z reálné implementace a zaobírat se kontextem komponent vůči zbytku systému (rozkreslit celou serverovou aplikaci do jednoho schématu není vzhledem k její obsáhlosti dost dobře možné).



Obrázek 6.2: Architektura a použité technologie.

Architektura serverové části je datacentrická a servisově orientovaná podle vzoru *Operation Script* (viz Kap. 5). Doménový model tvoří pouze POJO objekty bez aplikační logiky, která je soustředěna v servisní vrstvě. Výhodou tohoto přístupu je zejména jednoduchá implementace SOA architektury, která s vnitřní servisní vrstvou nepřímo souvisí.

6.2.2 Doménový model

Základ celé architektury centrálního serveru je sada doménových POJO objektů, umístěných v balíku `beans`. Objekty jsou včetně vzájemných vazeb znázorněny na Diagramu 6.1.

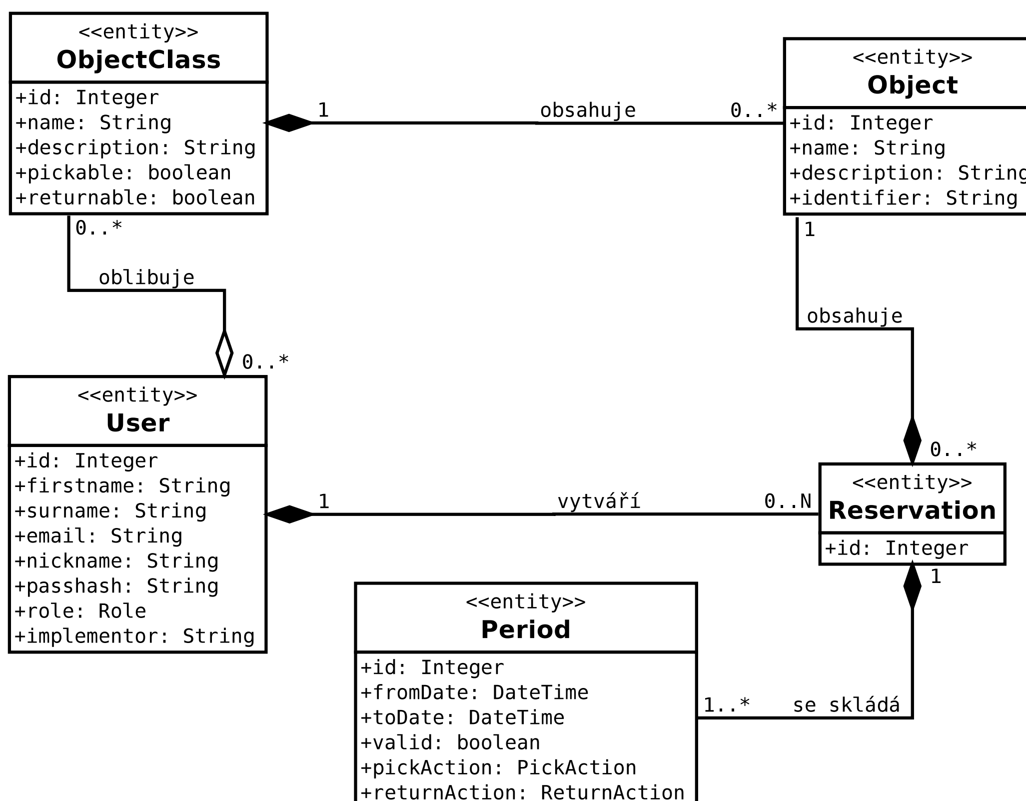


Diagram 6.1: Doménový diagram systému.

Následuje popis jednotlivých doménových objektů a jejich vazeb.

ObjectClass

POJO objekt reprezentující třídu objektů. Kromě specifikací již předem určených atributů (`name` a `description`) přidává také binární příznaky `pickable` a `returnable`, které definují nutnost potvrzení vyzvednutí a vrácení objektů, které patří do této třídy.

Object

Reprezentuje předmět rezervace a výpůjčky.

User

Objekt reflektující uživatele systému. K atributům, které přímo vyplývají z požadavků na systém, jsou ještě navíc definovány atributy `Role` a `Implementor`. Budeme se jimi zabývat v dalším textu.

Reservation

Reprezentace rezervace objektu. Jejím cílem je svázat uživatele, objekt a časové periody náležející této rezervaci do jednoho logického celku.

Period

Časová perioda rezervace. Definuje časové rozpětí rezervace (atributy `fromDate` a `toDate`) a platnost rezervace (atribut `valid`). Za zmínku však stojí zejména atributy udávací „životní cyklus“ periody rezervace – `pickAction` a `returnAction`.

Datový typ `pickAction` atributu je výčtový `enum PickAction` se členy `PICKED`, `NOT_PICKED` a `NOT_PICKABLE`. Tento atribut je přímo svázaný s objektem rezervace – pokud je objekt nastavený jako nepotvrzovaný při vyzvednutí, jsou všechny periody takové rezervace automaticky nastaveny jako `NOT_PICKABLE`, v opačném případě pak na `NOT_PICKED`. Systém tak může od sebe odlišit jednotlivé požadované akce pro rezervace a podle toho řídit další chování. Při vyzvednutí se atribut překlopí do stavu `PICKED`. Obdobná situace platí i pro atribut `returnAction`.

6.2.3 Datová vrstva

V projektu je plně využito možností relačních databází a ORM mapování. Jako mapovací framework byl použit Hibernate. Na Digramu 6.2 je znázorněna základní struktura relační databáze, včetně vzájemných vazeb jednotlivých databázových entit (primární klíče tabulek jsou podtrženy a značeny tučným písmem, povinně vyplnitelné položky začínají černě vybarveným kolečkem).

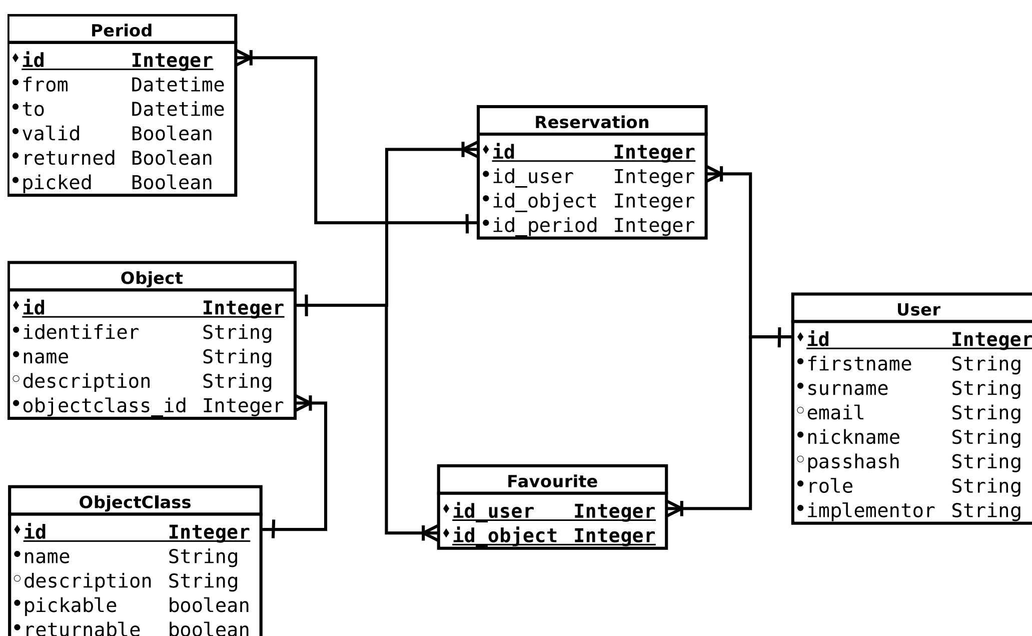


Diagram 6.2: ER diagram databáze.

Doménové POJO objekty jsou mapovány pomocí Hibernate na korespondující databázové entity. Každý doménový objekt má k sobě vytvořen odpovídající mapovací `hbm.xml` soubor, ve kterém jsou mapovací direktivy včetně vzájemných entitních vazeb.

Spring ORM

Spring ORM je jakási mezivrstva mezi frameworkem Hibernate a zbytkem aplikace. Umožňuje zejména konfiguraci Hibernate napojit přímo na konfigurační soubory Spring Frameworku, dále do něj zavádí některé své vlastní implementace jako je `sessionFactory` a transakční manažer (díky tomu bude například možné řídit transakce pomocí anotací přímo z aplikačního kódu). Pomáhá tedy bezproblémově integrovat Hibernate se zbytkem aplikace. Veškerá konfigurace persistentní vrstvy je v XML souboru `persistence.xml` a až na výjimky (viz dále) nijak nevybočuje ze standardní konfigurace.

GenericDAO

Již dříve jsme mluvili o návrhovém vzoru *Data Gateway* a DAO objektech. V RAT systému je pro každý doménový objekt vytvořen odpovídající DAO objekt, který

odděluje aplikační vrstvu aplikace od kódu závislého na databázovém systému (respektive na Hibernate frameworku). Nicméně protože DAO objekty obvykle zajišťují pouze CRUD operace a spouštění některých specifických dotazů, je kvůli snaze o minimalizaci kódu vytvořena hierarchická struktura využívající genericity, dědičnosti a aspektového programování.

Na Diagramu 6.3 je vidět základní struktura DAO vrstvy. Rozhraní každého DAO objektu musí dědit od `GenericDAO` rozhraní. To definuje základní CRUD operace, které lze provádět s objekty. Rozhraní specifické pro doménový objekt poté může tyto základní operace rozšířit o další metody (v našem případě je to rozhraní `ObjectClassDAO`, které umožňuje navíc vyhledat třídu objektu podle jejího jména). Takové rozhraní budeme dále nazývat jako *rozšiřující rozhraní*. Konkrétní implementace generického DAO rozhraní (`GenericDAOHibernateImpl`) již má odkaz na Hibernate `sessionFactory` (ta již umožňuje přímý objektový přístup do relační databáze).

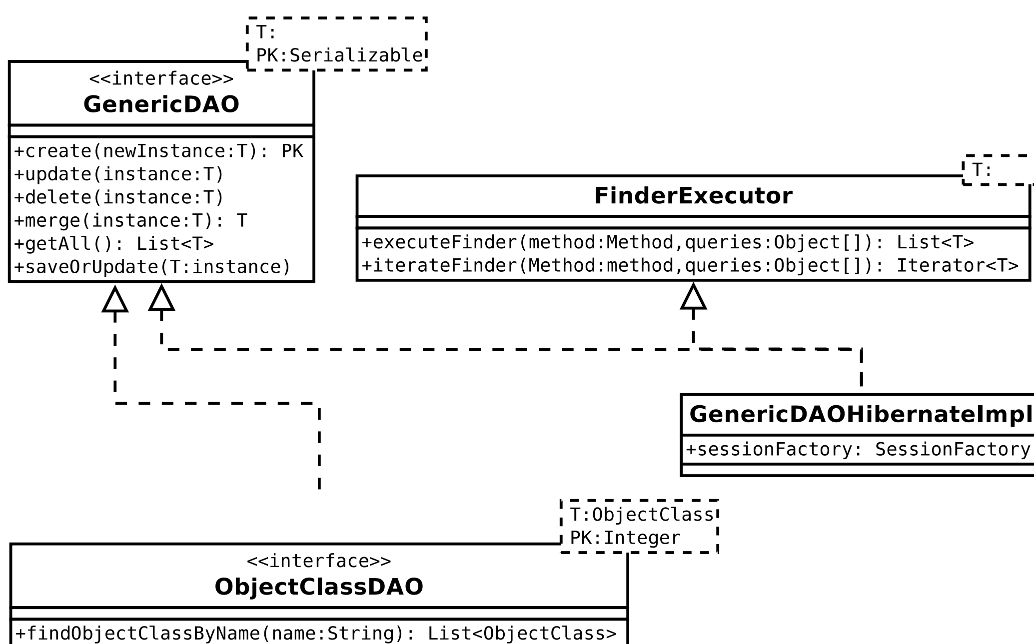


Diagram 6.3: Diagram tříd GenericDAO.

Nicméně z diagramu je patrné, že chybí třída implementující rozšiřující rozhraní, která by tak mohla překrýt metodu s patřičným HQL dotazem pro získání třídy objektu podle jejího jména. Jistě bychom ji mohli vytvořit programově, nicméně přesně tomu jsme se chtěli hned zpočátku vyhnout (a v programu žádné takové třídy skutečně nenajdeme).

Lepší možnost je využít znázorněný `FinderExecutor`, jehož implementace za použití malé knihovny v balíku `genericdao` umožní HQL dotazy načítat přímo

z Hibernate frameworku na základě názvu mapované entity (to se děje v v metodě `executeFinder()`). HQL dotazy následně stačí pojmenovat stejně jako je název metody rozšiřujícího rozhraní, která má daný dotaz volat, a umístit je do mapovacího XML souboru entity. Samotná implementace je poté díky aspektovému programování doslova „poskládána“ přímo za běhu programu. Nově vzniknuvší *proxy DAO* dědí od `GenericDAOHibernate` a implementuje `ObjectClassDAO`, dotazy jsou poté načítány na základě aspektu přímo z `hbm.xml` souborů.

Injektování rozšiřujících rozhraní na generickou DAO implementaci je součástí konfiguračních souborů `persistence.xml` a `genericdao.xml`. Všechna DAO rozšiřující rozhraní a generické implementace jsou součástí balíku `dataaccess.dao`.

Hibernate Search

Jedním z požadavků na systém bylo i zajištění vyhledávání v datech některých entit na základě jejich atributů. To může opět velmi jednoduše zajistit framework Hibernate, který implementuje nepřímé indexované vyhledávání v databázi (čímž se liší od vyhledávání pomocí SQL nebo HQL). Nejenže je toto vyhledávání mnohem rychlejší, protože funguje na principu velmi sofistikované hashovací tabulky a nemusí mít přímý přístup do databáze, je také flexibilnější, protože programátor má možnost jeho implementaci programově ovlivnit. Hibernate sám zajistí správu indexování při jakékoliv změně v databázi (indexy jsou mimo jiné vytvořeny při každém startu aplikace). Jediný požadavek kladený na stranu programátora je vložení anotací do patřičných POJO objektů.

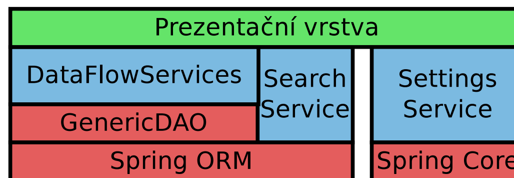
V aplikaci je poté vytvořena `SearchService`, která vyhledávání zprostředkovává. Ta sice obchází standardní DAO mechanismus a přistupuje přímo k Hibernate frameworku, nicméně svým určením patří do aplikační vrstvy.

Nastavení aplikace

Jedním z požadavků na systém je i možnost konfigurace některých časových primitiv, jako je například minimální a maximální délka periody rezervace atp. Tyto direktivy jsou uloženy ve formě *Java property* souboru pod jménem `rat.properties`. Pro jejich načítání nebylo třeba nic manuálně programovat, pouze stačilo nakonfigurovat Spring framework tak, aby tento soubor načetl při startu aplikace.

6.2.4 Servisní vrstva

Na Obrázku 6.3 je znázorněno jednoduché scéma servisní vrstvy. Část označená jako `DataFlowServices` je přímým pokračováním DAO objektů a poskytuje tedy rozhraní do databáze, nicméně tentokrát je možné v každé servisní třídě referencovat jakýkoliv DAO objekt z datové vrstvy. `SearchService` poté přistupuje přímo k `sessionFactory` ze Spring ORM.



Obrázek 6.3: Schéma servisní vrstvy.

DataFlowServices

Na Diagramu 6.4 je možné vidět základní architekturu té části servisní vrstvy, která kontroluje přístup do databáze. Základní implementace `GenericDataServiceImpl` váže bázový DAO objekt na servisní vrstvu a implementuje základní rozhraní. *Rozšiřující rozhraní* (v diagramu je to `ObjectClassService`) poté umožňuje definovat další rozšiřující metody a funkce.

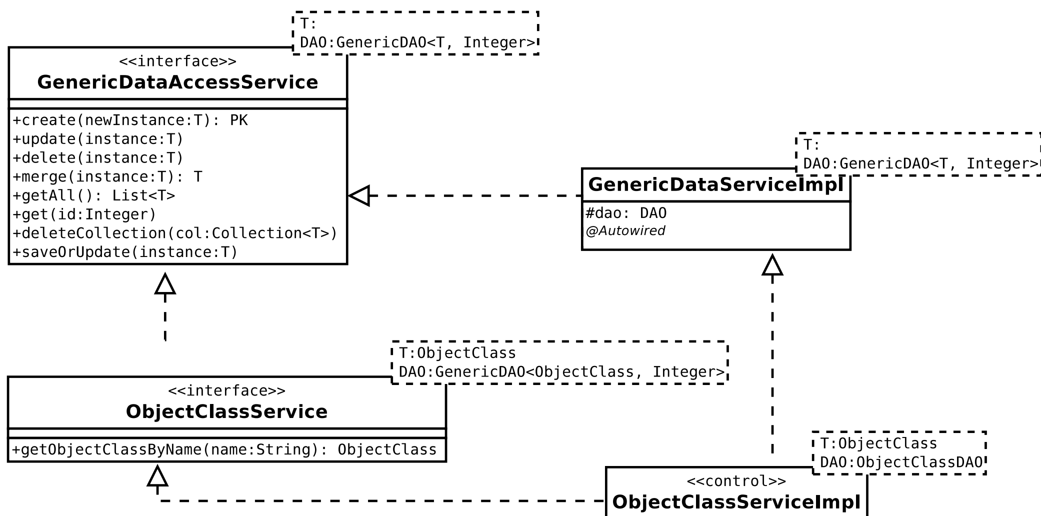


Diagram 6.4: Diagram tříd servisní vrstvy.

Servisní vrstva je z ostatních částí aplikace referencována přes tyto rozšiřující rozhraní. Díky tomu je možné velmi jednoduše tzv. *mockovat* servisní vrstvu a tím usnadnit testování nadřazených vrstev. Všechny jednotlivé servisní implementace jsou zaregistrovány do Spring aplikačního kontextu přes anotaci `@Service` a je možné použít pokročilé metody referencování jako je například *autowiring*.

Servisní vrstva pak zajišťuje i transakční řízení přístupu do databáze (přes anotaci metod `@Transactional`), požadavky specifikace na odstraňování objektů z databáze (to obvykle zajišťuje Hibernate přes *cascade delete*, bohužel tuto techniku není možné vždy použít) a zejména v sobě koncentruje veškerou aplikační logiku související se správou rezervací, jejich vyzvedávání, vracení atd. Podrobný popis je možné najít v programové dokumentaci. Kompletní implementace je v balíku `dataaccess.service`.

Settings Service

Úkolem je zpřístupnit konfigurační direktivy správy rezervací pro ostatní části systému. Spring Framework sám *injektuje* při vytváření instance potřebné parametry přímo z konfiguračního souboru a tyto jsou pak přístupné přes klasické *gettery*. V této servise jsou také inicializovány přednastavené hodnoty, které jsou použity v případě, kdy data z konfiguračního souboru neodpovídají očekávanému formátu.

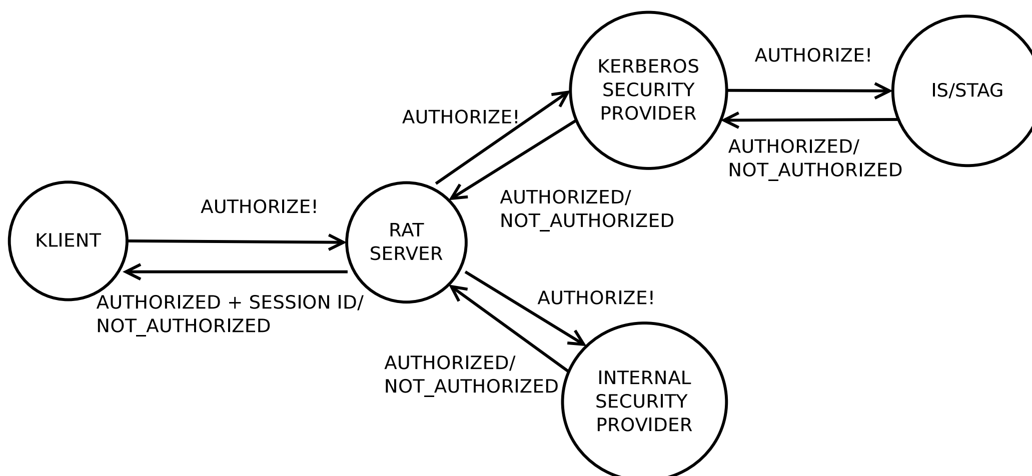
6.2.5 Spring Security

Součástí specifikace jsou i požadavky na uživatelské role systému a integraci s celouniverzitním kontem Orion. Uživatel se může autorizovat buď oproti interní databázi, kde bude uloženo heslo v hashované podobě, nebo je umožněno přihlášení proti celouniverzitnímu systému IS/STAG pomocí protokolu Kerberos – v tomto případě naopak nesmí být heslo v interní databázi uloženo. Pro implementaci byl použit modul Spring Security.

Spring Security je součást Spring Frameworku, která umožňuje pokročilou správu autorizace a uživatelských rolí. Pro každého uživatele, který byl úspěšně ověřen, vytvoří unikátní identifikátor (`sessionID`), který je klientovi uživatele zaslán jako reakce na pokus o přihlášení. Klienti se poté následně ověřují pomocí tohoto unikátního klíče, bez zadání jména a hesla. Framework také řeší správu rolí (viz dále) a umí na základě příchozího `sessionID` umožnit nebo zakázat přístup do systému, a to na třech úrovních – na úrovni volání metod, na úrovni URL požadavku a na úrovni vykreslovaných HTML stránek pomocí tzv. *security tagů* (viz dále).

Úplný popis tohoto modulu jde dalece nad rámec této práce. Pro naše účely postačí vědět, že Spring Security umožňuje programově vytvořit a zaregistrovat vlastní autorizační mechanismy (*Security Providers*). Framework poté ověřuje příchozí autorizační data pomocí zaregistrovaných autorizačních poskytovatelů a teprve v okamžiku, kdy některý z těchto poskytovatelů úspěšně ověří klientskou stranu, vygeneneruje framework příslušné `sessionId` a úspěšně autorizuje uživatele.

Jak ukazuje Obr. 6.4, bylo těchto vlastností využito pro implementaci požadavků na autorizaci uživatelů.



Obrázek 6.4: Základní schéma autorizace.

Security Implementor

Jak již bylo zmíněno v dřívějším textu, existují dva diametrálně odlišné způsoby autorizace. Obě jsou pevně svázané s uživatelským účtem (je možné mezi nimi volit při vytváření tohoto účtu). Abychom jednotlivé typy účtů od sebe oddělili, vznikl již ve fázi návrhu pro doménový objekt uživatele nový atribut – *Implementor*. To je vlastně jen textový řetězec, který umožňuje mapovat účet s autorizačním poskytovatelem. Každý zaregistrovaný autorizační poskytovatel poskytuje svůj unikátní *Implementor* idenfifikátor a proto je tedy označení atributu v databázi jako textový řetězec nepřesné – ve skutečnosti je tento řetězec jedním z dynamického výčtu identifikátorů zaregistrovaných autorizačních poskytovatelů (viz dále).

Implementace autorizačních poskytovatelů

Na Diagramu 6.5 je znázorněna základní implementace autorizačních poskytovatelů. Registrovat vlastní autorizačmí mechanismus do Spring Security frameworku je možné buď jako `AuthenticationProvider`, který má k dispozici celý autorizační kontext přihlašování (jméno i heslo), nebo jako `UserDetailsService`, která funguje v rámci zmíněného `AuthenticationProvideru`. Ta má k dispozici pouze jméno uživatele a jejím úkolem je buď vrátit instanci třídy `UserDetails` (která reprezentuje přihlášeného uživatele a musí do ní být uloženo všechno podstatné včetně hesla a role), nebo vyhodit výjimku `UserNotFoundException`.

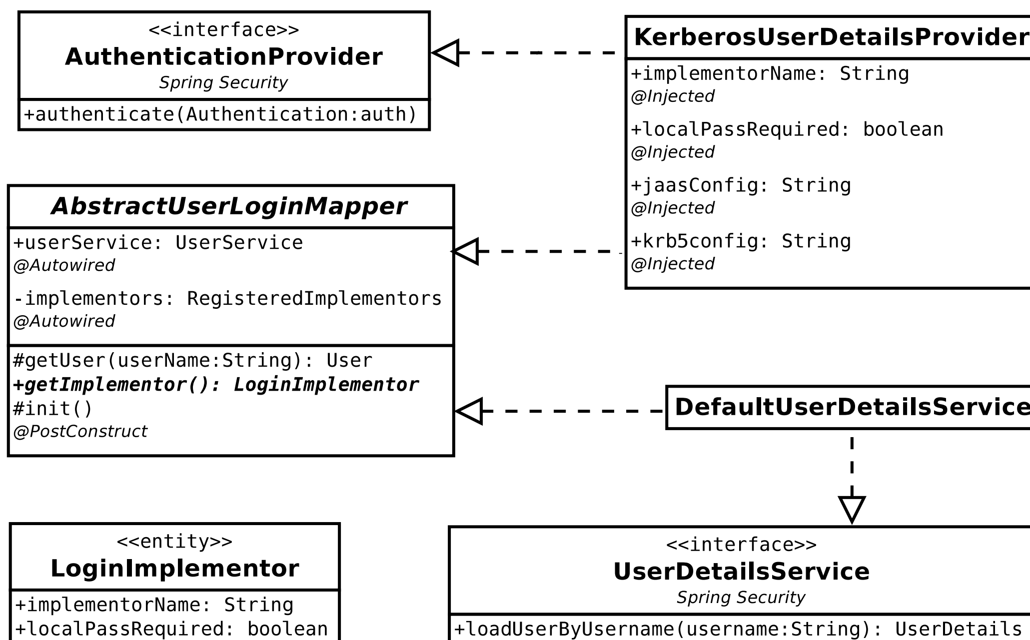


Diagram 6.5: Diagram tříd autorizačních poskytovatelů.

Každý autorizační poskytovatel musí implementovat metodu `getImplementor()`. Ta vrací instanci POJO třídy `LoginImplementor`, ve které jsou všechny důležité údaje o poskytovateli. V aplikaci existuje veřejný statický seznam všech takových objektů. Do něj se jednotliví poskytovatelé registrují hned po svém instancování v rámci metody `init()` (anotace `@PostConstruct`).

Poskytovatel autorizace oproti interní databázi implementuje rozhraní `UserDetailsService` a funguje v rámci přednastaveného autorizačního poskytovatele integrovaného ve Spring Security. Jak je vidět, vlastní implementace má referenci na `UserService` a tak může libovolně komunikovat s datovou vrstvou. Jejím jediným úkolem je tedy pouze najít uživatele v databázi podle jeho přihlašovacího jména, ověření hesla provede Spring Security.

Zajímavější situace nastává v případě autorizace oproti IS/STAG. Zde je již nutné znát celý přihlašovací kontext, protože jméno a heslo se musí odeslat na vzdálený autorizační server. Proto poskytovatel implementuje rozhraní `AuthenticationProvider`. Jak je patrné, i ten má referenci na servisní vrstvu zpřístupňující interní seznam uživatelů – to je kvůli mapování mezi uloženým záznamem a autorizačním poskytovatelem. Každý poskytovatel se nejprve pokusí vyhledat příslušného uživatele v databázi, a pokud nesedí *Implementor* uživatele se záznamem o aktuálně spuštěném poskytovateli, je autorizace rovnou odmítnuta (uživatel má jiného poskytovatele autorizace).

Důvod, proč jsou ve třídě tohoto poskytovatele členské proměnné přístupné přes `settery` je prostý – při registraci můžeme zaregistrovat tohoto poskytovatele vícekrát a *injektovat* mu jiné konfigurační soubory, název a další parametry. Tento poskytovatel je tedy obecně použitelný pro jakýkoliv Kerberos ověřovací protokol a lze tedy například vytvořit autorizaci uživatele oproti systémům jiných univerzit pouhou registrací v XML konfiguračním souboru, jak ukazuje tento výpis:

```
<bean id="kerberosZCU"
class="cz.zcu.fav.rat.security.service.KerberosUserDetailsProvider">
        <property name="jaasConfig" value="jaas.conf" />
        <property name="krb5Config" value="krb5.conf" />
        <property name="jaasContextName" value="WSKerberosAuth" />
        <property name="implementor" value="ZCU-KERBEROS" />
        <property name="localPasswordRequired" value="false" />
</bean>
```

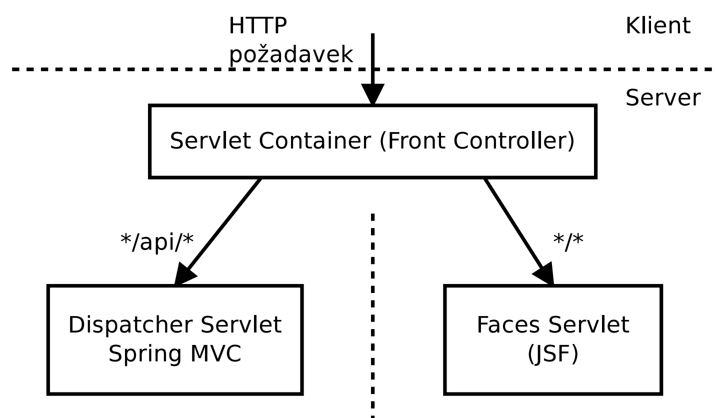
Uživatelské role

V systému jsou definovány pomocí výčtového typu role *ADMIN*, *MASTER* a *USER*. Přidávání nových rolí není bez rozsáhlých programových změn možné. Každý uživatel má v databázi uloženu svoji roli v systému. Tato role je při úspěšné autorizaci uložena do aplikačního kontextu Spring Frameworku v rámci již zmiňované instance třídy `UserDetails`. Autorizace přístupu na základě rolí je plně automatická a je řízena pouze za pomoci anotací, konfiguračních souborů a *security tagů*. Na základě anotací je zabezpečen například přístup k některým metodám servisní vrstvy, které editují databázové položky (např. role *USER* není oprávněna mazat objekty).

Implementace všech zmíněných bezpečnostních mechanismů se nachází v v balíku `security`.

6.2.6 Prezentační vrstva

Aplikační vrstva se skládá ze dvou velkých komponent – JSF pro webové rozhraní a Spring MVC pro webové REST služby, které budou poskytované pro mobilní aplikace. Jak ukazuje schéma znázorněné na Obr. 6.5, aplikace rozhoduje na základě URL příchozího požadavku o tom, který framework bude dále HTTP požadavek obsluhovat.



Obrázek 6.5: Základní schéma filtrování požadavků.

6.2.7 Webové rozhraní

Architektura použití JSF nijak nevybočuje z již dříve rozebraných postupů. Pro většinu webových komponent a celkového grafického rozvržení a stylu webového rozhraní je využita zejména TLD knihovna PrimeFaces.

V programovém Java balíku `managed` je veškeré programové vybavení aplikace související s JSF. Balík se dále dělí na další sadu podbalíků:

- `managed.bean` – všechny *backing beany* pro *Faces* stránky.
- `managed.converter` – dodatečné konvertory, které transformují data z *backing beans* do podoby, která je zobrazitelná v TLD komponentách.
- `managed.locale` – utility, které zpřístupňují lokalizované řetězce JSF do programové části.
- `managed.validator` – dodatečné validátory formulářů, které nelze kontrolovat přímo pomocí komponent TLD knihoven.

Využívá se integrovaný šablonovací systém *Facelets*. Všechny *Faces* stránky jsou umístěny mimo zdrojové kódy ve složce `WebContent`. S ohledem na mapování Spring Security je rozdělení následující:

- `*/*` – stránky, na které mohou přistupovat uživatelé i bez autorizace (iCal rozhraní).
- `*/templates/*` – šablony, ze kterých se skládá webové rozhraní systému.
- `*/login/*` – stránky určené pro přihlášení.
- `*/secured/*` – stránky zobrazitelné pro všechny přihlášené role.
- `*/admin/*` – stránky zobrazitelné pouze pro roli *Administrátor*.

Integrace JSF – Spring

Bohužel, Spring Framework nemá plnou podporu pro integraci s JSF (a naopak). To přináší některé problémy, které bylo třeba uspokojivě vyřešit.

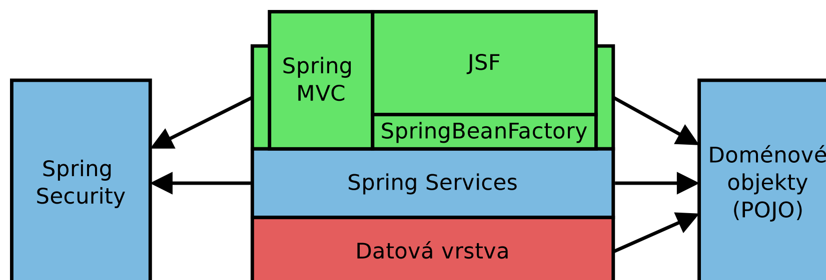
Spring Security poskytuje možnost vytvářet *security tagy*, které vytvářejí restrikce na vykreslování jejich vnořených elementů (neboli JSF komponent) na základě role přihlášeného uživatele. To velmi dobře funguje v prostředí JSP servletů. V JSF ale nexistuje integrovaná Spring Security TLD knihovna, a je tedy nutné ji vytvořit a naimportovat ručně. Knihovna je umístěna v `WebContent/WEB-INF` pod názvem `springsecurity.taglib.xml`.

Větším problémem je však samotné spojení *backing beans* s aplikačním kontextem Spring Frameworku (*WebApplicationContext*). JFS má vlastní programový kontext (*FacesContext*) a vlastní správu komponent. Lze tedy zjednodušeně říci, že JSF komponenty (validátory, konvertory atp.) „nevidí“ servisní vrstvu, která je v aplikačním kontextu Springu.

Jedno z možných řešení je vyjmout všechny komponenty z JSF kontextu a spravovat je Spring Frameworkem. Toto řešení funguje, ale má mnoho omezení – není možné používat žádné JSF specifické anotace, tedy nelze například dost dobře určovat *scope* nebo jednoduše *injektovat* parametry z URL. Spring má sice vlastní sadu anotací pro *scope*, ty však nejsou plně kompatibilní s JSF (a některé zcela chybí).

Jednodušší řešení je vytvořit programový můstek mezi oba frameworky. V balíku `managed.beans` je umístěna statická třída `SpringBeanFactory`, která umožňuje získat reference na libovolnou instanci třídy z aplikačního kontextu Spring Frameworku, která byla zaregistrována jako `@Component`, `@Service` nebo `@Controller`.

Komponenty je možné získat na základě zaregistrovaného jména nebo rozhraní. Spojení JSF a servisní vrstvy je ukázáno na Obr. 6.6.



Obrázek 6.6: Spojení Spring a JSF.

JSF komponenty mohou být v závislosti na svém *scope* dočasně serializovány. Například kalendáře vyžadují AJAX komunikaci mezi klientem a serverem. Proto jsou příslušné *managed beans* anotovány jako `@ViewScoped` (instance je udržována dokud uživatel neopustí stránku), aby byl vždy zachován jejich stav včetně *Modelu*. Je nutné označit všechny reference na objekty ze servisní vrstvy jako `transient`, aby nedošlo k jejich serializaci spolu s držitelem reference.

Tisk QR kódů

Pro tisk QR kódů byla použita open-source knihovna *itextpdf*. Programový kód nezávislý na JSF je umístěn v balíku `qrcode`.

Jak již bylo zmíněno, JSF je technologie ve které primární roli hraje *Pohled*, tedy konkrétní *Faces* stránka. Ty jsou nicméně primárně orientovány na generování HTML kódu. Pro tisk QR kódu je potřeba změnit HTTP odpověď na `Content-Type` na `application/pdf` a místo HTML zapsat přímo PDF soubor. To se však nedá realizovat pomocí žádné TLD knihovny.

Je tedy nutné celou architekturu JSF „překlopit" z MVP do MVC tak, abychom z *backing beany* udělaly *Controller*, který použije výše zmíněného balíku `qrcode` a zapíše výstup (PDF soubor) přímo do HTTP odpovědi.

Je nejprve nutné předat odpovědnost za vygenerování odpovědi z JSF stránky příslušné *backing beaně*:

```
<f:event type="preRenderView" listener="#{qrprinterbean.render}" />
```

Metoda `render()` poté načte HTTP odpověď, vyresetuje ji a následně ji může znovu vytvořit:

```
HttpServletResponse response = externalContext.getResponse();
response.reset();
response.setContentType("application/pdf");
OutputStream browserStream = response.getOutputStream();
ObjectQRCodeGenerator gen = ...načti ze SpringBeanFactory...
gen.createPDF(browserStream, objectId);
browserStream.close();
facesContext.responseComplete();
```

Samotný generátor PDF jenom zapíše PDF dokument do výstupního proudu. Je možné tedy používat tuto třídu univerzálně pro jakýkoliv výstupní proud.

Export kalendáře do iCal formátu

Pro export kalenáře je je použita knihovna `iCal4j`. Programový kód nezávislý na JSF je v balíku `ical`. Princip změny fungování JSF pro export kalenáře je totožný s vytvářením QR kódů.

6.2.8 REST webové služby

Pro komunikaci mezi mobilní aplikací a centrálním serverem byly vytvořeny REST služby, které zajišt'ují mobilním aplikacím všechnu potřebnou funkcionalitu. Služby jsou zabezpečeny pomocí Spring Security obdobně jako webové rozhraní. Po každé úspěšné HTTP autorizaci uživatele server vloží do odpovědi `sessionId`, které je možné používat jako autorizační token v následné komunikaci. Tím je sice porušena bezestavovost REST rozhraní, na druhou stranu není nutné klienta při každém požadavku ověřovat proti vzdálenému serveru (pokud to uživatelský účet klienta vyžaduje), což přináší znatelné urychlení komunikace. Některé služby jsou také orientovány spíše procedurálně než datově (viz dále). Implementace je umístěná v balíku `rest`.

Zde budou uvedeny pouze implementační detaily. Úplný popis REST webových služeb je možné najít v Příloze B.

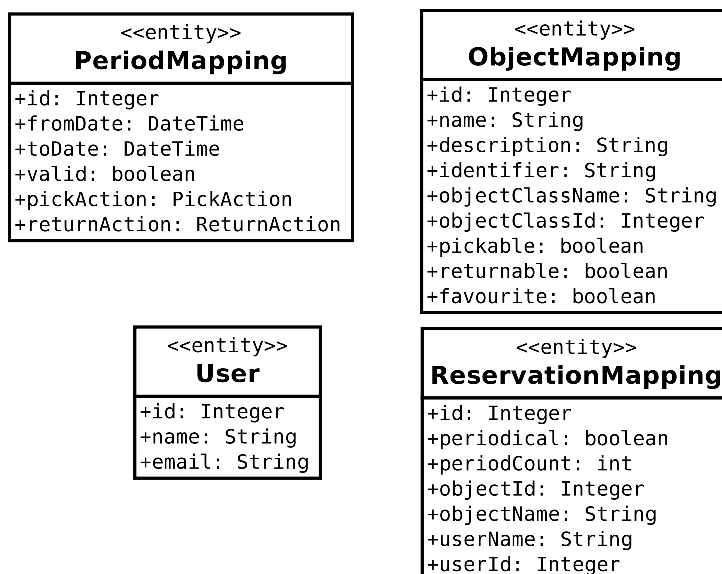
Serializace dat

Jako přenosový formát dat byl zvolen XML. Jednotlivé objekty jsou serializovány a deserializovány pomocí technologie JAXB (*Java Architecture for XML Binding*). Nicméně protože serializace přímo doménových objektů není příliš vhodná (svázání

s Hibernate `session`, zanášení modelu netypickými anotacemi) a navíc chceme minimalizovat přenosy dat, byla vytvořena nová sada *mapovacích objektů*, které lépe reflektují potřeby mobilní aplikace a navíc umožní oddělit serializační anotace od doménového modelu. S ohledem na struktuxu XML dokumentu lze objekty rozdělit dále na *mapovací doménové objekty* a *mapovací kontejnery*.

Úkolem mapovacích kontejnerů je umožnit přenášení více objektů v rámci jedné reference, reprezentují tedy vlastně seznam instancí stejné třídy. Pomocí JAXB anotace `@XmlRootElement` je poté definován název kořenového elementu takového seznamu při XML serializaci. Takovými kontejnery jsou např. `PeriodsMapping`, `ReservationsMapping`, atd.

Mapovací doménové objekty jsou poté reprezentace doménových objektů, které jsou ovšem přímo určené k serializaci. Jejich atributy se přitom od doménových objektů mírně liší, jak ukazuje Obr. 6.7.



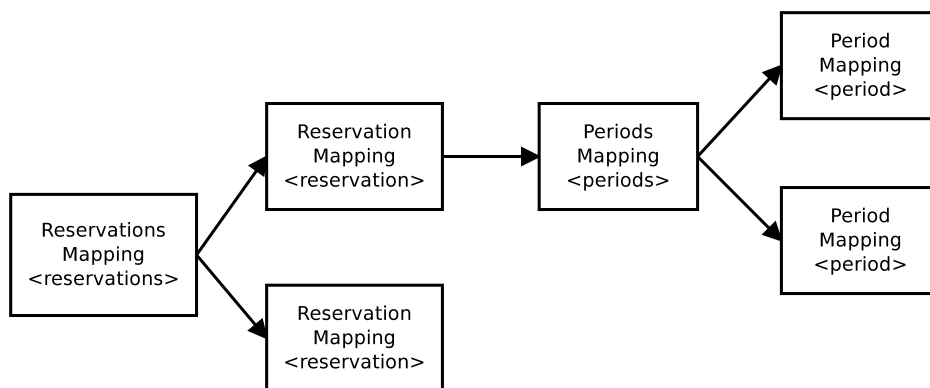
Obrázek 6.7: Základní mapovací objekty.

Například `ObjectMapping` nahrazuje referenci na třídu objektu dvojicí atributů `objectClassId` a `objectClassName`. Dále dědí její `pickable` a `returnable` atributy. Cílem těchto změn je poskytnout maximum informací v jednom požadavku a umožnit v rámci v dalších požadavků jejich zpřesnění (mobilní aplikace například v profilu rezervace zobrazí jméno objektu a pokud bude chtít uživatel zobrazit i další informace o tomto objektu, je možné zaslat požadavek příslušné webové službě na základě atributu `objectId`).

Třídy těchto mapovacích objektů jsou umístěny v balíku `rest.beans`.

Přemapování doménových objektů

Pro přemapování je využita kaskáda *factory* tříd. Ty jsou umístěné v balíku `rest.factory` a jsou zaregistrovány do Spring Frameworku jako `@Component`. Každá z nich zajišťuje přemapování jednoho typu objektu. V případě, kdy je vazba mezi doménovými objekty typu 1:N (například `Reservation` – `Period`), je využito kontejnerových mapovacích objektů. V případě přemapování kolekce instancí třídy `Reservation` je tedy zavolána mapovací metoda z `ReservationsMappingFactory`, ta vytvoří novou instanci `ReservationsMapping`, poté přemapuje každý objekt z kolekce pomocí *factory* `ReservationMappingFactory`, a tak dále. Každá úroveň zanoření této kaskády pak je vlastně úroveň zanoření výsledného XML dokumentu, který vznikne JAXB serializací vzniknuvšího `ReservationsMapping` objektu (Obr. 6.8). *Factory* umí mapovat i obráceně, tj. z mapovacích objektů do doménových.



Obrázek 6.8: Vztah mapovacích objektů a XML.

Spring MVC

Výše zmíněných principů je využito k vytváření XML dokumentů, které jsou přikládány k HTTP odpovědi nebo jsou deserializovány z HTTP požadavku. V balíku `rest.controller` jsou třídy (*Řadiče*), jejichž metody jsou za pomoci Spring MVC přímo použité pro vytvoření webových služeb. U každé metody je nutné pomocí anotací určit podporovaný HTTP protokol a sadu URL parametrů, které budou *injektovány* samotným Spring Frameworkem. Metody mají přímý přístup do servisní vrstvy aplikace a využívají zmíněného *factory* balíku pro konvertování příchozích XML souborů na doménové objekty a naopak. Roli *Modelu* pak hrají příchozí nebo odchozí data a jako *Pohled* je možné interpretovat JAXB *Marshaller*, který je zaregistrován pro každý *Řadič* v souboru `applicationContext.xml`.

REST error handling

Při konstruování odpovědi na HTTP požadavek může ve webové službě dojít k chybě aplikace a následně může být vyhozena výjimka. Abychom tyto stavy ošetřili a korektně namapovali do HTTP kódů, je v balíku `rest.errorhandling` komponenta `RestExceptionHandler`, která umožňuje zachytávání výjimek z aplikačního kódu.

Výjimka je vždy zachycena a přemapována na XML dokument se zprávou o chybě a dalšími konfigurovatelnými údaji (třída `XmlRestErrorConverter`). Ten je přiložen k HTTP odpovědi. Dále bylo nutné v souboru `RAT-servlet.xml` (konfigurační soubor *Spring Dispatcher servletu*) definovat chybový kód, který bude s výjimkou svázán:

```
<!-- 404 - Unknown -->
<entry key="UnknownResourceException" value="404, _exmsg" />

<!-- 403 - Unauthorized -->
<entry key="UnauthorizedAccessRestException" value="403, _exmsg" />

<!-- 500 (catch all): -->
<entry key="Throwable" value="500, error.internal" />
```

Jak je vidět, existují dvě namapované uživatelské výjimky pro situace, kdy je zadán špatný identifikátor zdroje nebo se uživatel snaží přistoupit ke zdrojů, které leží mimo jeho kompetence (například se pokouší editovat rezervaci, která není jeho). Všechny ostatní výjimky (instance rozhraní `Throwable`) se vyhodnocují jako interní chyba serveru.

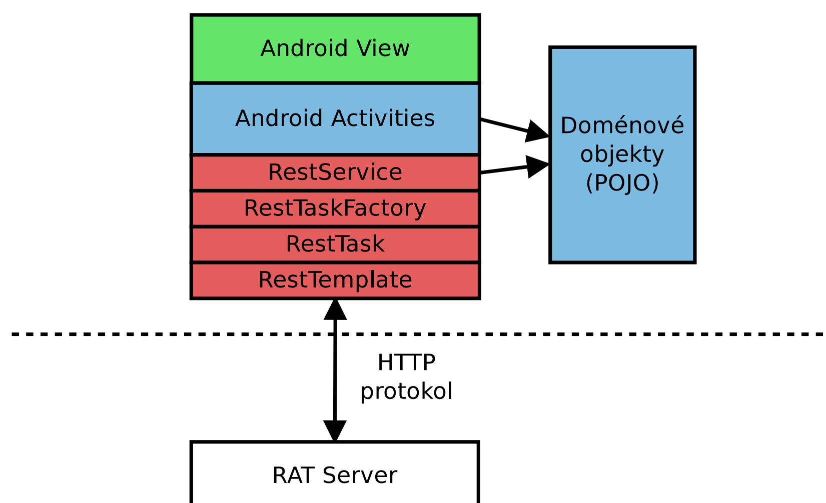
6.3 RatDroid

RatDroid je název klientské aplikace systému RAT pro operační systém Android. Tato aplikace se připojuje k REST webových službám centrálního serveru a zajišťuje uživatelům funkčnost specifikovanou pro mobilní zařízení v Kap. 4.

6.3.1 Základní architektura aplikace

Na Obrázku 6.9 je znázorněno zjednodušené schéma mobilní aplikace, včetně rozpisu největších užitých komponent. Jak je vidět, i mobilní aplikace sleduje základní

rozvržení třívrstvé architektury. Klient neobsahuje žádnou aplikační logiku s výjimkou základních validací uživatelského vstupu a víceméně hraje roli *tenkého klienta*.



Obrázek 6.9: Architektura a použité technologie.

6.3.2 Doménový model

Doménový model aplikace kopíruje strukturu mapovacích POJO objektů implementovaných na serveru. Při komunikaci se serverem dochází k serializaci a deserializaci těchto objektů. Protože v aplikačním frameworku Androidu chybí některé standardní Java knihovny, není možné použít např. JAXB technologii. Proto byl použit *Simple Framework*, který na těchto knihovnách implmementačně nezávisí. Tento framework poskytuje komponentu `SimpleXmlHttpMessageConverter`, kterou je možné zaregistrovat přímo do Spring REST komunikačního rozhraní (viz dále). Serializace je řízena na základě anotací.

Samotné doménové objekty jsou umístěny v balíku `bean`. Některé pomocné XML konvertory nutné pro serializaci objektů jsou v balíku `xml`.

6.3.3 Datová vrstva

Jako datovou vrstvu je možné interpretovat komunikační rozhraní mezi mobilním klientem a centrálním serverem. Cíle této vrstvy však zůstávají stejné – bylo třeba oddělit kód specifický pro REST komunikaci od vlastního aplikačního kódu. Dále bylo třeba zohlednit asynchronní povahu této komunikace. Veškerá implementace datové vrstvy se nachází v balíku `service`.

RestTemplate

Pro základní REST komunikaci je použita komponenta `RestTemplate` z knihovny *Spring For Android*. Spring zde již nehraje roli rozsáhlého aplikačního frameworku, jedná se spíše o malou knihovnu poskytující užitečnou funkcionalitu pro vzdálenou OAuth autorizaci a REST komunikaci.

Komponenta `RestTemplate` poté podporuje spoustu užitečných funkcí, jako je správa *Cookies*, HTTP autorizace, automatická deserializace dat přenášených jako XML nebo JSON atp.

RestTask

Komponenta `RestTask` je základní komunikační úloha aplikace. Bez ohledu na přenášená data nebo URL cílové webové služby jsou všechny uskutečněné pokusy o komunikaci řešeny jako instance objektu `RestTask`.

První vlastností této komponenty je realizace stavové komunikace na základě serverové `sessionId`. Nad rámec obvyklé REST komunikace je vystavěn mechanismus, který umožňuje přenášet jméno a heslo uživatele pouze pokud je to nezbytně nutné.

Samotná programová realizace poté počítá se dvěma pokusy o komunikaci s webovou službou . Při prvním pokusu je do HTTP požadavku vloženo `sessionId` (pokud existuje). Pokud se po odeslání požadavku ze serveru vrátí odpověď s HTTP kódem `404 - Unauthorized`, znamená to, že vypršela platnost `sessionId`. V těchto situacích je odeslán naprosto stejný HTTP požadavek jako v prvním kroku, tentokrát však s příslušným jménem a heslem. Z odpovědi je poté (pokud tato také nekončí chybovým kódem, v tom případě má uživatel špatné jméno nebo heslo) vyjmuto nové `sessionId`, které je poté používáno v další komunikaci.

Dalším úkolem této komponenty je vytvoření asynchronní komunikace. Třída dědí od systémové `AsyncTask` a vlastní výkonný kód je pouštěn asynchronně vůči hlavnímu vláknu aplikace. Aby bylo možné nějak notifikovat zbytek aplikace o výsledku komunikace, musí se každé instanci třídy `RestTask` předat implementace rozhraní `IRestListener`.

RestTaskFactory

Jak již název napovídá, `RestTaskFactory` zajišťuje vytváření jednotlivých komunikačních úloh. Dále udržuje referenci na `sessionId` a rekonfiguruje nastavení

`RestTemplate` komponenty pokud uživatel změní nastavení komunikace.

RestService

Třída `RestService` je implementace již zmiňovaného návrhového vzoru *Data Gateway*. Obsahuje sadu metod, které jsou namapovány na jednotlivé webové služby a je tedy skutečným rozhraním mezi REST webovými službami a doménovým modelem aplikace.

6.3.4 Aplikační vrstva

Aplikační vrstva je realizována pomocí řady tříd, které obvykle dědí od třídy `SkeletonRestActivity`. Tato základní třída rozšiřuje známou systémovou třídu `Activity` a současně implementuje rozhraní `IRestListener` tak, aby každá *aktivita* aplikace mohla reagovat na dokončenou komunikaci se serverem a aktualizovat prezentační vrstvu. Protože může být jakákoliv instance *aktivity* kdykoliv mobilním systémem za jistých okolností „uklizena“ (*destroyed*), je již na úrovni `SkeletonRestActivity` zajištěno podchycení takové události a korektní ukončení komunikace, pokud tato zrovna probíhá.

Obdobné třídy jako je `SkeletonRestActivity` existují i pro další komponenty systému – *fragmenty* (`SkeletonRestFragmentActivity`) a vícepoložkové *aktivity* (`SkeletonRestListActivity`).

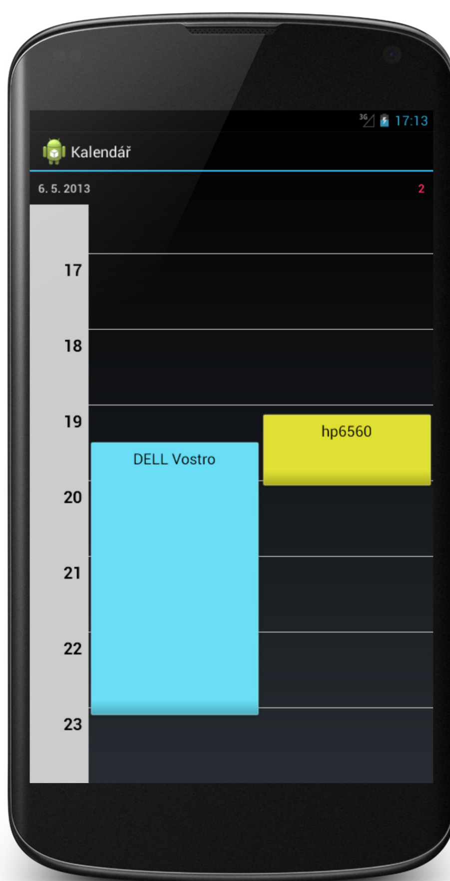
6.3.5 Prezentační vrstva

Pohled prezentační vrstvy je v aplikačním prostředí Androidu obvykle zapisován ve formě jako XML dokumentů. Každá *aktivita* poté může sobě přiřadit patřičný *Pohled* a dále s ním pracovat. *Pohled* je možné z *aktivit* vytvářet i programově, čímž se prezentační vrstva přibližuje například klasickému frameworku Swing.

Tento přístup neposkytuje žádné možnosti *bindování* a dalších pokročilých programovacích technik. Vše musí být programovově řízeno z *aktivit* pomocí posluchačů, včetně reakcí na události apod., což znatelně znepřehledňuje aplikační kód. Na druhou stranu lze v XML zapsané (ač velice jednoduché) *Pohledy* používat u celé řady aktivit, což se například u technologie JSF realizuje jen velice obtížně.

6.3.6 Shrnutí

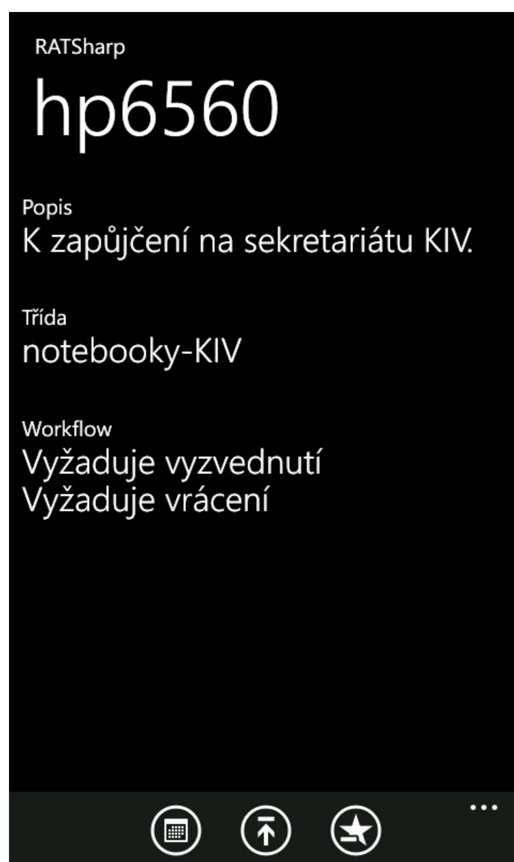
Na Obr. 6.10 je ukázka jedné z obrazovek aplikace *RatDroid*. Díky napojení na webové služby centrálního serveru je možné efektivně vytvářet a spravovat vlastní rezervace, kdy jedinou podmínkou je dostatečně stabilní připojení k síti. Aplikace se maximálně snaží využít možnosti dotykového displeje a je možné ji ovládat jak v horizontálním tak ve vertikálním módu přístroje.



Obrázek 6.10: Ukázka mobilní aplikace *RatDroid*.

6.4 RatSharp

Nad rámec této práce byla vytvořena mobilní aplikace s názvem *RatSharp*. Tato aplikace je určena pro mobilní operační systém Windows Phone 7 a implementuje stejnou sadu požadavků jako klient pro systém Android. Je tedy plně použitelná v rámci celého informačního systému RAT. Ukázka jedné z obrazovek této aplikace je na Obr. 6.11.



Obrázek 6.11: Ukázka mobilní aplikace *RatSharp*.

6.5 Testování a nasazení systému

Během celého vývoje byl dostupný produkční server, na kterém docházelo k automatickému stažení aktuální revize zdrojových kódů ze *Subversion* repozitáře projektu, sestavení webové aplikace a nasazení na webový kontejner Tomcat. V raných fázích vývoje se jednalo pouze o pomůcku, jak průběžně kontrolovat sestavitelnost aplikace i na jiných systémech než je počítač vývojáře.

V okamžiku, kdy byla hlavní serverová část již použitelná spolu s mobilní aplikací, byl produkční server spuštěn spolu na veřejné IP adrese a funkcionalita celého systému byla průběžně kontrolována jak autorem systému, tak zadavatelem práce. Pro účely hlášení chyb byl nasazen *bug tracking system*, do kterého byly jednotlivé nalezené chyby průběžně hlášeny ve formě úloh určených k vyřešení.

Mobilní aplikace byla vyvíjena a testována na referenčním zařízení *Nexus S*, které je ve vlastnicví Katedry informatiky a výpočetní techniky. Byl vytvořeno několik typických scénářů užití systému zaměstnanci katedry, souvisejících zejména se souběžnými rezervacemi téhož objektu různými uživateli a schopností mobilní aplikace rychle vyzvednout objekt, na který předtím nebyla vytvořena žádná rezervace (funkce *Vyzvedni*).

Scénáře užití byly průběžně ověřovány na výše zmíněném reálném zařízení. Dále byly vybrány některé existující movité i nemovité objekty nalézající se ve vlastnictví katedry. Tyto byly následně označeny patřičnými QR kódy a byly testovány reálné možnosti systému efektivně spravovat výpůjčky těchto objektů. Některé požadavky byly na základě tohoto testování shledány jako nevyhovující a byly ze specifikace odstraněny (tyto nakonec neimplementované požadavky nejsou součástí této práce), naopak některé vznikly během uživatelského testování jako snaha o zvýšení použitelnosti systému (například funkce *oblíbených objektů* nebo možnost zobrazit profil objektu přímo z editačních obrazovek rezervací). Díky tomu je možné říci, že vzniklý systém plně pokrývá požadavky a nároky na něj kladené a je schopen reálného nasazení.

7 Závěr

Cílem této práce bylo zobecnit problém správy výpůjček a rezervací objektů movitých i nemovitých, a to na základě případové studie stávajícího řešení na Katedře informatiky a výpočetní techniky při Západočeské univerzitě v Plzni. Nejprve byl vytvořen obecný logický model správy výpůjček a rezervací a definovány základní termíny a vazby v tomto modelu. Následně došlo k vytvoření kompletní specifikace požadavků na informační systém, který se snaží problém správy výpůjček a rezervaci efektivně řešit za využití unikátních možností mobilních zařízení.

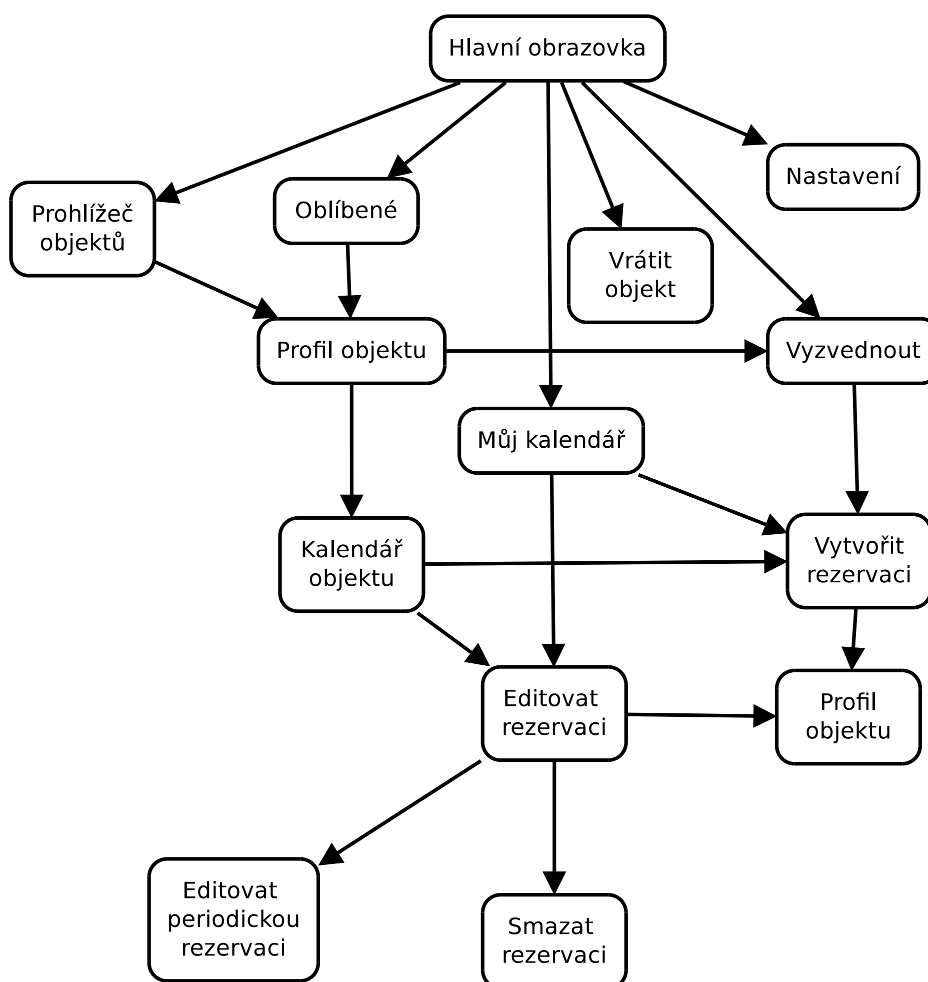
Na závěr byla předvedena konkrétní realizace tohoto informačního systému. Tato realizace vytváří architekturu klient-server, ve které jednotlivé klienty představují mobilní zařízení postavené na platformě Android. Tyto klienti se připojují k centrálnímu serveru za účelem vytváření a správy rezervací objektů, kdy každý objekt je jednoznačně identifikovýn pomocí QR kódu, který je na objektu fyzicky umístěn. Server také poskytuje pohodlné webové rozhraní. Je možné se přihlašovat s využitím uživatelských účtů Orion celouniverzitního systému IS/STAG.

Práci považuji za úspěšnou a tato by měla splňovat všechny požadavky, které na ni byly kladeny. Nad rámec původního zadání byla implementována klientská aplikace i pro mobilní platformu Windows Phone.

Možnost rozšíření této práce spočívá zejména v implementaci dalších způsobů přihlášení, kupříkladu za využití účtů webových služeb jako je GMail nebo Facebook a další sbližování těchto služeb se systémem. Dále je jistě žádoucí vytvořit klientské aplikace i pro další mobilní platformy tak, aby byla použitelnost systému co možná nejširší.

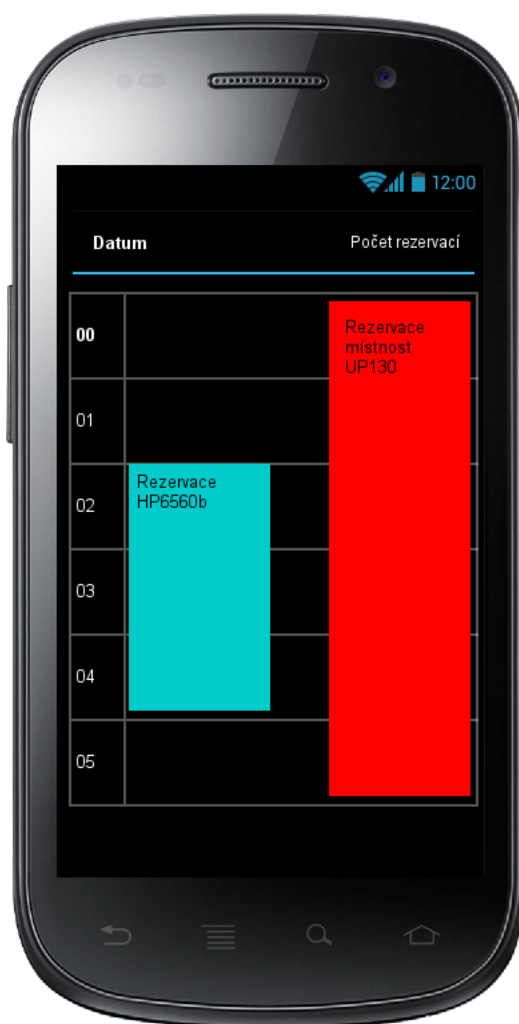
A Rozvržení mobilní aplikace

V této příloze je ukázáno základní schéma mobilní aplikace a některé návrhy obrazovek, které vznikly v rámci specifikace požadavků. Těmto návrhům by se měla finální implementace co nejvíce přiblížit. Obrázek A.1 ukazuje možné průchody obrazovkami mobilní aplikace. Ty při průchodu z hlavní obrazovky vytváří zásobník předchozích obrazovek a je možné se v cestě grafem vracet pomocí tlačítka *Zpět*, které by mělo at' už v hardwarové nebo softwarové podobě každé zařízení s mobilní platformou Android implementovat.



Obrázek A.1: Graf průchodu mobilní aplikací.

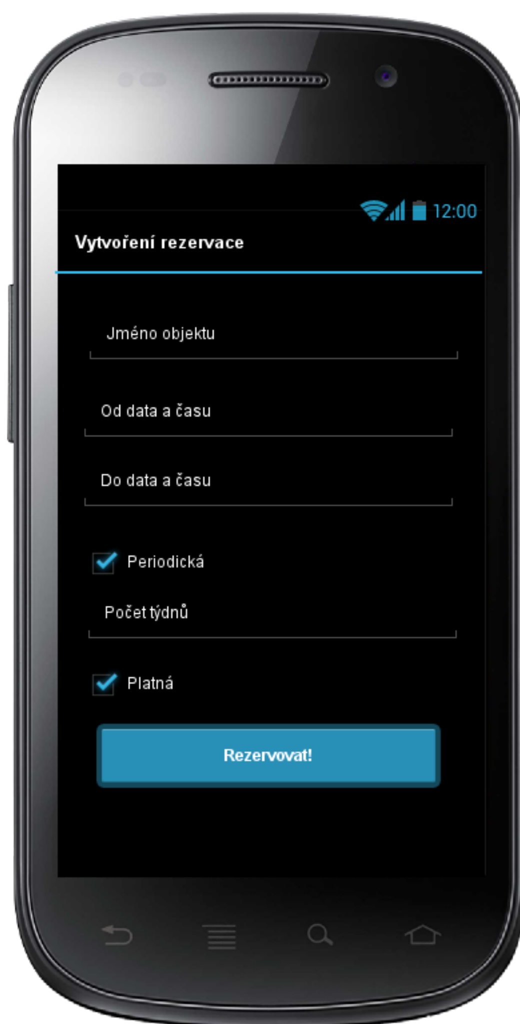
Obrázek A.2: Rozvržení hlavní obrazovky aplikace.



Obrázek A.3: Rozvržení měsíčního kalendáře.

Obrázek A.4: Rozvržení měsícního kalendáře.

Obrázek A.5: Kritéria vyhledávání objektů.



Obrázek A.6: Obrazovka pro vytváření a editaci rezervací.

B Popis webových REST služeb

V této příloze je uveden výčet všech webových služeb, které poskytuje centrální server informačního systému RAT. Služby vyžadují HTTP autorizaci a po přihlášení je vytvořena pro klienstkou stranu *session*, v rámci které je poskytován kontext pro přihlášeného uživatele (viz dále). URL všech služeb začíná relativní cestou `/api/rest/*`.

Na přiloženém médiu jsou uloženy XSD schémata XML reprezentace přenášených dat.

Seznam služeb

URI:	/users/user/name
HTTP Metoda:	GET
Atributy:	name=(String)
Popis:	Vrátí uživatele podle jeho přihlašovacího jména, nebo HTTP 404 - Unknown, pokud uživatel neexistuje.
XSD schéma:	user.xsd

URI:	/users/user
HTTP Metoda:	GET
Atributy:	id=(Integer)
Popis:	Vrátí uživatele podle jeho identifikátoru nebo HTTP 404 - Unknown, pokud uživatel neexistuje.
XSD schéma:	user.xsd

URI:	/users/user/favourites
HTTP Metoda:	GET
Popis:	Vrátí seznam objektů oblíbených uživatelem, jehož session je právě aktivní.
XSD schéma:	objects.xsd

URI:	/users/session/properties
HTTP Metoda:	GET
Popis:	Vrátí globální nastavení serveru, související s vytvářením rezervací (důležité pro mobilní kalendář).
XSD schéma:	properties.xsd

URI:	/users/session/reservations/occupation
HTTP Metoda:	GET
Atributy:	month=(Integer) year=(Integer)
Popis:	Vrátí ty dny v měsíci, ve kterých má majitel session vytvořené nějaké rezervace, a počet těchto rezervací v každém takovém dni.
XSD schéma:	occupations.xsd

URI:	/users/session/reservations/notreturned
HTTP Metoda:	GET
Popis:	Vrátí všechny nevrácené periody rezervací, které patří uživateli vlastnící session, jsou označeny jako nevrácené a je možné je vrátit.
XSD schéma:	reservations.xsd

URI:	/users/session/favourite
HTTP Metoda:	POST
Atributy:	id=(Integer)
Popis:	Přidá nový objekt do seznamu oblíbených objektů uživatele, který je majitelem session, a vrátí tento aktualizovaný objekt. Pokud objekt neexistuje, vrátí HTTP 404 - Unknown.
XSD schéma:	objects.xsd

URI:	/reservations/reservation
HTTP Metoda:	GET
Atributy:	id=(Integer)
Popis:	Vrátí rezervaci včetně všech period na základě předaného identifikátoru.
XSD schéma:	reservations.xsd

URI:	/users/session/reservation
HTTP Metoda:	DELETE
Atributy:	id=(Integer)
Popis:	Odstraní rezervaci patřící majiteli session ze systému. Pokud rezervace neexistuje, vrátí HTTP 404 - Unknown. Pokud je rezervace vedena na jiného uživatele, vrátí 401 - Unauthorized. Jinak vrátí odstraněnou rezervaci včetně všech jejích period.
XSD schéma:	reservations.xsd

URI:	/users/session/reservations
HTTP Metoda:	GET
Atributy:	since=(yyyy-MM-dd-HH-mm) to=(yyyy-MM-dd-HH-mm)
Popis:	Vrátí všechny periody (včetně jejich rezervací), jejichž časové úseky se alespoň částečně kryjí s časovým úsekem stanoveným URL atributy a jejichž majitelem je majitel session.
XSD schéma:	reservations.xsd

URI:	/users/session/reservation
HTTP Metoda:	PUT
Popis:	Vytvoří nové rezervace pro majitele session. Nevrací bud' žádnou rezervaci pokud se vytvoření podařilo, nebo tu rezervaci, která blokuje vytvoření rezervace nové.
XSD schéma:	reservations.xsd

URI:	/users/session/reservation/period
HTTP Metoda:	POST
Atributy:	since=(yyyy-MM-dd-HH-mm) to=(yyyy-MM-dd-HH-mm) valid=(true|false)
Popis:	Editování periody rezervace. Pokud perioda s předaným identifikátorem neexistuje, vrací HTTP 404 - Unknown. Pokud je rezervace periody vedena na jiného uživatele, vrátí 401 - Unauthorized. Vrací bud' editovanou periodu, nebo periodu, která editaci brání z důvodu kolize časů.
XSD schéma:	reservations.xsd

URI:	/objects/object/id
HTTP Metoda:	GET
Atributy:	id=(Integer)
Popis:	Vrátí objekt podle jeho identifikátoru. Pokud objekt s předaným identifikátorem neexistuje, vrátí HTTP 404 - Unknown.
XSD schéma:	objects.xsd

URI:	/objects/object/identifier
HTTP Metoda:	GET
Atributy:	identifier=(String)
Popis:	Vrátí objekt podle jeho textového identifikátoru. Pokud objekt s předaným identifikátorem neexistuje, vrátí prázdný seznam objektů.
XSD schéma:	objects.xsd

URI:	/objects/object/search
HTTP Metoda:	GET
Atributy:	type=(ALL|OBJ_NAME|CLASS_NAME|OBJ_DESCRIPTION) key=(String)
Popis:	Vrátí všechny nalezené objekty odpovídající vyhledávacím kritériím.
XSD schéma:	objects.xsd

URI:	/objects/session/object/pickup
HTTP Metoda:	POST
Atributy:	id=(Integer)
Popis:	Pokusí se o vyzvednutí objektu s předaným identifikátorem majitelem session. Vrátí buď' prázdný seznam pokud není jak rezervaci vyzvednout, cizí blokující rezervaci (pokud existuje), nebo vyzvednutou rezervaci, kterou majitel předtím vytvořil.
XSD schéma:	reservations.xsd

URI:	/object/reservations
HTTP Metoda:	GET
Atributy:	since=(yyyy-MM-dd-HH-mm) to=(yyyy-MM-dd-HH-mm) id=(Integer)
Popis:	Vrátí všechny periody rezervací vytvořených na objekt s předaným identifikátorem, jejichž časové rozmezí se alespoň částečně kreje s časovým rozpětím z parametrů.
XSD schéma:	reservations.xsd

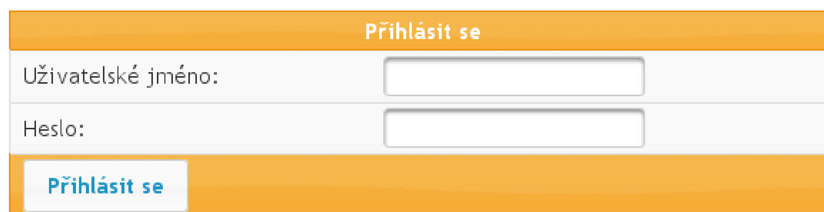
URI:	/objects/object/reservations/occupation
HTTP Metoda:	GET
Atributy:	month=(Integer) year=(Integer) id=(Integer)
Popis:	Vrátí ty dny v měsíci, ve kterých je vytvořena nějaká rezervace na objekt s předaným identifikátorem, a počet těchto rezervací v každém takovém dni.
XSD schéma:	occupations.xsd

C Uživatelská příručka

V této příloze bude uveden stručný manuál k používání webového rozhraní a mobilní aplikace *RatDroid* informačního systému RAT.

C.1 Webové rozhraní

Pro používání webového rozhraní je nejpre nutné zadat jméno a heslo na přihlašovací obrazovce. Bez tohoto aktu není možné žádným způsobem se systémem pracovat. Po úspěšném přihlášení bude uživatel „vpuštěn“ do systému a bude mu umožněno provádět úkony náležející jeho roli (Obr. C.1).



Obrázek C.1: Přihlašovací dialog.

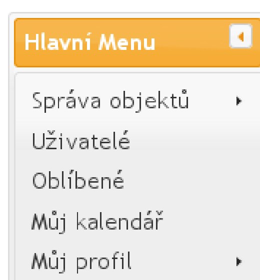
C.1.1 Základní navigace

Po úspěšném přihlášení bude po levé straně obrazovky dostupné navigační menu, které umožní uživateli přepínání mezi jednotlivými hlavními obrazovkami (Obr. C.2). Rozvržení tohoto menu je následující:

- *Správa objektů* – prohlížení všech objektů a tříd objektů v systému a provádění akcí nad objekty, které příslušejí roli přihlášeného uživatele.
- *Uživatelé* – prohlížení všech uživatelů v systému a provádění těch akcí nad uživatelskými účty, které příslušejí roli přihlášeného uživatele.
- *Oblíbené* – zobrazení oblíbených objektů přihlášeného uživatele.
- *Můj kalendář* – zobrazení kalendáře všech rezervací patřících přihlášenému uživateli.

- *Můj profil* – tato položka obsahuje možnosti správy uživatelského účtu přihlášeného uživatele, tj. editaci profilu, změnu hesla a odhlášení ze systému.

Menu je možné kdykoliv skrýt tlačítkem umístěným v pravém horním rohu menu.



Obrázek C.2: Navigační menu webového rozhraní.

C.1.2 Vícepoložkové zobrazení

Zobrazení objektů, tříd objektů, uživatelů a oblíbených objektů má vždy stejné rozvržení, proto jej popíšeme jako generickou komponentu vícepoložkového zobrazení. Ta je realizována jako fitrovatelná stránkovací tabulka (Obr. C.3).

Správa objektů

	Jméno	Identifikátor	Název třídy	Akce
	hp6560	hp6560-KIV	notebooky-KIV	
	DELL Vostro	DELL	notebooky-CIV	

Obrázek C.3: Vícepoložkové zobrazení objektů.

V prvních několika sloupcích jsou uvedeny některé základní údaje o listované položce (na Obrázku C.3 jsou to konkrétní objekty a jejich názvy, textové identifikátory atp.). Tyto je možné dále filtrovat podle klíče, pro jehož zadání je připraveno vstupní textové pole v hlavičce některých sloupců. Filtraci je možno kombinovat napříč všemi sloupci. Pokud se všechny položky nevejdou na jednu stránku, je možné celou tabulku stránkovat.

V posledním sloupci jsou poté dostupné akce, které může role přihlášeného uživatele provádět s položkami. Typickou akcí je zejména *zobrazení profilu* položky (objektu, třídy objektu, uživatele). Další akce vyplývají z požadavků kladených na systém a popis těchto akcí je možné získat z vyskakovací textové nápovědy zobrazené při najetí kurzoru myši na tlačítko akce.

C.1.3 Kalendář

Kalendář je nejdůležitější komponentou systému, protože umožňuje prohlížet, spravovat, vyzvedávat a vracet rezervace. Kalendářů v systému existuje celá řada (např. pro konkrétní objekt, administrátorský kalendář atp.), nicméně jeho možnosti si budeme demonstrovat na jeho nejtypičtějším užití – na kalendáři aktuálně přihlášeného uživatele (funkce *Můj kalendář*). Kalendář je zobrazen na Obrázku C.4.

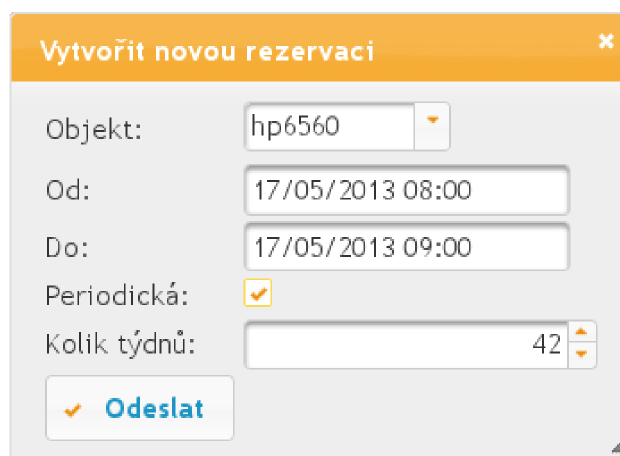
Obrázek C.4: Měsíční zobrazení kalendáře.

Kalendář umožňuje tři druhy pohledu – měsíční, týdenní a denní. Na všech těchto pohledech jsou pak zobrazeny vytvořené rezervace. Mezi zobrazenými časovými úseky je možné se posouvat pomocí posuvných tlačítek v pravém horním rohu kalendáře, nebo lze okamžitě zobrazit aktuální den. Na Obrázku C.5 je znázorněn týdenní pohled kalendáře. Pohled je posunovatelný v čase a jednotlivé události rezervací (kdy je každá na jiný objekt) se mohou překrývat.

Obrázek C.5: Týdenní zobrazení kalendáře.

Vytvoření rezervace

Rezervaci je možné vytvořit pouhým kliknutím na jakékoliv místo v kalendáři, ve kterém není žádná jiná rezervace. Následné dialogové okno umožňuje definovat časy rezervace (jsou přednastavené podle místa kliknutí do kalendáře) a periodicitu (Obr. C.6). Dále je nutné ze seznamu vybrat objekt, na který má být rezervace vytvořena. V seznamu objektů je možné vyhledávat stejným způsobem jako v tabulce objektů a pokud si uživatel není jist identitou objektu, je možné přímo ze seznamu otevřít nové okno s profilem objektu.



Obrázek C.6: Vytvoření rezervace.

Dialogové okno vytvoření objektu se může lišit v závislosti na zobrazeném kalendáři. Například role *Administrátor* musí v některých případech definovat i uživatele rezervace. Jeho výběr je implementován obdobně jako výběr objektu.

Editace rezervace

Rezervaci je možné editovat hned několika způsoby. Uživatel může rezervace přímo v kalendáři za pomoci kurzoru myši „roztahovat" nebo „smršťovat" a tím měnit jejich časové rozpětí. To je možné ve všech pohledech kalendáře. Pokud chce uživatel zachovat časové rozpětí a pouze změnit pozici rezervace v kalendáři, je možné jednotlivé rezervace „přetáhnout" mezi hodinami, dny, týdny atp. Pokud dojde ke kolizi rezervace, není tato editace uskutečněna.

Dalším způsobem editace je kliknutí na samotnou rezervaci. Následné dialogové okno umožňuje kromě samotné editace časového úseku rezervace také změnu platnosti periody a další řadu akcí včetně úplného odstranění rezervace, jejího vyzvednutí a vrácení a dále také informuje o stavu rezervace (Obr. C.7).

Tlačítka těchto akcí mohou být zašedivěna, pokud je např. rezervace nevyzvednutelná, umístěná v minulosti atp. Dostupnost akcí se řídí požadavky kladenými na systém. Také pro role *Správce rezervací* a *Administrátor* nabízí toto okno další možnosti editace, jako je například změna vlastníka rezervace.

Obrázek C.7: Editace rezervace.

C.1.4 Administrace

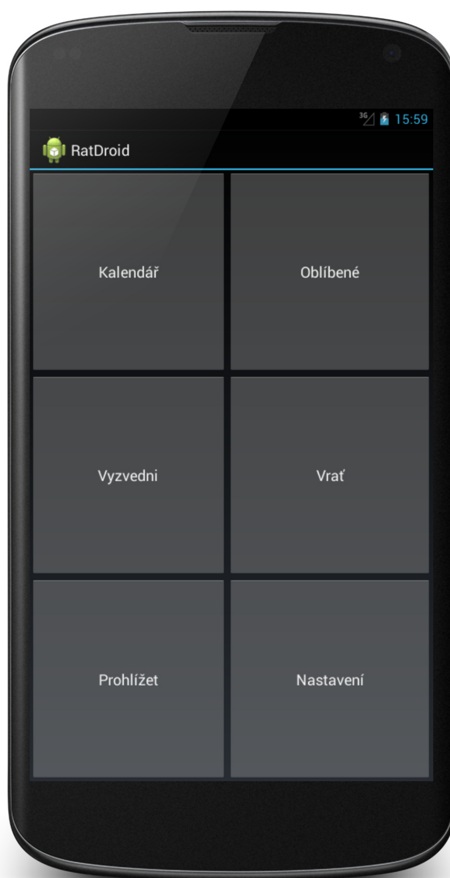
Pokud se uživatel přihlásí jako *Administrátor*, systém zpřístupní další možnosti správy systému. Zejména je možné editovat a mazat objekty, třídy objektů a uživatelské účty. Jak ukazuje Obrázek C.8, oproti běžným uživatelům také narůstá počet možných akcí. Odstranění jakkoliv položky je realizováno jako výběr jedné nebo více položek a následné kliknutí na tlačítko *Odstranit*, které je umístěno v patičce tabulky. Následně je nutné volbu ještě jednou potvrdit v dialogovém oknu.

Obrázek C.8: Vícepoložkové administrační zobrazení objektů.

Obdobně je realizováno vytváření položek. Při kliknutí na tlačítko *Nový*, umístěného v patičce tabulky, dojde k přesměrování na vstupní formulář nové položky.

C.2 Aplikace RatDroid

Mobilní aplikace RatDroid nenabízí žádné možnosti administrace systému, pouze základní správu vlastních oblíbených položek a rezervací. Na Obrázku C.9 je znázorněna hlavní obrazovka aplikace, tvořící základní navigační menu aplikace.



Obrázek C.9: Hlavní obrazovka aplikace.

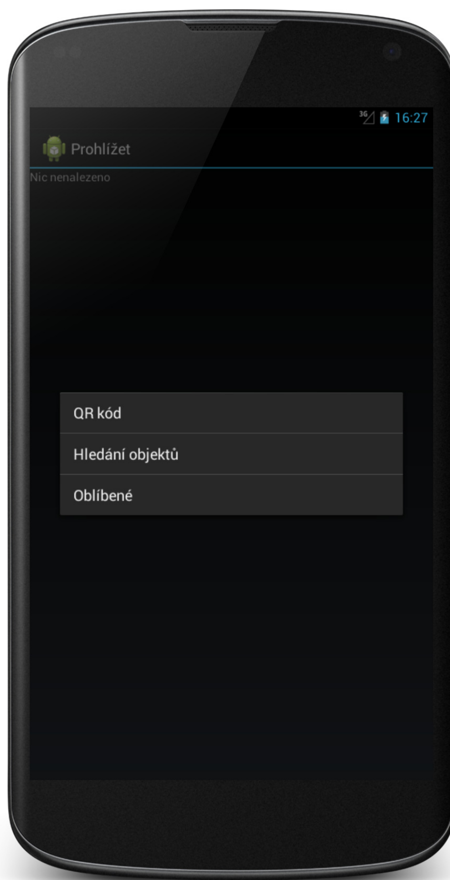
C.2.1 Nastavení

Narozdíl od webového rozhraní vyžaduje mobilní aplikace základní nastavení direktiv pro připojení k centrálnímu serveru. Pokud se připojení k serveru nezdaří, jsou vypnuty všechny navigační prvky na obrazovky, které připojení k serveru explicitně vyžadují. Jednotlivé položky nastavení sledují obvyklý scénář nastavení připojení (*URL*, *Port* atp.), dále je nutné zadat jména heslo uživatele. Je možné

kdykoliv za běhu aplikace změnit jméno a heslo uživatele. Aplikace změnu uživatelského jména rozpozná a přepne kontext uživatele (tj. kalendáře, oblíbené položky atp.).

C.2.2 Funkce „Prohlížet"

Aplikace umožňuje prohledávat seznam existujících objektů. Je možné buď použít čtečku QR kódů, výběr z oblíbených objektů přihlášeného uživatele nebo použít vyhledávání. Obrazovka na Obrázku C.10 ukazuje situaci po otevření obrazovky s funkcí *Prohlížet*.



Obrázek C.10: Prohlížení objektů.

Výběr z dialogového menu reprezentuje akci výběru objektů, které se po dokončení akce zobrazí v seznamu ležícím pod tímto dialogovým menu. Menu je kdykoliv možné vyvolat znovu stisknutím (hardwarového nebo emulovaného) tlačítka *Menu*,

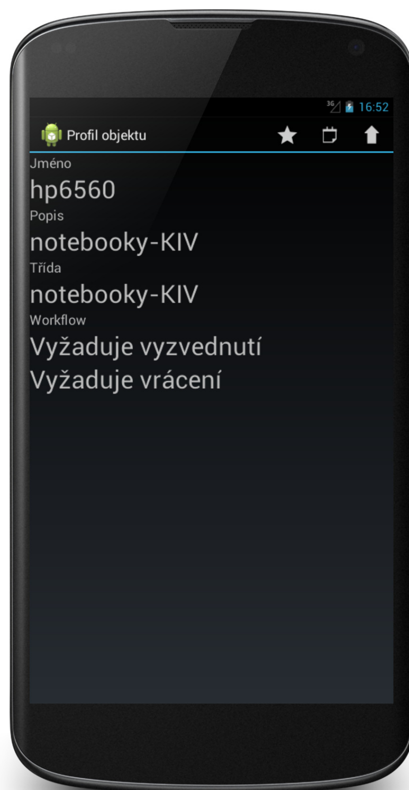
a tím změnit aktuální selekci objektů. Výběrem objektu ze seznamu dojde k vyvolání obrazovky s profilem objektu (viz dále).

C.2.3 Funkce „Oblíbené“

Aplikace umožňuje zobrazení všech oblíbených objektů přímo z hlavního menu aplikace. Při dotyku na položku ze seznamu objektů dojde k zobrazení profilu tohoto objektu obdobně jako v případě obrazovky *Prohlížet*.

C.2.4 Profil objektu

Profil objektu obsahuje všechny důležité informace o objektu. Současně v pravém horním rohu obsahuje sadu akcí, které je možné nad objektem provést – přidat nebo odebrat jej ze seznamu oblíbených položek, zobrazit jeho kalendář nebo jej vyzvednout (viz dále).



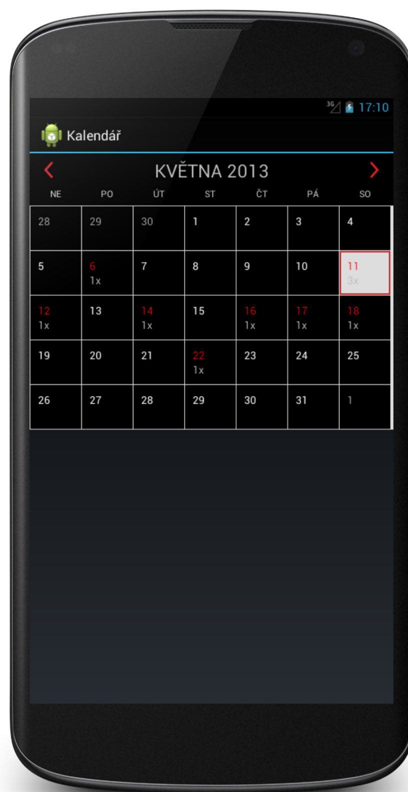
Obrázek C.11: Profil objektu.

C.2.5 Funkce „Vrať"

Aplikace umožňuje vracení objektů podle pravidel uvedených ve specifikaci požadavků. Narozdíl od webového rozhraní tak ale nedělá prostřednictvím kalendáře, ale za pomoci seznamu všech objektů, které si daný uživatel v minulosti vypůjčil a nepotvrdil jejich vrácení (pokud to objekt vyžaduje). Pouhým dotykem položky ze seznamu dojde k jejímu vrácení a návratu na hlavní obrazovku.

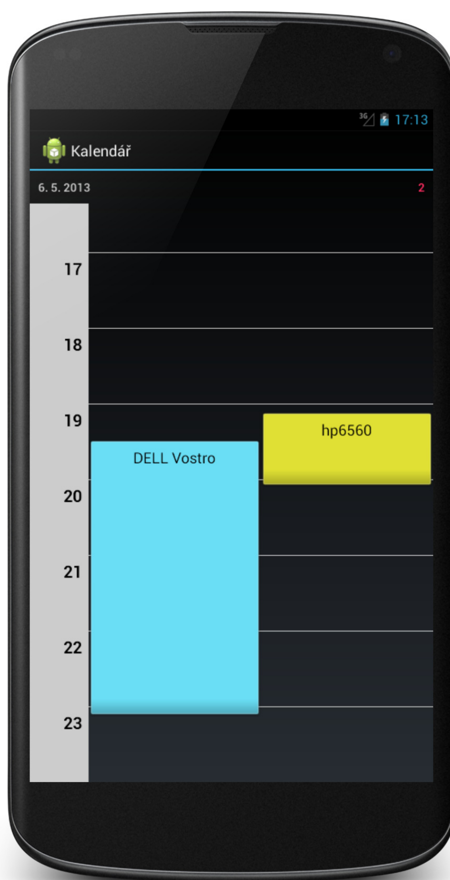
C.2.6 Kalendář

V mobilní aplikaci existují dva kalendáře – vázaný na přihlášeného uživatele a vázaný na objekt. Uživatelský kalendář je možné zobrazit z hlavního menu navigačním tlačítkem *Můj kalendář*, kalendář objektu pak lze zobrazit z profilu tohoto objektu akčním tlačítkem. Kalendář se skládá z měsíčního (Obr. C.12) a denního pohledu.



Obrázek C.12: Měsíční pohled kalendáře.

Měsíční pohled zobrazuje červeně všechny dny, ve kterých je vytvořena nějaká rezervace. Znázorňuje také počet těchto rezervací v jednom dni. Navigace mezi jednotlivými měsíci je možná pomocí bočních navigačních šipek. Při dotyku některého dne z kalendáře dojde k vyvolání denního pohledu kalendáře (Obr. C.13).



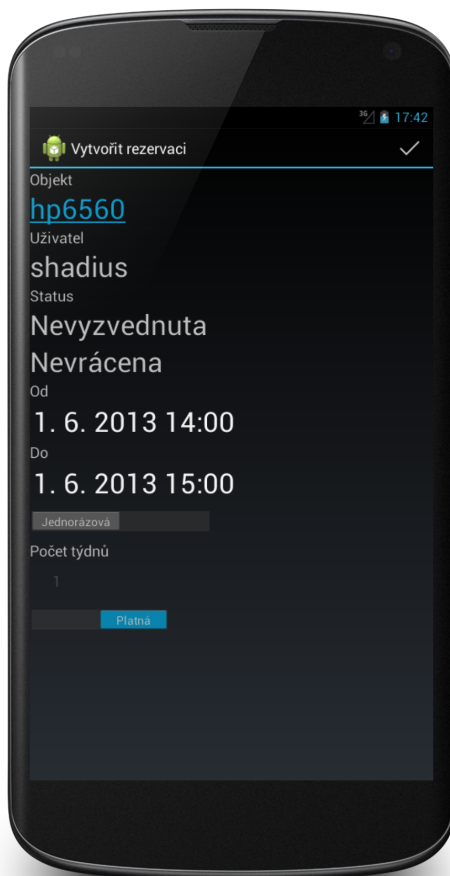
Obrázek C.13: Denní pohled kalendáře.

Denní pohled vypadá přibližně jako kalendář webového rozhraní a také se s ním obdobně pracuje. Je možné s ním dotykovými gesty libovolně vertikálně posouvat. Při dotyku na místo, kde není žádná rezervace, dojde k vyvolání obrazovky určené k vytvoření rezervace (pokud se uživatel nesnaží vytvořit rezervaci „v minulosti“). Při dotyku samotné rezervace se poté zobrazí editační obrazovka.

C.2.7 Vytváření a editace rezervace

Obrazovky vytváření a editace rezervace jsou velmi podobné dialogům na centrálním serveru. Jediným rozdílem je snaha maximálně využít prostředky doty-

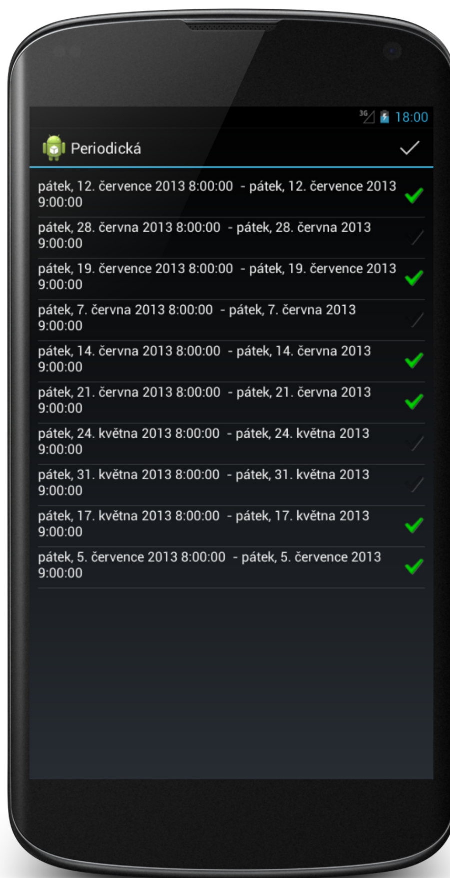
kových displejů mobilních zařízení (Obr. C.14). Pro vytvoření rezervace je vždy nejprve nutné definovat rezervovaný objekt – z pozice kalendáře objektu je již objekt vlastně identifikován, nicméně v kalendáři uživatele se před samotnou obrazovkou určenou k vytvoření objektu zobrazí komponenta podobná dříve zmíněné obrazovce *Prohlížet*, ve které uživatel musí rezervovaný objekt vybrat.



Obrázek C.14: Denní pohled kalendáře.

Obrazovka pro editaci rezervací přináší oproti webovému rozhraní jednu možnost navíc – je možné zobrazit všechny části periodické rezervace v jedné obrazovce a přímo nastavovat jejich platnost (Obr. C.15). Je to kompromis vůči menším displejům mobilních zařízení, kdy není zcela jednoduché se zorientovat v rozsáhlém kalendáři. Navigační tlačítko pro zobrazení této obrazovky je v pravém horním rohu editační obrazovky spolu s tlačítkem pro odstranění rezervace.

Jinak je obrazovka editace rezervace prakticky totožná s editačním dialogovým oknem ve webovém formuláři, pouze nenabízí možnosti vyzvednutí a vrácení. Ty byly v mobilní aplikaci implementovány jiným způsobem.



Obrázek C.15: Editace platnosti periodických rezervací.

C.2.8 Funkce „Vyzvedni“

Tato funkce je dostupná přímo z hlavní obrazovky aplikace a funguje odlišně než je tomu u webového rozhraní. Jejím cílem je půjčování a vyzvedávání objektů co nejvíce urychlit. Nejprve se zobrazí komponenta podobná dříve zmíněné obrazovce *Prohlížet*, ve které musí uživatel určit vyzvedávaný objekt (pokud uživatel vyzvedává objekt přímo z profilu objektu, tato komponenta se nezobrazí). Po vybrání objektu dojde k automatického vyhledání nejbližší vyzvednutelné rezervace učiněné na tento objekt. Pokud taková rezervace existuje, je automaticky vyzvednuta a po uživateli není vyžadována žádná další akce. V opačném případě je uživatel

přesměrován na obrazovku určenou pro vytvoření rezervace, kde si může zvolit čas konce nově vzniknuvší rezervace. Takové rezervace nelze vytvářet jako periodické a jsou automaticky vedeny jako platné a vyzvednuté. Funkce *Vyzvedni* neumožňuje vytvářet rezervace na objekty, které nevyžadují potvrzení vyzvednutí.

D Administrátorská dokumentace

V této příloze bude uveden stručný manuál k nasazení všech částí informačního systému RAT.

D.1 Nasazení webové aplikace

Aplikace je distribuována jako *WAR* (*Web application ARchive*) soubor. Pro běh je vyžadován webový kontejner Tomcat ve verzi 7. Jiné kontejnery nebyly testovány a jejich použití je tedy bez záruky. Dále je potřeba nainstalovat Java běhové prostředí v minimální verzi 7. Pro kompilaci a sestavení *WAR* souboru přímo ze zdrojových kódů je dostupný *Maven* skript. Všechny závislosti spravuje také *Maven*.

Před samotným spuštěním kontejneru je však nutné nakonfigurovat zejména přístup do databáze, zabezpečení přenosu a další nastavení.

D.1.1 Přístup do databáze

Nejprve je nutné nakonfigurovat datovou vrstvu. V aplikaci není integrována žádná relační databáze a je tedy na bedrech administrátora její nasazení a patřičné nastavení aplikace. Konfigurační direktivy datové vrstvy aplikace jsou umístěny v souboru `WEB-INF/classes/persistence.xml`. Doporučená relační databáze pro běh aplikace je *H2 Database Engine*. Aplikace také již obsahuje patřičné knihovny potřebné pro připojení k této databázi. Je však možné použít libovolnou implementaci, jejíž dialekt framework Hibernate podporuje.

Pro vytvoření struktury databáze je možné využít mapování frameworku Hibernate. Před prvním spuštěním je nutné v konfiguračním souboru „odkomentovat“ následující řádek:

```
<prop key="hibernate.hbm2ddl.auto">update</prop>
```

Hibernate poté sám vytvoří relační strukturu včetně všech vazeb (po vytvoření struktury databáze ve však nutné řádek znovu zakomentovat). Následně je nutné se libovolným konektorem připojit do databázového systému a spustit následující SQL dotaz (syntaxe se může lišit v závisloti na použitém databázovém systému):

```
INSERT INTO "PUBLIC"."USER" ("USERID", "FIRSTNAME", "SURNAME",
"EMAIL", "NICKNAME","PASSHASH", "ROLE", "IMPLEMENTOR") VALUES(0,
'admin','admin', '', 'admin','21232f297a57a5a743894a0e4a801fc3',
'ADMIN', 'DEFAULT')
```

Spuštěním dotazu bude vytvořen administrátorský účet *admin* s heslem *admin*. Tento účet je pak možné použít pro vytvoření dalších uživatelských nebo administrátorských účtů.

Indexování databáze

Některé databázové entity jsou indexovány enginem Hibernate Search. Protože jsou tyto indexy uchovávány mimo databázi, je nutné definovat cestu k jejich uložení. To je možné ve výše zmíněném konfiguračním souboru v rámci následující direktivy:

```
<prop key="hibernate.search.default.indexBase">cesta_k_indexum</prop>
```

D.1.2 Konfigurace REST rozhraní

Pokud dojde při běhu webové REST služby k chybě, aplikace vygeneruje o této události zprávu a přiloží ji k HTTP odpovědi. Do této zprávy je možné vkládat některá užitečná data pro ladění aplikace. Jedním z nich je i e-mailová adresa (nebo URL *bug-tracking* systému), kam může klientská strana hlásit chybu serveru. Tato direktiva se nachází v konfiguračním souboru `WEB-INF/RAT-servlet.xml` a vypadá následovně:

```
<property name="defaultMoreInfoUrl" value="mailto:email"></property>
```

D.1.3 Nastavení systému rezervací

V souboru `WEB-INF/classes/rat.properties` jsou základní konfigurační direktivy rezervací. Je žádoucí provést konfiguraci těchto direktiv hned při nasazení systému. Následné změny této konfigurace nejsou podporovány zejména proto, protože nastavení systému si všichni mobilní klienti stáhnou ze serveru při prvním svém spuštění. Následující výpis současně ukazuje přednastavené hodnoty, které jsou použity v okamžiku, kdy se nezdaří parsování konfiguračního souboru (tato událost je zalogována):

```
reservation.duration.minimum=60 (minuty) - jedna hodina
reservation.duration.maximum=10080 (minuty) - jeden týden
reservation.periodical.maximum=52 (týdny)
reservation.expiration.limit=10 (minuty)
```

D.1.4 Nastavení jazyka aplikace

Pro webové rozhraní je podporována čestina a angličtina. Změna jazyka za běhu aplikace není možná. Mezi jazyky lze programově přepínat v konfiguračním souboru `WEB-INF/faces-config.xml`:

```
<locale-config>
    <default-locale>cs</default-locale>
    <supported-locale>en</supported-locale>
</locale-config>
```

D.1.5 Konfigurace SSL certifikátu

Systém vyžaduje speciální konfiguraci SSL certifikátu zejména proto, že Android velmi obtížně pracuje s certifikáty, které nebyly ověřeny žádnou certifikační autoritou. Pro vytvoření certifikátu (který bude uložený v tzv. *keystore*) je nutné použít knihovnu *Bouncy Castle*:

1. Nejprve je nutné stáhnout *Bouncy Castle* knihovnu, například z oficiálního Maven repozitáře (`bcprov-ext-jdkxx-1.xx.jar`).

2. Dále se tato knihovna musí nakopírovat do složek `JAVA_HOME/jre/lib/ext` a (pokud existuje) `JAVA_HOME/lib/ext`.

3. Do všech konfiguračních souborů `java.security` v `JAVA_HOME` je nutné přidat následující řádku:

   ```
   security.provider.7=org.bouncycastle.jce.provider.BouncyCastle
   Provider
   ```

4. Nyní je možné vygenerovat *keystore* s certifikátem, a to následovně:

   ```
   keytool -genkey -alias RAT -keystore RAT.keystore
   -storepass heslo -storetype BKS -provider
   org.bouncycastle.jce.provider.BouncyCastleProvider
   ```

Generátor se bude ptát na některé další údaje, důležitý je však pouze tzv. `CN`, neboli obecné jméno. Zde je nutné zadat doménu, na které bude server s aplikací spuštěn – bez tohoto kroku se nelze k serveru připojit z mobilní aplikace. Je také nutné si poznamenat přístupově heslo k vygenerovanému `RAT.keystore`.

Nyní je nutné konfigurovat Tomcat spolu s *Bouncy Castle* a vygenerovaným *keystore*, a to vložením následující direktivy do konfiguračního souboru `server.xml`, který je umístěn v Tomcat distribuci:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
maxThreads="150" scheme="https" secure="true" clientAuth="false"
sslProtocol="TLS" keystoreFile="RAT.keystore"
keystorePass="heslo" keystoreType="BKS" alias="RAT" />
```

Tím však konfigurace nekončí. Vygenerovaný *keystore* je nutné také zakomponovat do mobilní aplikace *Ratdroid* (viz dále).

D.1.6 Nastavení *Cookies*

Posledním krokem je nastavení *Cookies*. Těmi je mimo jiné přenášeno `sessionId` a je nutné nastavit jejich `Domain-Name`. Bez této konfigurace nefunguje mobilní aplikace *RatSharp*, protože Windows Phone *Cookies* bez doménového jména nepodporuje. Nastavení je v konfiguračním souboru `META-INF/context.xml`:

```
<Context sessionCookiePath="/RAT" sessionCookieDomain="domena" />
```

D.2 Nasazení mobilní aplikace *RatDroid*

Aplikace je určena pro Android 4.1.2 a vyšší. Mobilní aplikaci je bohužel vzhledem k integraci *keystore* nutné vždy sestavit ze zdrojového kódu. Pro sestavení je opět možné použit *Maven*. Aplikace využívá knihovny *Caldroid*, která je umístěna na přiloženém médiu. Všechny ostatní závislosti spravuje *Maven*.

D.2.1 Integrace ***keystore***

Projekt aplikace je nutné otevřít v libovolném vývojovém prostředí, ve kterém je možné vyvíjet aplikace pro mobilní platformu Android. Následně je třeba do složky `res/raw` nakopírovat pod názvem `rat_keystore` v předchozím kroku vytvořený *Bouncy Castle keystore*. Ve třídě `MyHttpClient` je poté statická konstanta `RAT_KEYSTORE_PASSWORD`, kterou je nutné inicializovat na heslo pro přístup do *keystore*.

Následně je možné aplikaci sestavit a distribuovat ji jako *APK* balíček.

E Obsah přiloženého média

Součástí této práce je rovněž i pamět'ové médium (DVD) obsahující tyto adresáře a soubory:

- `readme.txt` – textový soubor obsahující popis a význam jednotlivých adresářů na médiu.
- `/bin` – obsahuje *WAR* soubor s aplikací centrálního serveru.
- `/doc` – obsahuje PDF verzi této práce.
- `/src` – obsahuje zdrojové kódy mobilní aplikace *RatDroid* a webové aplikace *RatServer*.
- `/rest` – obsahuje XSD schémata XML reprezentace dat, které jsou přenášeny v rámci REST komunikace.

Literatura

[1] GALUSHA, C. Getting started with IT asset management. *IT Professional* [online]. vol. 3, issue 3, s. 37-40 [cit. 2013-05-10]. DOI: 10.1109/6294.939973. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=939973

[2] MANOHARAN, Sathiamoorthy. On GPS Tracking of Mobile Devices. *2009 Fifth International Conference on Networking and Services* [online]. IEEE, 2009, s. 415-418 [cit. 2013-04-17]. DOI: 10.1109/ICNS.2009.103. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4976795

[3] SPIRITO, M.A. On the accuracy of cellular mobile station location estimation. *IEEE Transactions on Vehicular Technology* [online]. roč. 50, č. 3, s. 674-685 [cit. 2013-04-17]. ISSN 0018-9545. DOI: 10.1109/25.933304. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=933304

[4] OLIVER, R. Why the BYOD boom is changing how we think about business it. *Engineering & Technology*. 2012, roč. 7, č. 10, s. 28. Dostupné z: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6413056&isnumber=6355539

[5] INGKHWAN, A. WI-FI Tracker: an Organization WI-FI Tracking System. *2006 Canadian Conference on Electrical and Computer Engineering* [online]. IEEE, 2006, s. 231-234 [cit. 2013-04-17]. DOI: 10.1109/CCECE.2006.277387. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4054833

[6] YEUNG, Wilson M. a Joseph K. NG. Wireless LAN Positioning based on Received Signal Strength from Mobile device and Access Points. *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*. IEEE, 2007, roč. 7, č. 10, s. 131-137. ISSN 1533-2306. DOI: 10.1109/RTCSA.2007.73. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4296845

[7] AGRAWAL, Rohit a Ashesh VASALYA. *Bluetooth Navigation System using Wi-Fi Access Points*. 2012, 8 s. Dostupné z: http://arxiv.org/abs/1204.1748

[8] GOZICK, Brandon, Kalyan Pathapati SUBBU, Ram DANTU a Tomyo MAESHIRO. Magnetic Maps for Indoor Navigation. *IEEE Transactions on Instrumentation and Measurement* [online]. roč. 60, č. 12, s. 3883-3891 [cit. 2013-04-18]. ISSN 0018-9456. DOI: 10.1109/TIM.2011.2147690. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5773083

[9] RIEHLE, T. H., S. M. ANDERSON, P. A. LICHTER, J. P. CONDON, S. I. SHEIKH a D. S. HEDIN. Indoor waypoint navigation via magnetic anomalies. *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society* [online]. IEEE, 2011, s. 5315-5318 [cit. 2013-04-18]. DOI: 10.1109/IEMBS.2011.6091315. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6091315

[10] INDOORATLAS. *IndoorAtlas* [online]. 2013 [cit. 2013-04-18]. Dostupné z: http://www.indooratlas.com

[11] CHUNG, Jaewoo, Matt DONAHOE, Chris SCHMANDT, Ig-Jae KIM, Pedram RAZAVAI a Micaela WISEMAN. Indoor location sensing using geomagnetism. *Proceedings of the 9th international conference on Mobile systems, applications, and services - MobiSys '11* [online]. New York, New York, USA: ACM Press, 2011, s. 141- [cit. 2013-04-18]. DOI: 10.1145/1999995.2000010. Dostupné z: http://portal.acm.org/citation.cfm?doid=1999995.2000010

[12] OHBUCHI, E., H. HANAIZUMI a LIM AH HOCK. Barcode Readers using the Camera Device in Mobile Phones. *2004 International Conference on Cyberworlds* [online]. IEEE, 2004, s. 260-265 [cit. 2013-04-18]. DOI: 10.1109/CW.2004.23. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1366183

[13] WANT, R. An Introduction to RFID Technology. IEEE Pervasive Computing [online]. roč. 5, č. 1, s. 25-33 [cit. 2013-04-18]. ISSN 1536-1268. DOI: 10.1109/MPRV.2006.2. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1593568

[14] NFC Phones list, all available handsets: Complete list with all mobile phones with nfc chip. List of NFC Phones, availability by country and by carrier [online]. 2013 [cit. 2013-04-18]. Dostupné z: http://www.nfc-phones.org/summary-list-of-all-available-nfc-phones/

[15] ABLESON, W. *Android in action.* 3rd ed. Greenwich, Conn.: Manning, c2012, xxviii, 632 p. ISBN 16-172-9050-5.

[16] HTML 5.1 Nightly: A vocabulary and associated APIs for HTML and XHTML. *World Wide Web Consortium (W3C)* [online]. 2013, 29.4.2013 [cit. 2013-04-29]. Dostupné z: http://www.w3.org/html/wg/drafts/html/master

[17] Introduction to Mobile Development: Developing Mobile Applications using Xamarin. *Xamarin: Build cross-platform iOS, Android, Mac and Windows apps with C# and .NET* [online]. 2013 [cit. 2013-04-29]. Dostupné z: http://docs.xamarin.com/guides/cross-platform/getting_started/introduction_to_mobile_development

[18] DAY, J.D. a H. ZIMMERMANN. The OSI reference model. *Proceedings of the IEEE* [online]. roč. 71, č. 12, s. 1334-1340 [cit. 2013-04-30]. ISSN 0018-9219. DOI: 10.1109/PROC.1983.12775. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1457043

[19] LIU, Ling a M ÖZSU. *Encyclopedia of database systems.* New York: Springer, c2009, 5 v. (lxix, 3749 p.). ISBN 978-038-7496-160.

[20] FOWLER, Martin. *Patterns of enterprise application architecture.* Boston: Addison-Wesley, c2003, xxiv, 533 p. ISBN 03-211-2742-0.

[21] ELLIOTT, James, Ryan FOWLER a Tim O'BRIEN. *Harnessing Hibernate.* Sebastopol: O'Reilly, 2008, xiv, 363 s. ISBN 978-0-596-51772-4.

[22] JOEL MURACH, Andrea Steelman, Ryan FOWLER a Tim O'BRIEN. *Murach's Java servlets and JSP: training.* 2nd ed. Fresno, CA: Mike Murach, 2008, xiv, 363 s. ISBN 18-907-7444-8.

[23] BERGSTEN, Hans, Ryan FOWLER a Tim O'BRIEN. *JavaServer faces: training.* 2nd ed. Sebastopol, CA: O'Reilly, c2004, xiv, 589 p. ISBN 05-960-0539-3.

[24] HEWITT, Eben, Ryan FOWLER a Tim O'BRIEN. *Java SOA cookbook: training.* 1st ed. Sebastopol, Calif.: O'Reilly, c2009, xxiv, 714 p. ISBN 05-965-2072-7.

[25] SOAP Specifications. *World Wide Web Consortium (W3C)* [online]. 2013 [cit. 2013-05-07]. Dostupné z: http://www.w3.org/TR/soap

[26] HTTP/1.1: Method Definitions. *World Wide Web Consortium (W3C)* [online]. 2013 [cit. 2013-05-05]. Dostupné z: http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

[27] WALLS, Craig. Spring in action. 3rd ed. Shelter Island: Manning, c2011, xxiii, 400 p. ISBN 19-351-8235-8.