Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Implementace webové aplikace s využitím webových služeb

Tomáš Holický

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Tomáš Holický

Informatika

Název práce

Implementace webové aplikace s využitím webových služeb

Název anglicky

Implementation of a web application utilizing web services

Cíle práce

Diplomová práce je tématicky zaměřena na implementaci webové aplikace s podporou webových služeb. Hlavním cílem práce je analyzovat, navrhnout a následně implementovat webovou aplikaci využívající webové služby. Dílčím cílem diplomové práce je charakterizovat servisně orientovanou architekturu.

Metodika

Metodika řešené problematiky diplomové práce je založena na studiu a analýze odborných informačních zdrojů. Vlastní řešení je realizováno formou analýzy, návrhu a následné implementace webové aplikace s využitím zvolených technologií. Na základě syntézy teoretických poznatků a výsledků řešení budou formulovány závěry diplomové práce.

Doporučený rozsah práce

60-80 stran

Klíčová slova

Servisně orientovaná architektura (SOA), Webová služba, Integrace, Java, SOAP, REST

Doporučené zdroje informací

BURKE, Bill. RESTful Java with JAX-RS 2.0. Second edition. xxi, 367 pages. ISBN 144936134x.

DAIGNEAU, Robert. Service design patterns: fundamental design solutions for SOAP/WSDL and RESTful Web services. Upper Saddle River, NJ: Addison-Wesley, 2012, xxv, 321 p. Addison-Wesley signature series. ISBN 032154420x.

ERL, T. *SOA : servisně orientovaná architektura : kompletní průvodce.* Brno: Computer Press, 2009. ISBN 978-80-251-1886-3.

FOWLER, Martin. Patterns of enterprise application architecture. Boston: Addison-Wesley, 2003, xxiv, 533 s. Addison-Wesley signature series. ISBN 0321127420.

KALIN, Martin. Java Web services: up and running. Second edition. Sebastopol, Calif: O'Reilly, 2013. ISBN 9781449365110.

Spring recipes: a problem-solution approach. Heidelberg: Springer, 2014. ISBN 1430259086.

Předběžný termín obhajoby

2016/17 ZS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 20. 2. 2016

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 20. 2. 2016

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 28. 06. 2016

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Implementace webové aplikace s využitím webových služeb" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 24.11.2016 _____

Poděkování

Rád bych touto cestou poděkoval vedoucímu diplomové práce panu Ing. Jiřímu Brožkovi, Ph.D. za jeho rady, připomínky a odbornou pomoc.

Implementace webové aplikace s využitím webových služeb

Souhrn

Tato diplomová práce se zabývá problematikou vývoje webové aplikace s využitím webových služeb. V teoretické části popisuje podstatu servisně orientované architektury a přibližuje možnosti integrace pomocí webových služeb. V praktické části se práce zabývá analýzou, návrhem a následnou implementací webové aplikace. Konkrétně se jedná o implementaci aplikace na správu úkolů, která je rozdělena do dvou částí. Na serverovou část aplikace, která vystavuje RESTové služby a na klientskou část, jenž je konzumuje.

Klíčová slova: Servisně orientovaná architektura, Webová služba, Integrace, Java, SOAP, REST, JavaScript, Framework AngularJS, Framework Spring, Jednostránková webová aplikace

Implementation of a web application utilizing web services

Summary

This master's thesis deals with difficulties of web application development utilizing web services. The theoretical part describes the essence of service oriented architecture and explains the options of integration using web services. The practical part deals with analysis, design and subsequent implementation of web application. Specifically, the implementation of application for task manager, which is divided into two parts. The server side of application exposes REST services and client side that consumes.

Keywords: Service oriented architecture, Web service, Integration, Java, SOAP, REST, JavaScript, Framework AngularJS, Framework Spring, Single Page Application

Obsah

1 Úvod

Potřeba sdílení informací mezi lidmi, systémy a procesy neustále roste a to má dopad i na vývoj aplikací. V dnešní době je brán především důraz na aplikace, které jsou flexibilní, rychle reagují na změny, spolupracují se svým okolím a to navíc napříč všemi možnými platformami. Všechny tyto aspekty vedly k tomu, že webové služby jsou v dnešní době základním stavebním kamenem pro distribuované aplikace. V současné době platí, že téměř každá aplikace potřebuje komunikovat s ostatními aplikacemi za účelem výměny dat. K tomu se v dnešní době nejvíce využívají webové služby. Webová služba je vlastně aplikační komponenta, která je přístupná pomocí standardního otevřeného protokolu.

Základní pravidla, jak navrhovat distribuované systémy, definuje koncept servisně orientované architektury. Jedním z možných prostředků, jak principy SOA realizovat, jsou právě webové služby. Obsahem práce je tedy popis dostupných architektur webových služeb a jejich následné použití v praktické části práce, která se zaměřuje na jejich reálné použití v podobě implementované webové aplikace.

Praktická část práce se tedy především zabývá analýzou, návrhem a následnou implementací webové aplikace s použitím již zmíněných webových služeb. Konkrétně se jedná o aplikaci pro správu úkolů. Webové služby ve výsledné implementaci se využívají ke komunikaci mezi serverovou a klientskou částí aplikace.

2 Cíl práce a metodika

2.1 Cíl práce

Prvotním cílem diplomové práce je přiblížit servisně orientovanou architekturu, představit její druhy a možnosti jejího využití. Dále představit dostupné architektury webových služeb. Největší pozornost bude věnována především architektonickému stylu REST, jenž bude použit v praktické části práce.

Hlavním cílem diplomové práce je definovat požadavky, analyzovat, navrhnout a následně implementovat webovou aplikaci, která bude využívat webové služby.

2.2 Metodika práce

Pro dosažení cílů práce se bude vycházet z teoretických informací, které budou čerpány z dostupných informačních zdrojů doplněných praktickými znalostmi v tomto oboru. Na základě syntézy výše zmíněných zdrojů bude popsán koncept servisně orientované architektury a dále budou přiblíženy typy webových služeb.

Celý proces vývoje webové aplikace bude vycházet z metodiky Unified Process (Unifikovaná metodika vývoje softwaru). V první řadě bude nutné definovat základní požadavky na aplikaci, tzv. případy užití. Na základě těchto požadavků bude provedena analýza proveditelnosti a návrh webové aplikace. Navržené řešení webové aplikace bude obsahovat detailní popis jednotlivých funkčností aplikace s pomocí různých typů diagramů či wireframů. Na základě analýzy a návrhu bude provedena implementace webové aplikace. V první řade se bude implementovat serverová část aplikace, která bude vystavovat webové služby. Poté se bude implementovat klientská aplikace v podobě Single Page aplikace, která dané služby bude konzumovat.

3 Teoretická východiska

Teoretická část práce objasňuje pojem SOA a představuje dostupné architektury webových služeb, z nichž se servisně orientovaná architektura skládá. Hlavní zaměření je především na REST služby a to z důvodu, že jsou využity v praktické části práce, tedy ve výsledné implementaci aplikace.

3.1 Servisně orientovaná architektura

SOA je forma technologické architektury, která podporuje servisní orientaci za použití webových služeb. Servisní orientace jako prostředek pro oddělování věcí do logických a nezávislých jednotek je v dnešní době velmi rozšířenou koncepcí. Současná SOA představuje otevřenou, rozšiřitelnou, federační a komponovatelnou architekturu, která podporuje servisní orientaci a skládá se ze služeb, které jsou autonomní, jednoduše zjistitelné, schopné zajistit kvalitu služeb, vzájemně spolupracovat a jsou implementované jako webové služby. SOA je tedy koncepce IT architektury, kde každá aplikace může nabízet své služby, které poté ostatní aplikace mohou konzumovat pomocí standardizovaného protokolu a předem dohodnutého rozhraní. Webové služby jsou v kontextu servisně orientované architektury jen černé skříňky, kde jejich vnitřní struktura je nezávislá na druhu a povaze okolí, kterým jsou využívány. Tímto způsobem se návrh aplikací výrazně zjednodušuje a zajišťuje jednoduchost v inkrementálním vývoji a budoucím rozšiřování. S tím je spojená i úspora nákladů, která nastane při budoucích změnách aplikační architektury a také větší konkurenceschopnosti v podobě rychlejší reakce na podněty trhu, než má konkurence. Aplikace, které jsou vyvinuté v konceptu SOA architektury, jsou jednodušeji integrovány s různými externími aplikacemi, než dříve používané monolitické aplikace. [1] [2]

3.2 Webová služba

Webová služba představuje významný posun od velkých monolitických aplikací k modelu založenému na komponentech. Každá webová služba je tedy samostatná komponenta, která na základě přesně definovaného rozhraní poskytuje určitou funkcionalitu a je identifikována prostřednictvím URI. Pod názvem webová služba se skrývá skupina technologií a metod, které spojují aplikace bez ohledu na použité platformě a umožňuje efektivní komunikaci mezi aplikacemi. Webová služba může vystupovat v různých rolích v závislosti v jakém kontextu se nachází. Například může být iniciátor, přenašeč nebo příjemce nějaké zprávy, není tedy možno jednoznačně určit, jestli to je klient nebo server. Obecně lze říci, že je to jednotka softwaru, která je schopná změnit svou roli vzhledem ke své zodpovědnosti za zpracování v dané situaci. Webová služba může vystupovat jako poskytovatel služeb, pokud je volána externím zdrojem a poskytuje popis služby, kterým nabízí informace o svých funkcích a o svém chování. Webová služba je v roli spotřebitele služeb za podmínek, že webová služba volá poskytovatele služeb odesláním zprávy. Spotřebitel služeb je obdobně jako klient v klasické architektuře klient-server. [1] [3]

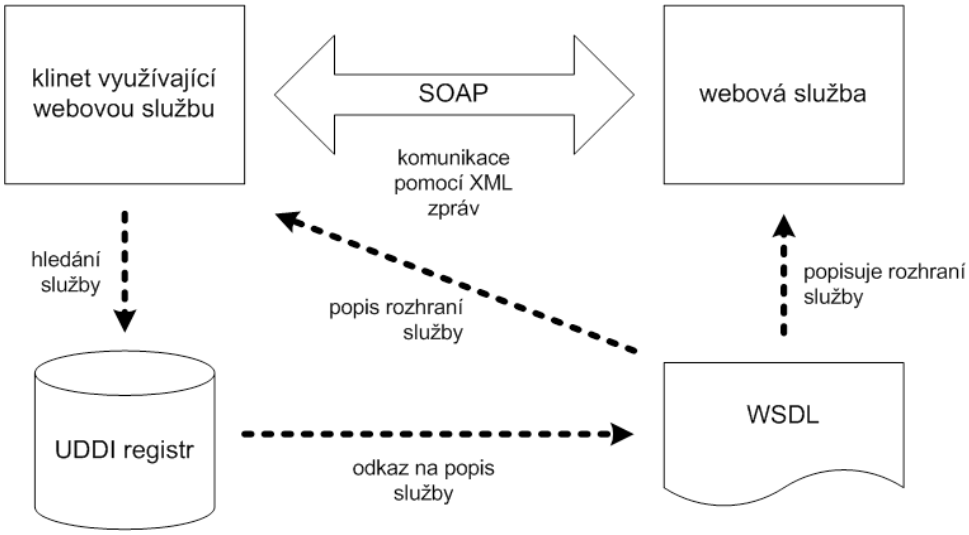
3.3 Architektury webových služeb

3.3.1 RPC

Technologická platforma používající se k tomu, aby komponenty na různých fyzických serverech spolu komunikovaly. Technologie RPC spoléhá na použití proxy serveru, který představuje vzdálenou komponentu poskytnutím lokální kopie jejího technického rozhraní, které má za úkol zahájit její vyvolání. Z pohledu architektury SOA mají aplikace založené na technologii RPC sklon soustředit se na výměnu členěných parametrů dat. [1]

3.3.2 SOAP

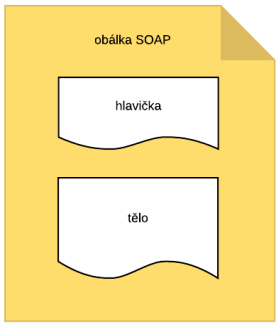
Veškerá komunikace mezi webovými službami je založena na zprávách, tím pádem je důležité aby, vybraný systém výměny zpráv byl standardizovaný a aby všechny služby bez ohledu na původ, používaly stejný formát a protokol přenosu. Tím standardem je SOAP, který se stal obecně přijímaným standardním protokolem přenosu pro zprávy zpracovávané webovými službami. Pro každou webovou službu by měl být k dispozici její popis pomocí WSDL dokumentu, z kterého je poté možno vygenerovat SOAP požadavek. Ve větších systémech se popis služby většinou zaregistruje do UDDI registru, který slouží jako seznam všech webových služeb, které je možné provolat. UDDI registr zjednodušuje vyhledávání webových služeb. Vzájemná komunikace mezi jednotlivými technologiemi je znázorněna na obrázku 1 a jejich popis je v následujících podkapitolách. SOAP nedefinuje jaký typ protokolu pro přenos SOAP zprávy má být použit, ale nejvíce se využívá protokol HTTP kvůli jeho rozšířenosti. V případě přenosu SOAP zprávy pomocí HTTP protokolu se daná SOAP zpráva nachází v těle HTTP požadavku nebo odpovědi. HTTP hlavička pak musí obsahovat SOAPAction pomocí čehož se HTTP požadavek identifikuje jako SOAP zpráva. [1] [3] [4]



Obrázek 1 – Vztah mezi technologiemi webových služeb [5]

3.3.2.1 Struktura SOAP

Hlavním cílem specifikace SOAP je tedy definovat standardní formát zprávy. Struktura tohoto formátu je poměrně jednoduchá a je zobrazena na obrázku 2. Každá SOAP zpráva je zabalena do obalu, který je známý jako tzv. obálka. Obálka zodpovídá za uchování všech částí zprávy. Každá zpráva může obsahovat hlavičku, což je oblast určená pro uchování metadat. Ačkoli hlavička není povinná, v mnoha případech servisně orientovaných řešení je zásadní částí celé architektury, neboť souvisí s používáním bloků hlavičky, pomocí kterých je možno implementovat řadu rozšíření. Bloky hlaviček SOAP zprávy mohou obsahovat velké množství pomocných informací, které souvisí s doručením a zpracováním obsahů zprávy. To výrazně odlehčuje službám, protože nemusí uchovávat logiku zpracování zpráv a navíc to posiluje současné SOA související s opětovným použitím, spoluprací a komponovatelností. [1] [3] [4]



Obrázek 2 – Struktura SOAP zprávy [3]

Obsah zprávy je uložen v těle zprávy, která se běžně skládá z formátovaných XML dat. Na obrázku 3 je zobrazen názorný příklad SOAP zprávy zaslané webové službě. V těle zprávy je požadavek na volání vzdálené funkce GetWeather, která zjistí počasí v dané lokalitě. Specifikace lokality je provedena vyplněním elementů CityName a CountryName.

```
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns:web="http://www.webserviceX.NET">
   <soap:Header/>
   <soap:Body>
      <web:GetWeather>
         <!--Optional:-->
         <web:CityName>Cannes</web:CityName>
         <!--Optional:-->
         <web:CountryName>France</web:CountryName>
      </web:GetWeather>
   </soap:Body>
</soap:Envelope>
```

Obrázek 3 – Požadavek SOAP zprávy [6]

Jak by mohla vypadat odpověď webové služby pro výše zmíněný požadavek je znázorněno na obrázku 4. V těle zprávy se nacházejí elementy, které obsahují informace o počasí v lokalitě, která přišla v daném požadavku.

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <soap:Body>
        <GetWeatherResponse xmlns="http://www.webserviceX.NET">
           <Location>Cannes, France (LFMD) 43-33N 006-57E 9M</Location>
           <Time>Jul 06, 2016 - 12:30 PM</Time>
           <Wind>from the SE (140 degrees) at 6 MPH (5 KT)</Wind>
           <Visibility>greater than 7 mile(s)</Visibility>
           <Temperature> 82 F (28 C)</Temperature>
           <DewPoint> 64 F (18 C)</DewPoint>
           <RelativeHumidity> 54%</RelativeHumidity>
           <Pressure> 29.91 in. Hg (1013 hPa)</Pressure>
           <Status>Success</Status>
        </GetWeatherResponse>
    </soap:Body>
</soap:Envelope>
```

Obrázek 4 – Odpověď SOAP zprávy [6]

3.3.2.2 WSDL

WSDL popisuje kontaktní bod poskytovatele služeb, který je spíše známý jako koncový bod služby neboli endpoint. Obsahuje formální definici koncového rozhraní a také zavádí fyzické umístění služby. Webová služba nabízí dostatečný popis svého rozhraní a tím umožňuje komukoliv konzumovat danou službu. Samotný dokument obsahuje jména dostupných operací, datové typy jejich parametrů, návratových hodnot a také kde se webová služba nachází a jakými protokoly s ní je možno komunikovat. Pokud je znám popis webové služby, poté je možno jednoduše vytvořit SOAP požadavek. SOAP zprávu je možné vytvořit ručně nebo pomocí automatizovaných nástrojů, které ji z daného WSDL dokumentu vygenerují. Protokol WSDL je založen na XML a vznikl za účelem strukturovaně definovat rozhraní webových služeb. Ukázka WSDL dokumentu je dostupná na následujícím odkazu [6]. [1] [4]

3.3.2.3 UDDI

Jak již bylo zmíněno, jedinou podmínkou pro provolání služby, je přístup k popisu služby. S rostoucím počtem služeb uvnitř a vně společnosti, mohou být mechanismy pro oznamování a objevování popisů služeb nezbytné. K tomu slouží registr UDDI, který specifikuje relativně příjemný standard pro definici struktury registrů, které uchovávají popisy služeb. Tento registr umožňuje tedy ukládat základní informace o službě jeho vlastníkovi, rozhraní a také například o podmínkách za jakých je danou webovou službu možno konzumovat. Tyto registry je možné ručně vyhledat a přistupovat k nim programově pomocí standardního rozhraní API. Dřívější koncepce UDDI jako veřejného registru plného odkazů na webové služby se v praxi neosvědčil. V dnešní době registry služeb nacházejí uplatnění zejména ve velkých organizacích. [1]

3.3.3 REST

V této části práce je představen zdrojově orientovaný architektonický styl REST, který se v dnešní době velmi používá při návrhu a tvorbě servisně orientovaných aplikací. Roy Fielding představil pojem REST v jeho disertační práci[1] v roce 2000. Na rozdíl od klasických webových služeb jako je XML-RPC a SOAP, které jsou orientovány procedurálně, REST určuje jak se přistupuje k datům. Je definován souborem architektonických principů, které jsou popsány v následujících podkapitolách. [7]

3.3.3.1 Adresovatelnost

Jak už bylo zmíněno, tak základním kamenem je tzv. zdroj neboli resource, což může být prakticky cokoli. Například nějaký dokument, objekt v databázi, nějaký stav stránky, výsledek nějakého výpočtu nebo prostě webová stránka. Jedinou podmínku, kterou musí každý zdroj splňovat je, že musí mít své jednoznačné umístění neboli URI. Přičemž URI je jednotný identifikátor zdroje, který má standardizovanou podobu. Struktura URI je znázorněna na obrázku 5.

[1] http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

scheme://*host:port/path?queryString#fragment*

Obrázek 5 - Struktura URI [8]

Scheme je protokol, který se používá ke komunikaci pro REST služby. Většinou to je HTTP nebo HTTPS. Host je DNS jméno nebo IP adresa. Poté následuje volitelný port, na kterém běží daný zdroj. Path je hierarchická struktura, která je oddělená zpětným lomítkem, obdobně jako u adresáře na disku. Otazník odděluje path od tzv. queryString, což je seznam parametrů, které jsou reprezentovány v datové struktuře klíč/hodnota a jsou odděleny speciálním znakem „&“. Poslední částí URI je fragment, který je oddělený „#“ a slouží jako identifikátor nějaké části zdroje, například obrázku v dokumentu. V tomto je tedy zásadní rozdíl oproti SOAP, kde se k jednotlivým endpointům přistupuje jako k operaci. [8]

3.3.3.2 Jednotné rozhraní

Pro manipulaci se zdroji je nutné používat sadu definovaných HTTP metod. Každá metoda má specifický význam, účel a je důležité tedy používat HTTP metody dle charakteru operace, které chceme s daným zdrojem provést. Základní HTTP metody jsou popsány v tabulce 1. Je mnohem více HTTP metod, jako například TRACE nebo CONNECT, ale nejsou podporovány RESTem. V případě RESTu se standardně používají jen 4 operace, tzv. CRUD operace, což v HTTP protokolu jsou GET, POST, PUT a DELETE. Je možné použít i vlastní metody, což HTTP standard povoluje, ale není to doporučeno. [7]

HTTP metoda	Popis
GET	Slouží jen pro čtení, tedy k získání aktuální datové reprezentace daného zdroje. Obdoba SELECT v SQL. Je to idempotentní operace, což znamená, že vícenásobné provedení operace má pokaždé stejný výsledek. Příklad: • /tasks/123 – načte datovou reprezentaci úkolu s id 123, • /tasks/state=new – načte seznam úkolů, jenž jsou ve stavu new.
PUT	Vytvoří nebo aktualizuje resource. Operace je obdobou UPDATE v SQL. Metoda je také idempotentní. Příklad: • /tasks/123 {name, state} – změní datovou reprezentaci úkolu s id 123.
DELETE	Používá se ke smazání existujícího resourcu. Obdoba DELETE v SQL. Tato operace je rovněž idempotentní. Identifikátor mazaného resource se nachází v poslaném URI. Příklad: • /tasks/123 – smaže úkol s id 123.
POST	Tato metoda je hodně blízká metodě PUT, ovšem velký rozdíl je, že není idempotentní. To znamená, že s každým dotazem se mění stav daného zdroje. Částečně se jedná o obdobu INSERT v SQL. Příklad: • /tasks/123 {name, state} – vytvoří nový úkol a vrátí identifikátor nově vytvořeného úkolu.
HEAD	Je stejná jako metoda GET, ale vrací jen response code a HTTP hlavičku. Poskytuje jen metadata o požadovaném cíli.
OPTIONS	Vrací možnosti komunikace pro dané URI (jaké podporuje metody).

Tabulka 1 – Základní HTTP metody [9]

3.3.3.3 Orientace na reprezentaci

Při komunikaci přes HTTP metody posíláme na dané URI nějakou reprezentaci zdroje. Preferovaná a nejčastější reprezentace zprávy je ve formátu JSON. Je možné použít i jiné formáty jako je například XML atd. Ovšem vždy je nutné dodržet pravidla protokolu HTTP využitím hlaviček Accept a Content-Type a to z důvodu, aby bylo přesně definováno, jaké formáty daný resource podporuje.

3.3.3.4 Bezstavová komunikace

Toho je docíleno tím, že server si neudržuje stav aplikace (žádná klientská session), pouze odbavuje requesty. Pokud aplikace vyžaduje udržování stavu aplikace, musí si jí zajistit sama. Klientské session údaje by tedy měly být uchovávány na straně klienta a pokud jsou potřeba na serveru, tak je možné je při každém požadavku přenést. To přináší velkou výhodu v podobě škálovatelnosti. [7]

3.3.3.5 HATEOAS

Posledním principem je HATEOAS, který říká, že klient by měl znát pouze vstupní bod (URL) do aplikace. Ostatní navigaci v aplikaci by již měl získávat ze serveru z response volání prostřednictvím hyperlinků. Kód 1 zobrazuje příklad použití, kdy v prvním kroku se načte úkol pomocí resource /tasks/123, který vrátí datovou reprezentaci. Ta mimo samostatných dat obsahuje navíc ještě informace o tom, jaké akce je možné s daným úkolem provést. To znamená, jakou operaci je možno volat a kde se v API nachází. Jak pojmenovávat jednotlivé hyperlinky je částečně standardizováno viz následující odkaz [10].

```
{
name : “lorem ipsum”,
state : “NEW”,
_links : {
delete : {uri: “/tasks/123”, method: DELETE},
self: {uri: /tasks/123/, method: GET}
}
```

Kód 1 – Hyperlinky v odpovědi zprávy

3.3.3.6 Response kódy

Při implementaci REST API je důležité správné použití HTTP response kódů, což jsou kódy, které nám sdělují dostatečnou odpověď o tom, co se s daným resourcem po zavolání RESTové metody stalo. Tabulka 2 vysvětluje základní HTTP response kódy.

Kód	Význam	Popis
200	OK	Standardní odpověď pro úspěšný HTTP požadavek.
201	Created	Vytvořen nový resource.
204	No content	Request je úspěšný, ale prázdná odpověď. (např. výsledek vyhledávání)
304	Not modified	Response beze změn, klient může využít záznam z cache.
400	Bad request	Neplatný/Nepodporovaný request. Špatně zadané parametry nebo formát obsahu payload.
401	Unauthorized	Vyžadována autentizace.
403	Forbidden	Přístup na resource odmítnut. Uživatel nemá přístup.
404	Not found	Resource se zadaným identifikátorem nebyl nalezen.
500	Internal server error	Chyba serveru, typicky včetně stack trace v payloadu response. Stack trace by se neměl v produkčním prostředí dostat na klienta. Správné ošetření na serveru je logovaní chyby a vrácení 500 s detailem chyby v payloadu.
503	Service unavailable	Nedostupný backend.

Tabulka 2 – Význam HTTP response kódů [11]

3.3.4 Datové formáty

V této kapitole jsou popsány datové formáty, které se využívají pro přenos dat mezi klientem a serverem v případě provolání webové služby.

3.3.4.1 JSON

Je jednotný formát pro výměnu dat. V dnešní době je jedním z nejpoužívanějších formátů pro výměnu dat na webu. Je to vlastně JavaScriptový zápis objektů, který především slouží k výměně krátkých strukturovaných dat webovými aplikacemi. Oproti XML je využíván zejména moderními AJAX aplikacemi. Syntaxe je převzatá z jazyku JavaScript. Jeho výhodou je, že je člověkem lehce čitelný a i zapisovatelný.

```
{"tasks":[
        {"name": "lorem ipsum", "state": "NEW"},
        {"name": "lorem ipsum", "state": "DONE"},
        {"name": "lorem ipsum", "state": "DONE"}
]}
```

Kód 2 – Příklad JSON formátu

3.3.4.2 XML

Je rozšiřitelný značkovací jazyk, který slouží v kontextu webových služeb pro přenos dat. Každý XML dokument se skládá z tzv. elementů, které jsou do sebe navzájem vnořené. Elementy se v textu vyznačují pomocí tzv. tagů. Elementům ve většině případů odpovídají dva tagy a to počáteční a ukončovací.

```
<?xml version="1.0" encoding="UTF-8"?>
<tasks>
    <task>
        <name>lorem ipsum</<name>
        <state>NEW</state>
    </task>
    <task>
        <<name>lorem ipsum</<name>
        <state>DONE</state>
    </task>
    <task>
        <<name>lorem ipsum</<name>
        <state>DONE</state>
    </task>
</tasks>
```

Kód 3 – Příklad XML formátu

4 Praktická část

V této části diplomové práce je popsán celý proces vývoje zvolené webové aplikace a to od definování požadavků aplikace, její analýzy, návrhu až po výslednou implementaci. Cílem praktické části je funkční webová aplikace pro správu úkolů. V celém procesu jsou dodržovány standardy pro vývoj softwaru dle metodiky Unified process. Tato metodika je založena na iterativním a přírůstkovém procesu. Jednotlivé iterace se skládají z následujících činností:

- Požadavky – zachycují, co by měla aplikace dělat,
- Analýza – stanovení proveditelnosti,
- Návrh – technický návrh jednotlivých požadavků,
- Implementace – tvorba softwaru,
- Testování – ověření, zda implementace funguje tak, jak se od ní očekává.

4.1 Požadavky na aplikaci

V prvotní fázi vývoje webové aplikace je nutné definovat základní požadavky a představu o cílovém řešení budované aplikace. Zdrojem požadavků nebyl reálný zákazník. Jde o ukázkovou aplikaci neboli prototyp, kde sám autor definoval základní požadavky, které jsou základem všech aplikací, neboť vyjadřují, co by měla aplikace dělat, bez ohledu na způsob dosažení dané funkcionality. V podstatě se rozlišují dva typy požadavků. Jsou to funkční požadavky, které určují, jaké chování systém bude nabízet a druhý typ požadavků jsou nefunkční požadavky, které definují vlastnosti nebo omezující podmínky daného systému, za nichž musí pracovat. Funkční požadavky jsou vyjádřeny pomocí případů užití tzv. use case. Případ užití popisuje funkce, které aplikace poskytuje k užitku jednoho nebo více aktérů. Všechny případy užití jsou zobrazeny na diagramu 1. [12]

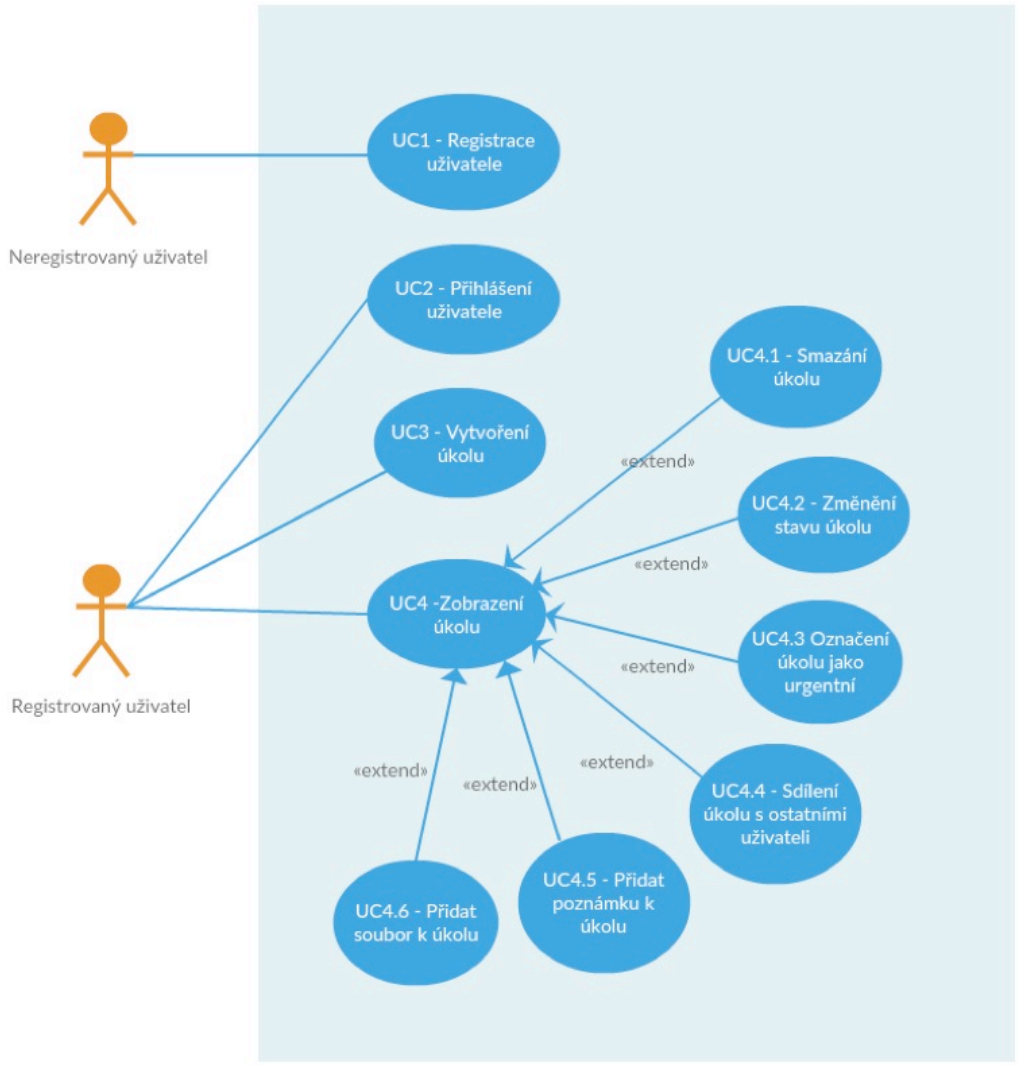


Diagram 1 – Use case diagram požadavků na aplikaci

Co se týče nefunkčních požadavků, tak by aplikace měla být z obecného hlediska použitelná, atraktivní, škálovatelná, udržitelná, zabezpečená a do budoucna snadno rozšiřitelná. V případě požadavku na výkon se u serverové části aplikace většinou měří počet transakcí provedených za určitý časový úsek a u klientských aplikací zase čas reakce aplikace na uživatelův požadavek. V případě implementované aplikace nejsou na výkon kladeny žádné specifické požadavky, neboť se reálně nebude nasazovat do produkce. Jak už bylo zmíněno, slouží jako prototyp. S tím souvisí i dostupnost aplikace. Webová aplikace by měla být optimalizována na nejrozšířenějších prohlížečích. Zabezpečení aplikace musí být dostatečné, tak aby byly chráněny osobní údaje uživatelů.

4.2 Analýza a návrh webové aplikace

Na základě sběru požadavků byla provedena analýza a následný návrh webové aplikace. Záměrem analýzy je tvorba analytického modelu, který se zaměřuje na to, co aplikace musí umět, ale nezabývá se detaily týkajícími se způsobu, jakým to udělá. Tuto otázku následně řeší návrh aplikace. Výsledný návrh řešení je popsán v následujících podkapitolách.

4.2.1 Datový model

Kapitola popisuje datový model aplikace, jak z pohledu logických vazeb ve schématu, tak z pohledu obsahu. Aplikace má za cíl evidovat následující vazby/informace:

- všechny uživatele, kteří přistupují do aplikace,
- uživatelské role každého uživatele,
- všechny spravované úkoly uživatelů,
- vztahy mezi uživateli a úkoly,
- poznámky k jednotlivým úkolům,
- soubory k jednotlivým úkolům.

Na diagramu 2 je znázorněno schéma datového modelu. Jedná se o fyzický datový model, který konkrétně popisuje uložení dat a plnohodnotně znázorňuje vazby mezi databázovými tabulkami. Popisuje také primární a cizí klíče, atributy a jejich datové typy. Význam jednotlivých tabulek a sloupců je popsán v následujících tabulkách.

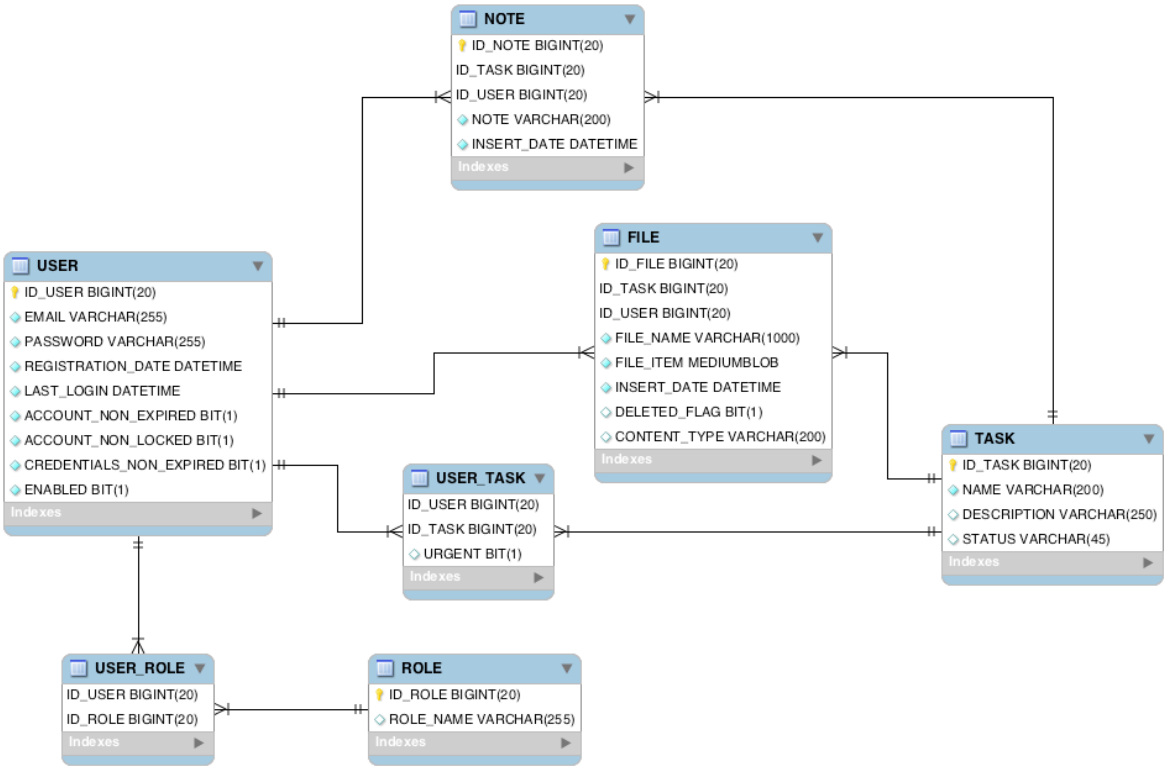


Diagram 2 – Datový model

Tabulka USER obsahuje všechny uživatele, kteří se zaregistrovali do aplikace a o každém z nich uchovává všechny potřebné informace.

Atribut	Popis
ID_USER	Unikátní identifikátor uživatele.
EMAIL	Emailová adresa.
PASSWORD	Heslo uživatele.
REGISTRATION_DATE	Datum registrace uživatele.
LAST_LOGIN	Poslední přihlášení uživatele.
ACCOUNT_NON_EXPIRED	Příznak, jestli nevypršela platnost uživatelského účtu.
ACCOUNT_NON_LOCKED	Příznak, jestli uživatelský účet není uzamčený.
CREDENTIALS_NON_EXPIRED	Příznak, jestli nevypršela platnost přihlašovacích údajů.
ENABLED	Příznak, jestli je práce s uživatelem povolena nebo zakázána.

Tabulka 3 – Databázová tabulka USER [vlastní zpracování]

Tabulka USER_ROLE je tzv. vazební tabulka, která uchovává vztahy mezi uživateli a uživatelskými rolemi. Jde o vztah M:N, což znamená, že uživatel může mít více uživatelských rolí a zároveň uživatelské role mohou být přiřazeny více uživatelům.

Atribut	Popis
ID_USER	Cizí klíč ke sloupci ID_USER v tabulce USER.
ID_ROLE	Cizí klíč ke sloupci ID_ROLE v tabulce ROLE.

Tabulka 4 – Databázová tabulka USER_ROLE [vlastní zpracování]

Tabulka ROLE je jen výčet uživatelských rolí, které je možno přiřadit uživatelům. Na základě těchto rolí, je řízen přístup v aplikaci k jednotlivým obrazovkám, operacím atd. Výčet rolí je následující:

- ROLE_USER
- ROLE_ADMIN

Atribut	Popis
ID_ROLE	Unikátní identifikátor role.
ROLE_NAME	Název uživatelské role.

Tabulka 5 – Databázová tabulka ROLE [vlastní zpracování]

Tabulka USER_TASK plní obdobnou funkci jako tabulka USER_ROLE. Uchovává tedy vztahy mezi uživateli a jejich úkoly.

Atribut	Popis
ID_USER	Cizí klíč ke sloupci ID_USER v tabulce USER.
ID_TASK	Cizí klíč ke sloupci ID_TASK v tabulce TASK.
URGENT	Příznak, který nám uchovává informaci o tom, zda je pro daného uživatele daný úkol urgentní.

Tabulka 6 – Databázová tabulka USER_TASK [vlastní zpracování]

Tabulka TASK uchovává data o všech vytvořených úkolech, které uživatelé vytvořili.

Atribut	Popis
ID_TASK	Unikátní identifikátor úkolu.
NAME	Název úkolu.
DESCRIPTION	Popis úkolu.
STATUS	Stav úkolu.

Tabulka 7 – Databázová tabulka TASK [vlastní zpracování]

Tabulka NOTE eviduje poznámky, které jsou vloženy k jednotlivým úkolům konkrétním uživatelem.

Atribut	Popis
ID_NOTE	Unikátní identifikátor poznámky.
ID_TASK	Cizí klíč ke sloupci ID_TASK v tabulce TASK.
ID_USER	Cizí klíč ke sloupci ID_USER v tabulce USER.
NOTE	Obsah poznámky.
INSERT_DATE	Čas vložení poznámky.

Tabulka 8 – Databázová tabulka NOTE [vlastní zpracování]

Tabulka FILE uchovává soubory, které jsou vloženy k jednotlivým úkolům konkrétním uživatelem.

Atribut	Popis
ID_FILE	Unikátní identifikátor souboru.
ID_TASK	Cizí klíč ke sloupci ID_TASK v tabulce TASK.
ID_USER	Cizí klíč ke sloupci ID_USER v tabulce USER.
FILE_NAME	Název souboru.
FILE_ITEM	Konkrétní soubor uložený v datové struktuře BLOB.
INSERT_DATE	Čas vložení souboru.
DELETED_FLAG	Příznak, jestli je soubor smazaný.
CONTENT_TYPE	Formát souboru.

Tabulka 9 – Databázová tabulka FILE [vlastní zpracování]

4.2.2 Procesní model

Kapitola popisuje základní procesy/postupy v implementované aplikaci. Realizace případu užití názorně ukazuje spolupráci skupin objektů pro dosažení požadovaného chování. Převádí se tedy případ užití na digramy tříd a interakcí. V případě implementované aplikace není zapotřebí vytvářet digramy interakcí pro jednotlivé případy užití, neboť se nejedná o složité procesy. Jsou popsány jen případy užití, kde je potřeba více objasnit jednotlivé interakce mezi entitami. [12]

4.2.2.1 Vytvoření úkolu

Při vytváření úkolu musí být uživatel přihlášený, jinak mu aplikace nezobrazí tlačítko pro vytvoření nového úkolu. Přihlášený uživatel vyvolá obrazovku pro vytvoření úkolu, zadá vstupní data a klikne na tlačítko vytvořit. Proběhne validace vstupních dat. Pokud jsou data v pořádku, úkol se vytvoří. V opačném případě se zobrazí validační hláška. Po úspěšném vytvoření úkolu se v databázi vytvoří záznam v entitě TASK a také se vytvoří vztah mezi přihlášeným uživatelem a daným úkolem ve vazební entitě USER_TASK. Při vytvoření úkolu se defaultně nastaví stav úkolu na „TODO“ a atribut URGENT v tabulce USER_TASK na false. To značí, že úkol je v stavu ke zpracování a že není urgentní.

4.2.2.2 Operace s úkolem

S vytvořeným úkolem je možné provádět různé operace. Může je provádět jen uživatel, který je přihlášený a má k danému úkolu vazbu v tabulce USER_TASK. Výčet operací je následující:

- Smazání úkolu – v případě mazání úkolu se daný úkol nesmaže z databáze, ale jen se mu nastaví stav na „CANCELED“,
- Změna stavu úkolu – stav úkolu se změní v tabulce TASK v atributu STATE. Stav úkolu může procházet stavy dle stavového diagramu 3,
- Označení úkolu jako urgentní – uživatel bude moci nastavit úkol jako urgentní, což v databázi bude znázorněno pomocí atributu URGENT v tabulce USER_TASK,
- Sdílení úkolu – uživatel může úkol sdílet s ostatními uživateli. Po sdílení úkolu s vybraným uživatelem se v databázové entitě USER_TASK vytvoří nová vazba mezi sdíleným uživatelem a daným úkolem,
- Přidání poznámky k úkolu – po přidání poznámky k vybranému úkolu se v tabulce NOTE vytvoří záznam, který bude obsahovat informace o poznámce. Také bude obsahovat id uživatele, jenž poznámku přidal a id úkolu, které značí k jakému úkolu byla poznámka přidána,
- Přidání souboru k úkolu – v případě přidání souboru k vybranému úkolu se v tabulce FILE vytvoří záznam, který bude uchovávat samotný soubor a také další

informace vztahující se k danému souboru. Obdobně jako u poznámky bude záznam obsahovat id uživatele, jenž soubor nahrál a také id úkolu, aby bylo zřejmé k jakému úkolu se soubor nahrál.

4.2.2.3 Stavy úkolu

Každý úkol má svůj stav, ve kterém se aktuálně nachází. Tato informace je uložena u každého úkolu v entitě TASK ve sloupci STATE. Na stavovém digramu 3 jsou znázorněny stavy, které mohou nastat při práci s úkolem. Význam jednotlivých stavů je následující:

- TODO – nově vytvořený úkol, který je potřeba vyřešit,
- DONE – úkol, který byl úspěšně vyřešen,
- CANCELED – úkol, který byl zrušen/smazán.

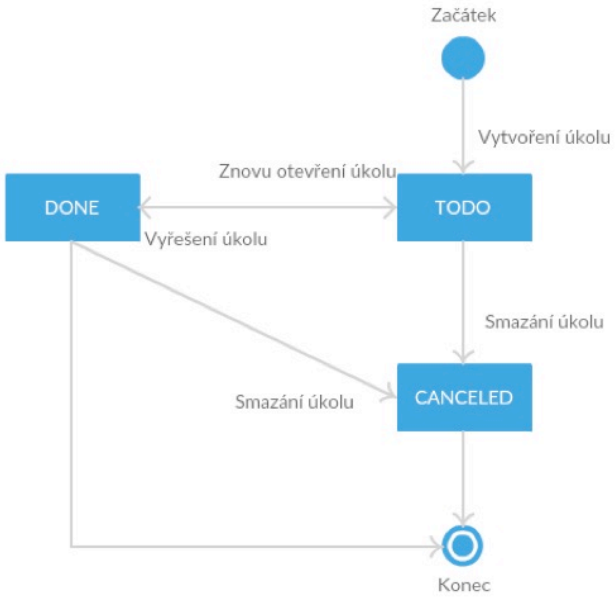


Diagram 3 – Stavy úkolu

4.2.3 Grafické uživatelské rozhraní

V této kapitole jsou popsány jednotlivé obrazovky pomocí tzv. wireframů. Wireframe je grafické rozložení prvků na dané obrazovce, jenž slouží jako náhled daného řešení. Wireframe minimalizuje rozdíl, který může nastat mezi původním zadáním a výslednou implementací. K vytvoření jednotlivých wireframů byl využit program Balsamiq Mockup.

4.2.3.1 Domovská stránka

Všichni uživatelé bez ohledu na jejich oprávnění mohou přistupovat na domovskou stránku aplikace. Tato stránka informuje uživatele o funkcionalitách dané aplikace. Pomocí této stránky se bude moci registrovaný uživatel přesměrovat na přihlašovací formulář a v případě, že uživatel ještě není zaregistrován do aplikace, tak bude moci přejít na registrační formulář.

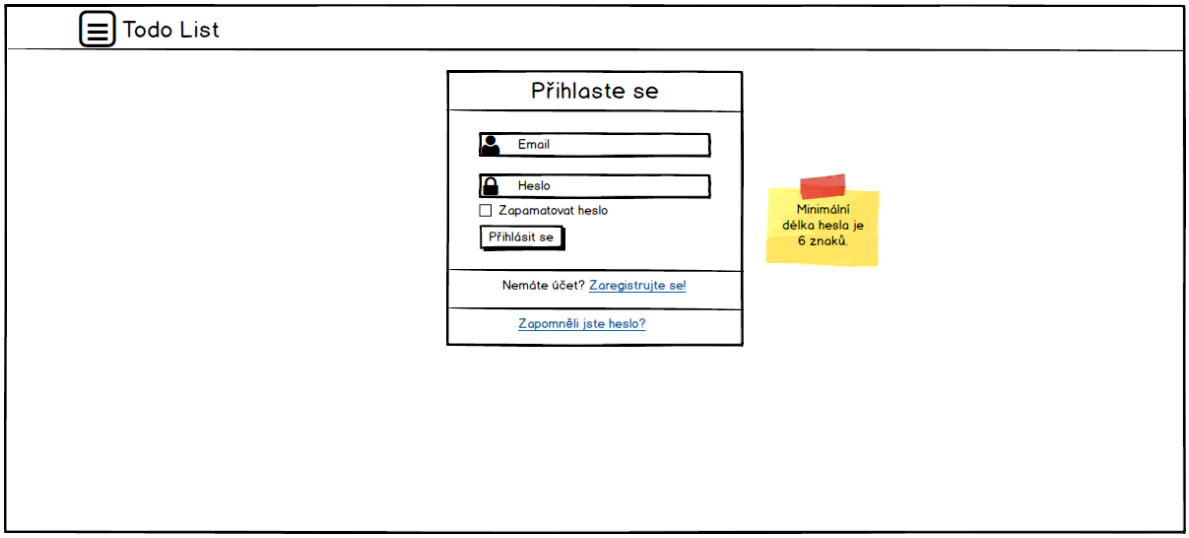
4.2.3.2 Registrace uživatele

Neregistrovaní uživatelé se mohou pomocí standardního registračního formuláře, registrovat do aplikace a využívat nabízených služeb. Při registraci uživatel podléhá validaci vstupních dat, které zadává do formulářů. Do textového pole email bude nutné zadat platnou emailovou adresu a pole heslo bude vyžadovat nejméně 6 znaků. Při registraci je nutné kontrolovat unikátnost emailů, aby nedošlo k vytvoření dvou stejných uživatelů.

Obrázek 6 – Obrazovka s registračním formulářem [vlastní zpracování]

4.2.3.3 Přihlášení uživatele

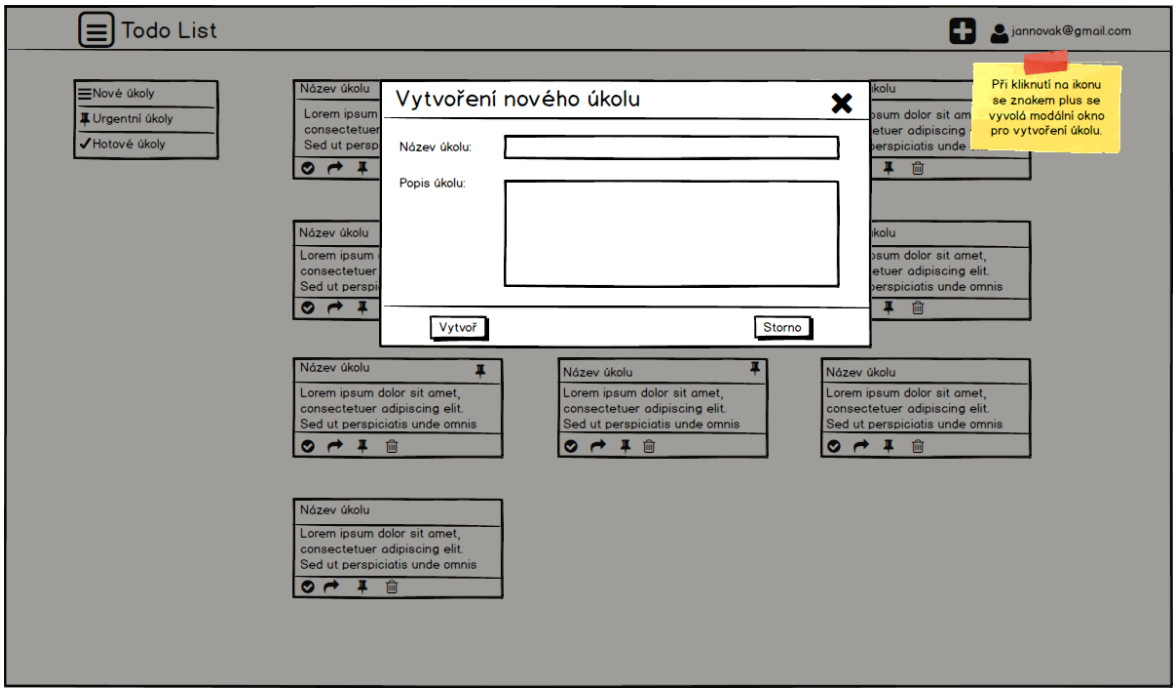
Každý uživatel může přistoupit na přihlašovací obrazovku, ovšem jen registrovaný uživatel se může úspěšně přihlásit do aplikace. Do aplikace se přistupuje přes standardní přihlašovací formulář. Pokud registrovaný uživatel zapomene heslo, je možné na registrační email poslat link pro vytvoření nového hesla. V případě zadání špatných přihlašovacích údajů se zobrazí chybová hláška s popisem problému.



Obrázek 7 – Obrazovka s přihlašovacím formulářem [vlastní zpracování]

4.2.3.4 Vytvoření úkolu

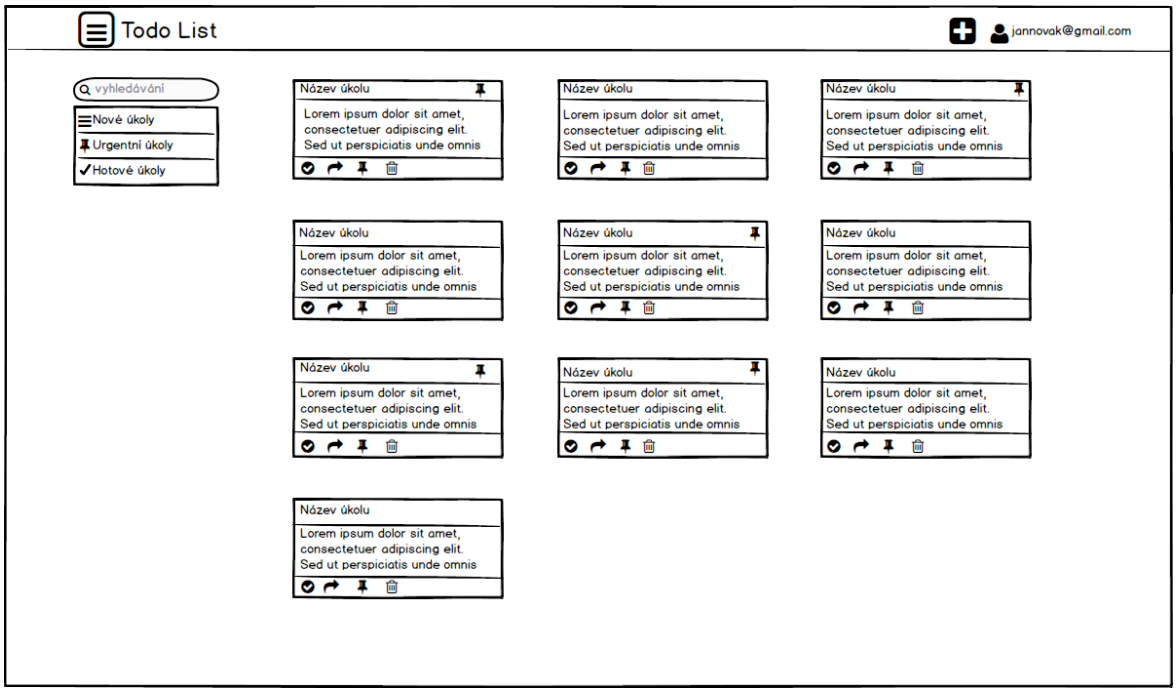
Modální okno sloužící pro vytvoření nového úkolu se vyvolá kliknutím na ikonu plus v pravém horním rohu. Tlačítko s ikonou plus bude dostupné ze všech obrazovek, pokud bude uživatel přihlášený. V modálním okně se nacházejí vstupní pole pro zadání názvu a popisu úkolu. Po zadání povinných polí a kliknutím na tlačítko vytvořit zmizí modální okno, daný úkol se založí a následně se zobrazí v seznamu úkolů. Vstupní pole podléhají validaci. Obě pole jsou povinné a maximální délka znaků může být v případě názvu úkolu 30 znaků a u popisu úkolu 200 znaků. Pokud si uživatel rozmyslí vytvoření úkolu, může modální okno zavřít křížkem v pravém horním rohu nebo použít tlačítko storno.



Obrázek 8 – Obrazovka pro vytvoření nového úkolu [vlastní zpracování]

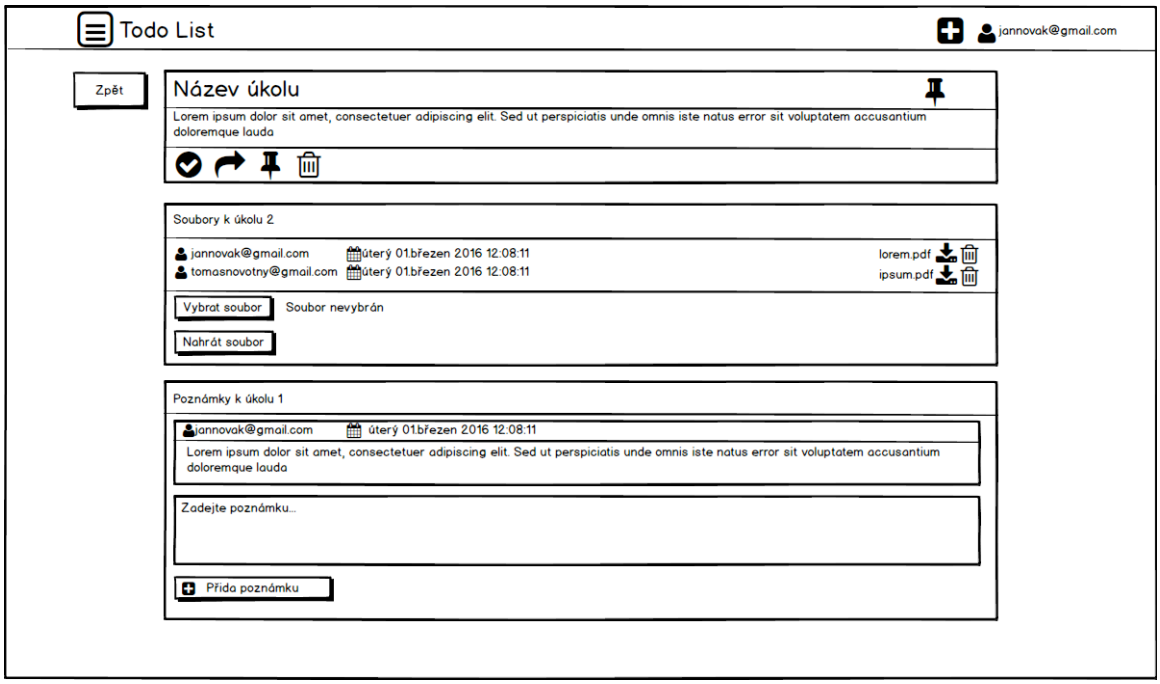
4.2.3.5 Zobrazení úkolu

Uživatel bude moci zobrazovat úkol dvojím způsobem. Prvním způsob zobrazení úkolu je znázorněn na obrázku 9, kde se uživateli zobrazí seznam všech úkolů na které má práva. Úkoly se budou automaticky třídit na základě jejich stavu a uživatel bude moci pomocí levého postranního menu přecházet mezi jednotlivými skupinami úkolů. Nad levým postranním menu se nachází vyhledávací pole, které umožní uživateli vyhledávat úkoly dle zadaného textu. Vyhledávání bude ignorovat velikost písmen a bude vyhledávat zadaný text jak v názvu, tak i v popisu úkolu.



Obrázek 9 – Obrazovka se seznamem úkolů [vlastní zpracování]

Druhý způsob zobrazení úkolu je znázorněn na obrázku 10. Tento způsob zobrazuje konkrétní úkol, který byl vybrán ze seznamu úkolů kliknutím na název vybraného úkolu. V tomto případě se tedy zobrazí detail daného úkolu, který zobrazuje standardní panel s úkolem a dostupnými operacemi. Navíc na této obrazovce je možné přidávat k danému úkolu poznámky a soubory, které se následně zobrazují pod panelem úkolu. Je možno nahrát jakýkoliv typ souboru, ale maximální povolená velikost je 5 MB. Nahrané soubory je poté možno opětovně stáhnout nebo smazat. Poznámku lze přidat, ale nelze ji smazat. Maximální počet znaků, které lze vložit do poznámky je 200 znaků. Ať už se jedná o soubor nebo poznámku, tak je vždy zřejmé, jaký uživatel to vložil a v jaký čas.



Obrázek 10 – Obrazovka s detailem úkolu [vlastní zpracování]

4.2.3.6 Operace s úkolem

Každý úkol bude zobrazen v samostatném panelu, jenž se skládá ze tří částí. První část sděluje uživateli, jak se úkol nazývá a na pravé straně zobrazuje různé tagy, které se vztahují ke konkrétnímu úkolu. Například se tam bude zobrazovat informace o tom, že daný úkol je označený jako urgentní nebo, že je sdílený s jinými uživateli. Prostřední část panelu zobrazuje popis úkolu a poslední část slouží pro operace s úkolem. Na základě stavu úkolu se zobrazují různé operace, které je možné s daným úkolem provést. Výčet operací je znázorněn v tabulce 10, kde je typ operace a stav úkolu ve kterém se musí daný úkol nacházet, aby se daná operace mohla provést.

Ikona	Operace	Stav úkolu
	Označení úkolu jako nový.	DONE
	Označení úkolu jako hotový.	TODO
	Sdílení úkolu.	TODO
	Označení úkolu jako urgentní.	TODO
	Smazání úkolu.	TODO, DONE

Tabulka 10 – Zobrazení jednotlivých operací na základě stavu úkolu [vlastní zpracování]

4.2.4 Návrh REST služeb

Pro komunikaci se serverem byl vybrán architektonický styl REST, kvůli jeho jednoduchosti, transparentnosti a především snadné škálovatelnosti. Výslednou implementací bude tedy RESTové rozhraní.

Jelikož REST je založen na reprezentaci a manipulaci s jednotlivými zdroji, tak jsou v této části práce identifikovány všechny zdroje a jejich případné vztahy. Prvním krokem při návrhu je identifikace hlavních datových entit, které se nacházejí v datovém modelu aplikace. Vybírají se jen silné entity, jenž reprezentují hlavní logiku aplikace a jsou schopny existovat samostatně jako stavící blok aplikační logiky. Při návrhu API je velmi důležitý návrh struktury URL adres jednotlivých zdrojů. Správně navržené URL adresy umožňují snadnější implementaci klientských aplikací a usnadňují porozumění aplikačního rozhraní. URL adresy nikdy nesmí obsahovat názvy CRUD operací. Hierarchie jednotlivých zdrojů není většinou jednoduchá, neboť zdroje mají mezi sebou vztahy, které je možno znázornit jako náležitost k jinému zdroji. Při návrhu API se tyto vztahy v hierarchické struktuře URL adresy zobrazují jako podřazené entity, které jsou umístěny pod jejich rodiče.

Důležitou součástí návrhu jsou návratové kódy, které musejí odpovídat výsledkům odpovědí HTTP požadavků. V následujících tabulkách jsou definovány jednotlivé RESTové zdroje s URL adresou a bližším popisem. Jsou zde uvedeny očekávané a chybové stavy, které mohou nejčastěji nastat. Pokud v aplikaci nastane neočekávaná chyba, tak aplikace tuto výjimku odchytí a vrátí se standardně HTTP kód 500. Samozřejmě se mohou vrátit i jiné HTTP kódy viz kapitola 3.3.3.6. [7]

Popis	Slouží pro přihlášení uživatele.
URL	api/login
Metoda	POST
Payload	{"username": string, "password": string}
Úspěšná odpověď	HTTP kód: 200 – úspěšné přihlášení, v hlavičce odpovědi vrácen přihlašovací token
Chybná odpověď	HTTP kód: 401 – špatné přihlašovací údaje

Tabulka 11 – Zdroj pro přihlášení uživatele [vlastní zpracování]

Popis	Slouží pro registraci uživatele.
URL	api/register
Metoda	POST
Payload	{"username": string, "password": string}
Úspěšná odpověď	HTTP kód: 200 – uživatel zaregistrován
Chybná odpověď	HTTP kód: 409 – tento uživatel již existuje

Tabulka 12 – Zdroj pro registraci uživatele [vlastní zpracování]

Popis	Slouží pro prodloužení platnosti tokenu.
URL	api/renew
Metoda	POST
Payload	{}
Úspěšná odpověď	HTTP kód: 200 – úspěšné prodloužení tokenu, prodloužený token vrácen v hlavičce odpovědi
Chybná odpověď	HTTP kód: 401 – neoprávněný přístup

Tabulka 13 – Zdroj pro prodloužení platnosti tokenu [vlastní zpracování]

Popis	Vrátí úkol podle id úkolu.
URL	api/tasks/{id}
Metoda	GET
Úspěšná odpověď	HTTP kód: 200 {"idTask": number, "name": string, "description": string, "urgent": boolean, "taskStatus": string}
Chybná odpověď	HTTP kód: 401 – neoprávněný přístup

Tabulka 14 – Zdroj pro získání úkolu podle id [vlastní zpracování]

Popis	Vrátí seznam úkolů podle stavu úkolu.
URL	api/tasks?status={status}
Metoda	GET
Úspěšná odpověď	HTTP kód: 200 [{"idTask": number, "name": string, "description": string, "urgent": boolean, "taskStatus": string}]
Chybná odpověď	HTTP kód: 401 – neoprávněný přístup

Tabulka 15 – Zdroj pro získání seznamu úkolů podle stavu [vlastní zpracování]

Popis	Vrátí seznam urgentních úkolů.
URL	api/tasks/urgent
Metoda	GET
Úspěšná odpověď	HTTP kód: 200 [{"idTask": number, "name": string, "description": string, "urgent": boolean, "taskStatus": string}]
Chybná odpověď	HTTP kód: 401 – neoprávněný přístup

Tabulka 16 – Zdroj pro získání seznamu urgentních úkolů [vlastní zpracování]

Popis	Vytvoří nový úkol.
URL	api/tasks
Metoda	POST
Payload	{"name": string, "description": string}
Úspěšná odpověď	HTTP kód: 201 – úkol vytvořen a v odpovědi vráceno id vytvořeného úkolu
Chybná odpověď	HTTP kód: 401 – neoprávněný přístup

Tabulka 17 – Zdroj pro vytvoření nového úkolu [vlastní zpracování]

Popis	Smaže úkol podle id úkolu.
URL	api/tasks/{id}
Metoda	DELETE
Úspěšná odpověď	HTTP kód: 200 – úkol úspěšně smazán
Chybná odpověď	HTTP kód: 401 – neoprávněný přístup

Tabulka 18 – Zdroj pro smazání úkolu [vlastní zpracování]

Popis	Aktualizuje stav úkolu.
URL	api/tasks/{id}?status={status}
Metoda	PUT
Úspěšná odpověď	HTTP kód: 200 – úkol úspěšně aktualizován
Chybná odpověď	HTTP kód: 401 – neoprávněný přístup

Tabulka 19 – Zdroj pro aktualizování stavu úkolu [vlastní zpracování]

Popis	Označí úkolu jako urgentní.
URL	api/tasks/{id}?urgent={boolean}
Metoda	PUT
Úspěšná odpověď	HTTP kód: 200 – úkol označen jako urgentní
Chybná odpověď	HTTP kód: 401 – neoprávněný přístup

Tabulka 20 – Zdroj pro označení úkolu jako urgentní [vlastní zpracování]

Popis	Sdílí úkol s jiným uživatelem.
URL	api/tasks/{id}?share={username}
Metoda	PUT
Úspěšná odpověď	HTTP kód: 200 – úkol sdílen s daným uživatelem
Chybná odpověď	HTTP kód: 401 – neoprávněný přístup

Tabulka 21 – Zdroj pro sdílení úkolu s jiným uživatelem [vlastní zpracování]

Popis	Vrátí seznam poznámek ke konkrétnímu úkolu.
URL	api/tasks/{id}/notes
Metoda	GET
Úspěšná odpověď	HTTP kód: 200 [{"idNote": number, "idTask": number, "email": string, "note": string, "insertDate": date}]
Chybná odpověď	HTTP kód: 401 – neoprávněný přístup

Tabulka 22 – Zdroj pro získání seznamu poznámek [vlastní zpracování]

Popis	Vytvoří novou poznámku ke konkrétnímu úkolu.
URL	api/tasks/{id}/notes
Metoda	POST
Payload	string
Úspěšná odpověď	HTTP kód: 201 – poznámka úspěšně vytvořena a v odpovědi vráceno id poznámky
Chybná odpověď	HTTP kód: 401 – neoprávněný přístup

Tabulka 23 – Zdroj pro vytvoření poznámky ke konkrétnímu úkolu [vlastní zpracování]

Popis	Nahraje soubor ke konkrétnímu úkolu.
URL	api/tasks/{id}/files
Metoda	POST
Payload	multipart/form-data
Úspěšná odpověď	HTTP kód: 201 – soubor úspěšně nahrán a v odpovědi vráceno id souboru
Chybná odpověď	HTTP kód: 401 – neoprávněný přístup

Tabulka 24 – Zdroj pro nahrání souboru ke konkrétnímu úkolu [vlastní zpracování]

Popis	Vrátí seznam souborů ke konkrétnímu úkolu.
URL	api/tasks/{id}/files
Metoda	GET
Úspěšná odpověď	HTTP kód: 200 [{"idFile": number, "idTask": number, "email": string, "fileName": string, "insertDate": date}]
Chybná odpověď	HTTP kód: 401 – neoprávněný přístup

Tabulka 25 – Zdroj pro získání seznamu souborů [vlastní zpracování]

Popis	Vrátí konkrétní soubor podle id souboru.
URL	api/tasks/{idTask}/files/{idFile}
Metoda	GET
Úspěšná odpověď	HTTP kód: 200 – multipart/form-data, HTTP kód: 404 – soubor nenalezen
Chybná odpověď	HTTP kód: 401 – neoprávněný přístup

Tabulka 26 – Zdroj pro stáhnutí konkrétního souboru [vlastní zpracování]

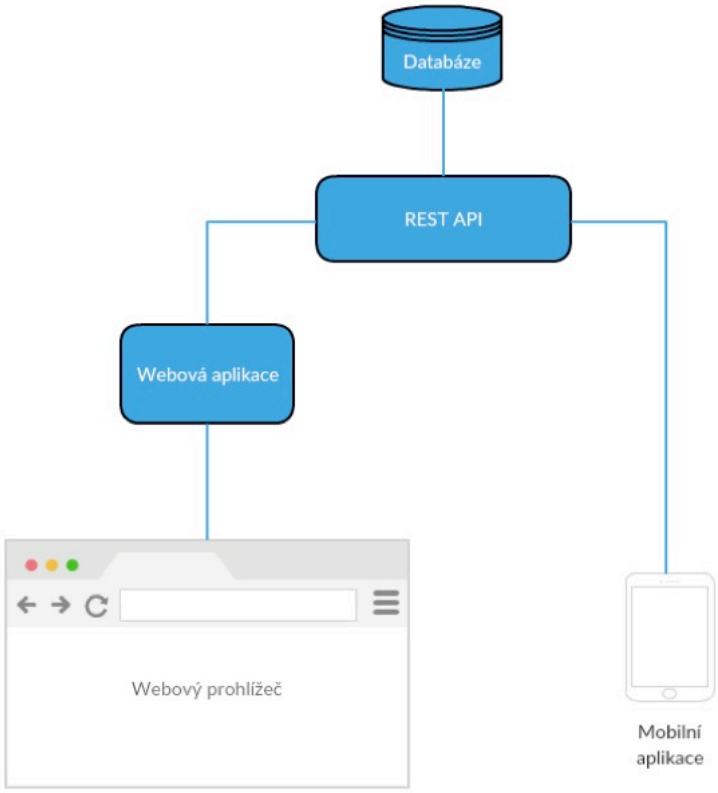
Popis	Smazání konkrétního souboru podle id souboru.
URL	api/tasks/{idTask}/files/{idFile}
Metoda	DELETE
Úspěšná odpověď	HTTP kód: 200 – soubor úspěšně smazán
Chybná odpověď	HTTP kód: 401 – neoprávněný přístup

Tabulka 27 – Zdroj pro smazání konkrétního souboru [vlastní zpracování]

4.3 Implementace webové aplikace

Cílem implementace je převod navrhovaného modelu do spustitelného kódu. Implementace výsledného řešení je rozdělena na dvě části. První část se zaměřuje na implementaci RESTových služeb, které poběží na serveru a druhá část této kapitoly popisuje implementaci klientské aplikace, která dané služby konzumuje.

Aplikace je standardně rozdělena do tří vrstev, jimiž jsou datová, aplikační a prezentační vrstva. Tří vrstvá architektura je v dnešní době velmi využívána, neboť její výhodou je lepší rozdělení výkonu mezi zařízení uživatele a server. Prezentační vrstva , tak může klidně běžet i na velmi slabých zařízeních. Rozložení vrstev ve výsledné aplikaci je znázorněno na obrázku 11. Vytvořené REST API bude umožňovat přistupovat a ukládat data do databáze s úkoly z kterékoliv platformy.



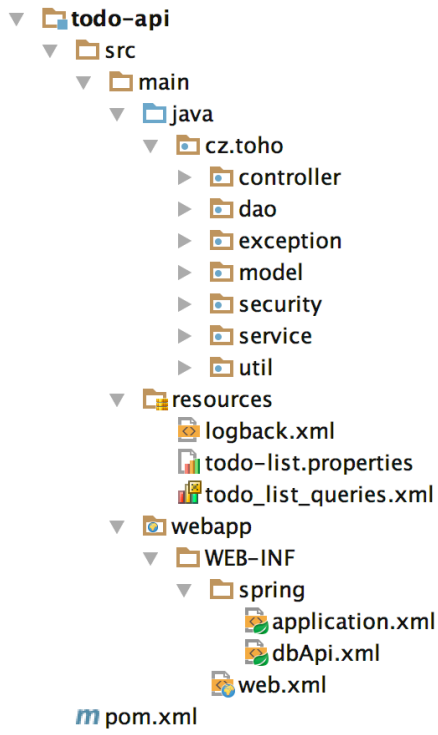
Obrázek 11 – Architektura aplikace [vlastní zpracování]

4.3.1 Serverová část

Pro vývoj webové aplikace byl zvolen programovací jazyk Java. Pro efektivní programování je důležité zvolit vhodný framework, který slouží jako podpora při vývoji aplikace. Pro implementaci serverové části aplikace byl zvolen framework Spring, který je v dnešní době nejvíce rozšířený pro Java platformu. Spring je open-source aplikační framework, jenž se nesoustřeďuje jen na jednu vrstvu aplikačního modelu, ale podporuje všechny. Spring je postaven na tzv. interface, tím pádem se dá jednoduše rozšířit. Více informací o Spring frameworku je možno získat na následujícím odkazu [13]. Na straně serveru je umístěna databáze a webová aplikace. Tato aplikace poskytuje data pomocí RESTových služeb. Celé řešení je vyvíjeno na tzv. localhostu, přičemž pro běh serverové části je využíván webový server Apache Tomcat a pro uložení dat slouží MySQL databáze.

4.3.1.1 Adresářová struktura aplikace

Před konkrétní implementací jednotlivých funkcionalit je důležité si definovat základní strukturu aplikace. Jelikož pro správu, řízení a sestavení aplikace byl vybrán nástroj Apache Maven, tak adresářová struktura projektu je dána dle jeho standardu. Apache Maven slouží ke správě a sestavování aplikací nad platformou Java. Základem Maven je vytvoření objektového modelu nad daným zdrojovým kódem, s nímž je možné dále provádět různé operace. Nejpoužívanějšími operacemi jsou kompilace, vytvoření balíků a nebo release aplikace. Objektový model projektu je definován v XML souboru, který se nazývá pom.xml (Project Object Model). Tento soubor se nachází v kořenovém adresáři projektu aplikace a obsahuje informace a konfigurační detaily pro různé operace. Více informací o nástroji Apache Maven je možné zjistit na následujícím odkazu [14]. Apache Maven usnadňuje vývoj aplikace a to zejména v řešení závislostí. Autor tedy nemusel stahovat potřebné knihovny ručně, ale jen definoval příslušné závislosti do pom.xml a dané knihovny se samy dotáhly. Základní struktura projektu je zobrazena na obrázku 12.



Obrázek 12 – Adresářová struktura projektu serverové části [vlastní zpracování]

Složka webapp je důležitou součástí projektu, neboť je základem pro každou webovou aplikaci. Obsahuje soubory, které jsou dostupné klientovi. Jako jsou například HTML soubory nebo obrázky. Dále také obsahuje podsložku WEB-INF, která má speciální význam. Tato složka není viditelná z webového prohlížeče klienta a tím pádem všechno, co je pod touto složkou, je klientovi skryto. Součástí složky WEB-INF je soubor web.xml (Deployment Descriptor), což je XML soubor, který slouží ke konfiguraci webové aplikace. Obsahuje informace popisující servlety, filtry, bezpečnostní pravidla, inicializační parametry a mnoho dalšího. Obsah souboru je znázorněn v kód 4.

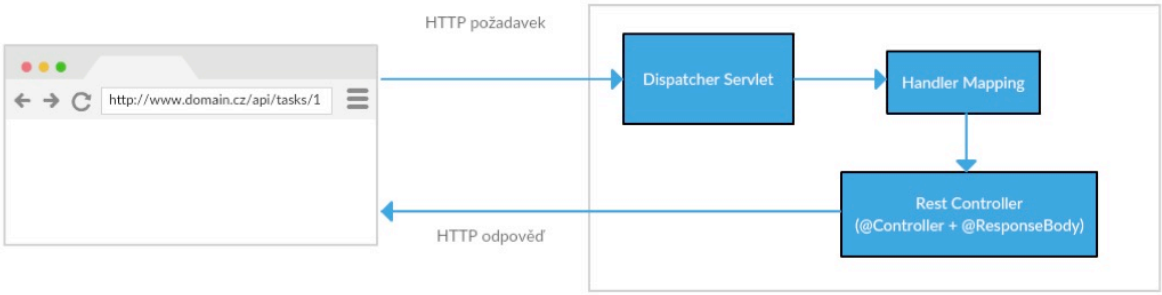
```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-
app_3_1.xsd"
         version="3.1">

    <display-name>TODO list web</display-name>

    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>

    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>WEB-INF/spring/application.xml</param-value>
    </context-param>

    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>WEB-INF/spring/application.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcherServlet</servlet-name>
        <url-pattern>/api/*</url-pattern>
    </servlet-mapping>

    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>
            org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
    </filter>

    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
        <dispatcher>ERROR</dispatcher>
        <dispatcher>REQUEST</dispatcher>
    </filter-mapping>

</web-app>
```

Kód 4 - Deployment Descriptor

Jak už bylo zmíněno, tak výsledná implementace aplikace je vyvíjena jako standardní webová aplikace. Při nasazení aplikace se načte soubor web.xml, který slouží k inicializaci aplikace a hlavně k inicializaci Springové kontextu. V konfiguračním

souboru web.xml se definuje tzv. listener se třídou ContextLoaderListener, který zajistí načtení hlavního konfiguračního souboru Spring, přičemž cesta k tomuto souboru je definována pomocí parametru, jenž se nazývá contextConfigLocatin. Požadovaná cesta se nachází v elementu param-value.

Pro manipulování se všemi HTTP požadavky a odpověďmi je využit Dispatcher Servlet, který je definován ve web.xml. Všechny požadavky, které mají být zpracovány pomocí Dispatcheru jsou mapovány k tomuto servletu pomocí elementu servlet-mapping. Když Dispatcher přijme HTTP požadavek, tak se poté poradí s HandlerMappingem jaký konkrétní Controller má zavolat. Controller poté vezme požadavek a zavolá odpovídající metodu, která provede na základě HTTP metody odpovídající obchodní logiku. Protože jsou implementovány REST služby, tak je využit RestController, který už poté jenom vrátí odpověď v těle zprávy. Funkčnost Dispatcher Servletu je znázorněna na obrázku 13. [15] [16]



Obrázek 13 – Funkce Dispatcher Servlet a Rest Controller [zpracováno podle [15]]

4.3.1.2 Datová vrstva

Téměř všechny aplikace potřebují někam ukládat svá data a následně je načítat zpět. K tomu se v dnešní době nejvíce používají databáze. V našem případě se jedná o komunikaci s databází MySQL. O komunikaci s databází se stará datová vrstva aplikace, která zajišťuje práci s daty. Programování databázové vrstvy jen s použitím základního JDBC je poměrně zdlouhavé a vyžaduje si vytvářet vlastní databázové spojení a s tím spojené další operace. V dnešní době už existují i jiná řešení, které nám usnadní práci s databází. V případě implementované aplikace byl zvolen pro komunikaci s databází tzv. JdbcTemplate od frameworku Spring, což je nadstavba nad standardní knihovnou v JDK, která slouží k přístupu do relační databáze.

Pro vytvoření spojení s databází je použito rozhraní JNDI pomocí frameworku Spring. Pro vytvoření DataSource v aplikaci je nutné nejprve definovat přístupové a další údaje k databázi na příslušném serveru. Pro dané řešení je využit Apache Tomcat server. V tomto případě je nutné do souboru context.xml, který se nachází na serveru ve složce conf vyplnit tzv. resource podle obrázku 14. Resource obsahuje základní konfigurační vlastnosti pro přístup k dané databázi. [17]

```
<Resource
  name="jdbc/todo"
  auth="Container"
  type="javax.sql.DataSource"
  maxActive="100"
  maxIdle="30"
  maxWait="10000"
  driverClassName="com.mysql.jdbc.Driver"
  url="jdbc:mysql://localhost:3306/todo"
  username="user"
  password="pass"
  />
```

Obrázek 14 – JNDI resource [vlastní zpracování]

Poté co se na serveru v souboru context.xml definuje konfigurace k databázi, pak v aplikaci už jen stačí inicializovat daný Datasource ve springovém kontextu pomocí jndi-lookup. Je potřebné jen zadat jdni-name, který je definován na obrázku 14 v atributu name. Když už je DataSource vytvořený je možné pomocí něho vytvořit JdbcTemplate, který nám umožní přístup do databáze. Výsledná implementace je znázorněna v kód 5. Všechny SQL dotazy do databáze se nacházejí v XML souboru, který se nazývá todo_list_queries.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/jee
       http://www.springframework.org/schema/jee/spring-jee.xsd">

    <!-- jndi data source -->
    <jee:jndi-lookup id="todoListDataSource" jndi-name="jdbc/todo"/>

    <!-- transaction manager -->
    <bean
id="todoListTransactionManager"class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="todoListDataSource"/>
    </bean>

    <!-- jdbc template -->
    <bean id="todoListJdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <constructor-arg type="javax.sql.DataSource" ref="todoListDataSource"/>
    </bean>

    <!-- list of queries into database -->
    <bean id="todoListQueryProps" class="org.springframework.beans.factory.config.PropertiesFactoryBean">
        <property name="locations" value="classpath:/todo_list_queries.xml"/>
    </bean>

</beans>
```

Kód 5 - Spring context pro vytvoření JdbcTemplate

Použití JdbcTemplate je už poté velmi jednoduché, což znázorňuje kód 6, kde se nachází metoda, která vrací konkrétní úkol podle id uživatele a id úkolu. Kód metody se vůbec nestará o konfiguraci připojení k databázi. Tuto práci za nás obstarává framework Spring pomocí již zmíněného JdbcTemplate.

```
public Task getTask(final Long idTask, final Long idUser) throws TodoListDaoException {
    try {
        final String sqlQuery = todoListQueryProps.getProperty("getTask");
        return this.todoListJdbcTemplate.queryForObject(sqlQuery, taskMapper, idTask, idUser);
    } catch (Exception e) {
        LOGGER.error("Error while loading task", e);
        throw new TodoListDaoException(e);
    }
}
```

Kód 6 – Metoda vracející úkol pomocí JdbcTemplate

Java Bean todoListQueryProps definovaná ve Spring contextu v kód 6, slouží k získání SQL dotazu, který se poté použije při zavolání metody queryForObject. Poslední bean, která nebyla vysvětlena je taskMapper, jenž odkazuje na instanci třídy TaskMapper, která implementuje Springové rozhraní RowMapper. Tato třída slouží k mapování výsledku daného SQL dotazu do instance konkrétní třídy. V tomto případě se jedná o třídu Task. Implementace třídy TaskMapper je znázorněna v kód 7.

```
@Component
public class TaskMapper implements RowMapper<Task> {
    @Override
    public Task mapRow(final ResultSet rs, final int rowNum) throws
SQLException {
        Task task = new Task();
        task.setIdTask(rs.getLong("id_task"));
        task.setName(rs.getString("name"));
        task.setDescription(rs.getString("description"));
        task.setUrgent(rs.getBoolean("urgent"));
        task.setTaskStatus(TaskStatusEnum.valueOf(rs.getString("status")));
        return task;
    }
}
```

Kód 7 – Mapování výsledku SQL dotazu do instance konkrétní třídy

Všechny metody, které přistupují k databázi přes JdbcTemplate jsou seskupeny v jednotlivých DAO třídách dle typu dat se kterými pracuje. DAO představuje rozhraní, které umožňuje přístup k databázi pomocí definovaných metod. Všechna logika, která se vztahuje k práci s databázemi, se tedy nachází v jednotlivých DAO třídách.

4.3.1.3 Aplikační vrstva

Aplikační vrstva využívá DAO pro přístup k datům, které pak transformuje, podle obchodní logiky a předává je prezentační vrstvě. K aplikaci obchodní logiky se používají tzv. service, což jsou rozhraní, které implementují potřebné metody. V této aplikaci se pro

přenos mezi aplikační a prezentační vrstvou využívá architektonického stylu REST. Implementace REST služby pomocí frameworku Spring je znázorněna v kód 8.

```
@RestController
@RequestMapping("/tasks")
public class TaskController {

    @Autowired
    private TaskService taskService;

@Secured(TodoConstants.ROLE_USER)
@RequestMapping(value = "/{id}", method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Task> getTask(@PathVariable("id") Long idTask) {
    final UserAuthentication activeUser = (UserAuthentication)
SecurityContextHolder.getContext().getAuthentication();
    if (activeUser != null) {
        try {
            final Task task = taskService.loadTask(activeUser.getName(), idTask);
            LOGGER.info("Loaded task with id: {}", idTask);
            return new ResponseEntity<>(task, HttpStatus.OK);
        } catch (TodoListException e) {
            LOGGER.error("Error while load task by given id: " + idTask, e);
            return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }

    return new ResponseEntity<>(HttpStatus.UNAUTHORIZED);
  }
}
```

Kód 8 – Implementace REST služby

Přístup k výše uvedené metodě dochází při provolání určitého HTTP požadavku, jenž je ovládaný pomocí Dispatcher Servlet. Ten rozhodne s pomocí HandlerMapping, který Controller má zavolat. Označení třídy jako Controller v implementované aplikaci dochází pomocí anotace @RestController, jenž dědí z @Controller a navíc přidává anotaci @ResponseBody, která zajistí, že návratové hodnoty třídních metod budou obsaženy v těle HTTP odpovědí. Anotace @Controller se obecně používá pro MVC komponenty a zajišťuje označení třídy jako Controller a rovněž ji zaregistruje do Spring contextu. Anotace @RequestMapping se používá pro mapování příchozích HTTP požadavků na konkrétní controller dle URI, HTTP metody, typu datové reprezentace média a očekávaných parametrů dané metody. V případě kód 8 se tedy při provolání URL adresy /tasks/{id} dostane požadavek do třídy TaskController, kde se poté provolá metoda getTask, která vrátí instanci třídy Task dle zadaného parametru idTask. Pro dynamické mapování parametrů z URI se používá anotace @PathVariable, která extrahuje id úkolu obsažené v URI adrese do proměnné idTask. Pomocí anotace @Autowired je možné injektovat dané třídě potřebné závislosti ze Springového kontejneru. [18]

4.3.1.4 Zabezpečení aplikace

Aplikace je zabezpečena pomocí frameworku Spring security, což je framework, který poskytuje jak autentizaci, tak oprávnění uživatele při provolávání RESTových služeb. Aby zabezpečení fungovalo je nutné do web.xml přidat bezpečnostní filtr, který se musí jmenovat springSecurityFilterChain. Implementace bezpečnostního filtru je znázorněna v kód 9. Tento filter odchytává všechny příchozí požadavky dle definované URL v elementu url-pattern a pomocí třídy DelegatingFilterProxy provolá konkrétní třídu, kde se nachází pravidla pro zabezpečení aplikace.

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>

<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>ERROR</dispatcher>
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

Kód 9 – Konfigurace Spring security ve web.xml

V kód 10 je znázorněna implementace zabezpečovací logiky aplikace, kde do třídy HttpSecurity se nastavují požadované vlastnosti zabezpečení. Hned první věcí, která se v této metodě nastavuje, je stavovost aplikace. Z daného řešení je zřejmé, že aplikace je bezstavová. Ve standardní webové aplikaci, autentizační proces probíhá tak, že když klient zkusí přistoupit na zabezpečenou URL a není ověřený, automaticky je přesměrován na přihlašovací formulář, kde se může znovu přihlásit. Nicméně pro aplikaci, která má poskytovat RESTové služby toto chování nedává moc smysl. Řešením je v tomto případě vracení HTTP kódu 401, který signalizuje klientské aplikaci, že daný uživatel nemá oprávnění. Toto je v kódu, řešeno pomocí metody unauthorizedEntryPoint, která vždy vrací HttpServletResponse s kódem 401. Pomocí metody antMatchers se definují přístupy k jednotlivým URL zdrojům. Z implementace je zřejmé, že je potřeba autentizace pouze u zdroje s URL začínající /api/. K ostatním zdrojům je možné přistupovat bez autentizace.

Jedinou výjimkou jsou ještě případy přihlášení a registrace, které jsou řešeny pomocí filtrů. Před každým požadavkem se prvně zavolají všechny definované filtry a až poté se ověřuje, jestli má na daný zdroj oprávnění. Stejným způsobem je řešen i přístup ke zdroji v případě již autentizovaného uživatele. Přihlášení do aplikace je implementováno pomocí filtru, který je mapovaný na URL adresu /api/login. Při provolání této adresy se k tomu

zavolá příslušný filtr, který se pokusí z těla requestu získat přihlašovací údaje, které jsou potom pomocí třídy CustomerUserDetailService ověřeny v databázi. Pokud ověření přihlašovacích údajů proběhne úspěšně, pak se vytvoří JSON Web Token, který se přidá do hlavičky odpovědi. [19]

```
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and()
                .exceptionHandling().authenticationEntryPoint(unauthorizedEntryPoint())
                .and()

                .exceptionHandling().and()
                .anonymous().and()
                .servletApi().and()
                .headers().cacheControl().and()
                .authorizeRequests()

                //allow anonymous resource requests
                .antMatchers("/").permitAll()
                .antMatchers("/api/**").authenticated()

                //allow anonymous POSTs to login
                .antMatchers(HttpMethod.POST, "/api/login").permitAll()
                .antMatchers(HttpMethod.POST, "/api/register").permitAll()
                .antMatchers(HttpMethod.POST, "/api/renew").authenticated()

                .anyRequest().permitAll().and()
                // custom JSON based authentication by POST of {"username":"<name>","password":"<password>"}
                which sets the token header upon authentication
                .addFilterBefore(new StatelessLoginFilter("/api/login", tokenAuthenticationService,
                authenticationManager()), UsernamePasswordAuthenticationFilter.class)

                .addFilterBefore(new StatelessRegisterFilter("/api/register", userService),
                 UsernamePasswordAuthenticationFilter.class)
                // stateless authentication filter for REST requests
                .addFilterBefore(new StatelessAuthenticationFilter(tokenAuthenticationService),
                 UsernamePasswordAuthenticationFilter.class)

                // filter for renewing token
                .addFilterBefore(new StatelessRenewFilter("/api/renew", tokenAuthenticationService),
                 UsernamePasswordAuthenticationFilter.class)

                .csrf().disable();
}
```

Kód 10 – Konfigurace zabezpečovací logiky

Logika, která ověřuje, že token zaslaný klientem je validní, se nachází ve třídě TokenAuthenticationService. Konkrétně to řeší metoda getAuthentication, která je znázorněna v kód 11. V prvé řadě se metoda zkusí vytáhnout z hlavičky requestu token. Pokud ho najde, tak poté pomocí externí knihovny JWT zkusí token dekódovat a ověřit jeho platnost. Jestliže všechno proběhne v pořádku, nastaví se daný uživatel do Spring security kontextu, ve kterém pak aplikace bude pracovat pro získání nebo zapsání požadovaných dat.

```
public Authentication getAuthentication(HttpServletRequest request) {

    final String token = request.getHeader(AUTH_HEADER_NAME);

    if (token == null) {
        return null;
    }

    Jwt jwt = JwtHelper.decodeAndVerify(token, getSigner());
    String json = jwt.getClaims();
    UserClaims userClaims = fromJSON(json);

    long timestamp = new Date().getTime() / 1000;
    if (userClaims.getExp() <= timestamp) {
        return null;
    }

    return new UserAuthentication(userClaims);
}
```

Kód 11 – Metoda pro získání autentizace uživatele

Jako datová reprezentace tokenu při přenosu mezi serverem a klientskou částí aplikace je využit JSON Web Token. Tato datová reprezentace se využívá k zaslání informací, které mohou být ověřené a důvěryhodné na základě digitálního podpisu. Přenášené informace jsou v JSON objektu, který je kryptograficky zakódován do výsledného tokenu a následně zaslán klientovi většinou pomocí HTTP hlavičky. Struktura JSON Web tokenu je reprezentována jako sekvence znaků, které jsou zakódované v base64url. Celý token se skládá ze 3 částí, které jsou rozdělené tečkou. Struktura tokenu, která je použita v implementované aplikaci je zobrazena na obrázku 15. Více informací o JSON Web Tokenu je možné se dozvědět z následujícího odkazu [20].

```
eyJhbGciOiJIUzI1NiJ9.
eyJleHAiOiE0Nia3OTE1NDksInVzZXJuYW1lIioiYWFhQGFhLmN6liwicm9sZXMiOlsiUk9MRV9BRE1JTilslJPTEVfVVNFUiJdfQ.
4wmOhUJHh9aAxRBfgbz2q52VMxpl4f38PeyZ8giUEgk
```

Obrázek 15 – Struktura JSON Web Tokenu [vlastní zpracování]

První část tokenu, tzv. hlavička obsahuje metadata o daném tokenu. Metadata většinou obsahují typ podpisu a šifrovací algoritmus. V našem případě hlavička deklaruje, že kódovaný objekt je JSON Web Token a že je podepsaný použitím šifrovacího algoritmu HMAC SHA-256.

```
{
    "alg": "HS256"
}
```

Obrázek 16 – Hlavička JSON Web Token [vlastní zpracování]

Prostřední část tokenu obsahuje tzv. payload, ve kterém se většinou nacházejí informace o identifikaci uživatele. Jako je například id uživatele, email a nebo navíc nějaká další metadata. JSON objekt, který je znázorněn na obrázku 17, obsahuje datum expirace tokenu, uživatelské jméno uživatele a kolekci jeho přístupových rolí.

```
{
    "exp": 1468791549,
    "username": "user@email.cz",
    "roles": [
    "ROLE_ADMIN",
    "ROLE_USER"
    ]
}
```

Obrázek 17 – Prostřední část JSON Web Token [vlastní zpracování]

Signature je poslední částí JSON Web Token, která je generována kombinací zakódované JWT hlavičky a zakódovaného JWT Payload, jenž je následně podepsána pomocí šifrovacího algoritmu, který je definován v hlavičce. V našem případě se jedná o HMAC SHA-256. Tajný klíč sloužící k podpisu je držen na serveru a tím je možné ověřit stávající přicházející tokeny, ale vytvořit i nové. [21]

```
HMACSHA256(
    base64UrlEncode(header) + "." +
    base64UrlEncode(payload),
    test
)
```

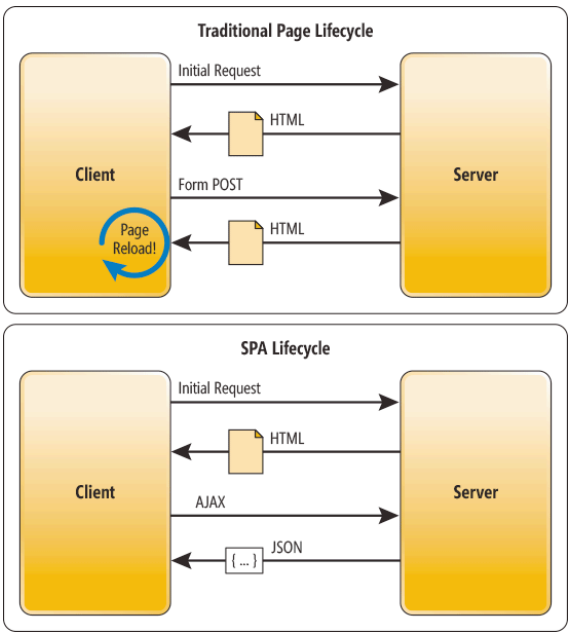
Obrázek 18 – Signature JSON Web Token [21]

4.3.2 Klientská část

Jelikož byla zvolena technologii REST, je možné implementovat klientskou část aplikace nad jakoukoliv platformou. Vývoj webových aplikací se neustále mění a jde kupředu. Poslední trendy současné doby jsou tzv. Single Page Application. Proto v daném řešení byl zvolen pro implementaci prezentační vrstvy aplikace framework AngularJS. Klasické webové aplikace, které mají více webových stránek fungují tak, že na serveru spojí HTML dokument s daty a až poté zašlou hotovou stránku prohlížeči. Ovšem AngularJS a ostatní frameworky podporující SPA naopak vezmou data ze serveru a obsah v šabloně se nahradí až v prohlížeči. V současné době je AngularJS nejspíše nejoblíbenější JavaScriptová knihovna pro vývoj Single Page aplikací. Je to otevřený framework, který je vyvíjen společností Google a je postaven na architektuře MVC/MVVM s podporou dependency injection a obousměrným data-binding umožňující vytvářet uživatelské rozhraní deklarativním způsobem. Jazyk pro deklaraci šablon uživatelského rozhraní je HTML, který je rozšířen o direktivy. Tato práce nemá za cíl popsat základní práci a technologie frameworku AngularJS. Pro více informací o tomto frameworku je možné se dozvědět na následujícím odkazu [22].

4.3.2.1 Single Page Application

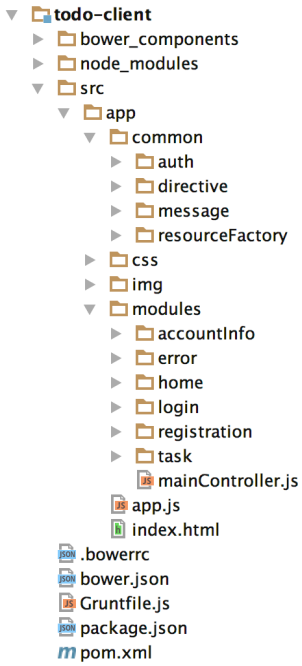
Single Page Application je typ webové aplikace, která má téměř veškerou funkcionalitu umístěnou na jedné stránce a server používají jen jako úložiště a zdroj dat. Při prvním přístupu na URL, kde se nachází SPA aplikace, se načte konkrétní soubor a k němu všechny další zdroje definované v hlavičce HTML dokumentu. Tyto zdroje obsahují většinou CSS soubory, JavaScript soubory atd. Všechna klientská logika se zpracovává na jedné stránce a když je potřeba se dotázat na něco serveru, pak se to děje pomocí AJAX. Tím je možné uživateli vyměnit část webové stránky, aniž bychom museli znovu načíst a vykreslit celý dokument. Hlavní výhodou SPA je rychlá odezva, neboť se ze serveru nestahují celé HTML stránky při každém požadavku, ale jen data. Prohlížeč pak tedy nemusí překreslovat vždy celou stránku. Porovnání chování a přenosu informací mezi serverem a klientem klasické webové aplikace a jednostránkové aplikace je znázorněn na obrázku 19. [23]



Obrázek 19 – Porovnání klasické a SPA webové aplikace [23]

4.3.2.2 Adresářová Struktura aplikace

Klientská část aplikace je začleněna jako samostatný Maven modul projektu. Součástí tohoto modulu jsou zdrojové kódy klientské aplikace v podobě HTML a JavaScriptových souborů, které se nacházejí pod složkou src. Obecné funkcionality a společné komponenty se nacházejí pod složkou common. Jsou to především různé typy vlastních implementací direktiv nebo služby sloužící pro zabezpečení aplikace, ale také továrny pro vytvoření REST endpointů. Nejdůležitějším souborem je index.html, neboť tento soubor se načte při vstupu na konkrétní URL, kde se nachází SPA aplikace. Tento soubor, jak už bylo řečeno, obsahuje všechny potřebné zdroje. Ve složce modules se nacházejí jednotlivé části aplikace, které spolu souvisí a tvoří funkční celky. Každý tento modul obsahuje HTML šablonu a controller. Controller slouží pro práci a prezentaci dat získaných typicky přes REST služby. Controller tvoří datový kontrakt mezi serverovou a klientskou částí aplikace, kterému říkáme model. Model se ukládá na $scope controlleru a doplňuje se o další atributy, jako jsou například pomocné proměnné pro vykreslení uživatelského rozhraní. Model by měl odpovídat požadavkům controlleru, příslušné šabloně a neměl by obsahovat zbytečné, či nepotřebné informace.



Obrázek 20 – Struktura klientské části aplikace [vlastní zpracování]

4.3.2.3 Bower

Bower slouží k instalaci správných verzí frontend balíčků a jejich vzájemných závislostí, které jsou potřebné pro vývoj aplikace. Pro instalaci konkrétního balíčku stačí zadat pouze do příkazové řádky bower install angular, čímž se začne stahovat z repozitáře balíček angular, který se uloží do složky /bower_components. Vlastnosti bower lze nastavit v konfiguračním souboru, jenž se nazývá bowerrc.json. Tento soubor obsahuje konfigurační proměnné pro Bower. Například proměnná directory určuje cestu, kam se nainstalují balíčky. Více informací o konfiguračních proměnných lze získat na následujícím odkazu [24]. Manifest soubor bower.json slouží k uchování jednotlivých názvů balíčků a jejich verzí. Obsah tohoto souboru je blíže popsán v následujícím zdroji [25]. Soubor bower.json vyvíjené aplikace je zobrazen v kód 12, kde je možno vidět jaké balíčky jsou použity pro vývoj klientské části aplikace. [26]

```
{
  "name": "todo-list",
  "description": "Todo list",
  "version": "0.0.1",
  "authors": [
    "Tomas Holicky <holickytomas@gmail.cz>"
  ],
  "license": "Tomas Holicky",
  "private": true,
  "dependencies": {
    "angular": "1.4.x",
    "angular-mocks": "1.4.x",
    "jquery": "~2.1.1",
    "bootstrap": "~3.3.6",
    "angular-resource": "1.4.x",
    "angular-cookies": "~1.4.x",
    "angular-animate": "1.4.x",
    "angular-aria": "1.4.x",
    "angular-ui-router": "^0.2.10",
    "font-awesome": "~4.1.0",
    "angular-bootstrap": "~1.0.3",
    "angular-ui-tree": "2.x",
    "angular-webstorage": "0.11.*",
    "angular-jwt": "0.0.*",
    "angular-messages": "1.4.x",
    "angular-i18n": "1.4.x",
    "angular-ui-notification": "0.2.0",
    "ng-file-upload": "12.2.12"
  },
  "devDependencies": {
    "angular-mocks": "1.4.x",
    "angular-scenario": "1.4.x"
  },
  "resolutions": {
    "bootstrap": "~3.3.6",
    "angular": "~1.4.x"
  },
  "appPath": "app"
}
```

Kód 12 – Obsah souboru bower.json

4.3.2.4 NPM

NPM je nejpoužívanější balíčkový systém, který je součástí instalace Node.js. NPM slouží především ke sdílení kódu, který je poté možné lehce znovupoužít. S NPM se pracuje podobně jako s bower, tedy přes příkazový řádek, pomocí něhož se zadá jednoduchý příkaz npm install a název balíčku, který se chce nainstalovat.

Soubor package.json je strukturován jako JSON objekt a měl by ho obsahovat každý projekt, neboť obsahuje základní informace o projektu. Důležitou částí je zejména část dependencies, jenž obsahuje seznam balíčků na kterých je daná aplikace závislá a to včetně verze. V části devdependencies jsou moduly, které slouží při vývoji daného balíčku. Více informací o NPM je možné se dozvědět na následujícím odkazu [27].

4.3.2.5 Grunt

Grunt je jen skript, který slouží webovým vývojářům pro automatizaci opakujících se úkolů. Typickým příkladem je minifikace CSS nebo JavaScript souborů nebo jejich spojování do jednoho souboru. Pro práci s Gruntem je potřeba mít nainstalovaný Node.js a NPM. Pomocí NPM se nainstaluje jednoduše rozhraní Gruntu pro příkazovou řádku a poté se nainstalují potřebné Grunt pluginy z kterých se pak vytvářejí jednotlivé úkoly. Názvy a verze Grunt pluginů jsou definovány v package.json v sekci devdependencies, o kterém byla řeč v předcházející kapitole. Výsledný skript s jednotlivými úkoly je psán do souboru Gruntfile.js. V kód 13 je znázorněn jeden z úkolů, který spojuje jednotlivé JavaScriptové knihovny do jedné pomocí pluginu grunt-contrib-concat.

```
concat: {
    options: {
        separator: '\n'
    },
    lib: {
        src: [
            // Jquery
            '<%=cfg.bower%>/jquery/dist/jquery.min.js',
            '<%=cfg.bower%>/jquery-ui/ui/jquery-ui.js',

            // Angular
            '<%=cfg.bower%>/angular/angular.js',
            '<%=cfg.bower%>/angular-bootstrap/ui-bootstrap-tpls.js',
            '<%=cfg.bower%>/angular-cookies/angular-cookies.min.js',
            '<%=cfg.bower%>/angular-ui-router/release/angular-ui-router.min.js',
            '<%=cfg.bower%>/angular-ui-tree/dist/angular-ui-tree.js',
            '<%=cfg.bower%>/angular-aria/angular-aria.js',
            '<%=cfg.bower%>/angular-animate/angular-animate.min.js',
            '<%=cfg.bower%>/angular-resource/angular-resource.min.js',
            '<%=cfg.bower%>/bootstrap/dist/js/bootstrap.min.js',
            '<%=cfg.bower%>/angular-jwt/dist/angular-jwt.min.js',
            '<%=cfg.bower%>/angular-webstorage/angular-webstorage.min.js',
            '<%=cfg.bower%>/angular-messages/angular-messages.min.js',
            '<%=cfg.bower%>/angular-ui-notification/dist/angular-ui-notification.min.js',
            '<%=cfg.bower%>/ng-file-upload/ng-file-upload.min.js',

            // Languages
            '<%=cfg.bower%>/angular-i18n/angular-locale_cs.js'
        ],
        dest: '<%=cfg.dist%>/js/lib.js'
    }
}
```

Kód 13 – Grunt úkol spojující JavaScriptové knihovny do jednoho souboru

Po vytvoření potřebných úkolů je bude možné spustit pomocí příkazového řádku zadáním příkazu grunt a název úkolu. Úkoly je také možné seskupovat do různých funkčních celků, aby se nemusel každý úkol spouštět samostatně. Jednotlivé názvy úkolů jsou zaznamenány v poli, jak je vidět z kód 14. Také je možné do pole s úkoly zadávat i už seskupené úkoly, což v našem případě jsou seskupené úkoly s názvem javascript. Pro statickou kontrolu JavaScript kódu se používá JSHint, který zajišťuje kvalitu aplikace a je spouštěn pomocí Gruntu. [28]

```
grunt.registerTask('javascript', ['jshint:sources', 'ngAnnotate:build', 'concat']);
grunt.registerTask('default', ['jshint:grunt', 'clean', 'javascript', 'ngtemplates', 'copy:assets']);
```

Kód 14 – Seskupené Grunt úkoly

4.3.2.6 Základ aplikace

Základ aplikace tvoří dva soubory, které se nazývají index.html a app.js. První zmíněný soubor je výchozí HTML soubor, který se načte v prohlížeči, když uživatel přistoupí na konkrétní URL adresu, kde se daná aplikace nachází. V daném HTML souboru v sekci <HEAD> se nachází další odkazy, které načtou CSS styly, JavaScriptové knihovny, HTML šablony a také JavaScriptové soubory aplikace. Všechny tyto potřebné soubory jsou pomocí Gruntu spojeny vždy do jednoho výsledného souboru. Jak je vidět z kód 15, kde soubor lib.css obsahuje všechny potřebné CSS styly, které se používají

v aplikaci. Soubor lib.js obsahuje JavaScriptové knihovny. V souboru template.js se nacházejí všechny HTML šablony aplikace, které používá AngularJS. Tento soubor je vytvořen pomocí balíčku grunt-angular-templates, který vytvoří standardní modul pomocí AngularJS, kde jednotlivé HTML šablony jsou spravovány pomocí templateCache. Poslední odkazovaný soubor je application.js, což jsou spojené JavaScriptové soubory implementované aplikace. Výhodu spojení jednotlivých souborů je, že se nemusí v sekci <HEAD> všechny potřebné soubory odkazovat. Každý JavaScript kód implementované aplikace je interpretován v tzv.Strict módu, což je funkce, která lépe detekuje chyby a tím pádem se předchází chybám po překladu aplikace.

```
<!-- STYLES -->
<link rel="stylesheet" type="text/css" href="lib.css">

<!-- JAVASCRIPT -->
<script type="text/javascript" src="js/lib.js"></script>
<script type="text/javascript" src="js/templates.js"></script>
<script type="text/javascript" src="js/application.js"></script>
```

Kód 15 – Odkazy v sekci <HEAD>

Důležitou částí index.html souboru je direktiva ng-app, která inicializuje AngularJS aplikaci a automaticky jí spustí. Tato direktiva očekává parametr, jenž určuje hlavní modul aplikace. Tento modul je pak použit jako kořenový modul aplikace. AngularJS aplikace zpracovává jen DOM elementy a jeho potomky u kterých byla definována tato direktiva. Každá HTML stránka může obsahovat ng-app pouze jednou.

```
<html lang="cs" ng-app="cz.toho.todoClient">
```

Kód 16 – Použití direktivy ng-app

Root modul se nachází již v zmíněném app.js souboru. Modul se v AngularJS definuje velmi snadno. Například jako kořenový modul definovaný v app.js, který je znázorněn v kód 17. Parametry modulu jsou název modulu a pole závislých modulů. Moduly začínající na cz.toho jsou vlastní implementace, ostatní jsou externí moduly.

```
var todoClient = angular.module('cz.toho.todoClient', [
    'ui-notification',
    'ui.router',
    'cz.toho.taskResourceFactory',
    'cz.toho.userResourceFactory',
    'cz.toho.auth.directives',
    'cz.toho.common.directives',
    'cz.toho.main',
    'cz.toho.templates',
    'cz.toho.home',
    'cz.toho.task',
    'cz.toho.login',
    'cz.toho.accountInfo',
    'cz.toho.registration',
    'ngResource',
    'ngMessages',
    'cz.toho.auth.stateRouter'
])
```

Kód 17 – Definování modulu v AngularJS

Důležitý modul je ui.router, který má na starosti routování jednotlivých stránek aplikace. Je to hierarchický strom jednotlivých stavů, které při průchodu aplikací mohou nastat. Pokud stav není definován, aplikace je přesměrována na definovanou defaultní adresu. Každý modul může mít konfigurační část, která se provede při inicializaci modulu. V této části jsou definovány konstanty nebo providery. Další částí, kterou modul může mít, je tzv. run block, který obsahuje potřebné věci ke spuštění aplikace.

4.3.2.7 Direktivy

V této kapitole bude řeč především o vlastních implementacích direktiv. Direktivy nám umožňují obohatit standardní HTML o nové funkčnosti. V kód 18 je zobrazena direktiva, která se stará o zobrazení profilu uživatele.

```
.directive('profile', function (authService) {
    return {
        scope: {},
        restrict: 'E',
        link: function (scope) {
            scope.logout = function () {
                authService.logout();
            };
        },
        templateUrl: 'common/directive/template/profile.html'
    };
}
```

Kód 18 – Direktiva profil uživatele

Daná funkce vrací konfigurační objekt direktivy, který obsahuje několik vlastností. Vlastnost scope je kontext direktivy, kde je možné mít definované pomocné proměnné, které předávají data šabloně. Způsob použití direktivy je dáno vlastností restrict. Hodnoty této vlastnosti mohou nabývat hodnot viz tabulka 28.

Restrict	Popis	Použití v HTML šabloně
E	Název elementu	`<profile></profile>`
A	Atribut	`<div profile></div>`
C	Třída	`<div class="profile"></div>`
M	Komentář	`<!-- directive: profile -->`

Tabulka 28 – Vlastnost restrict [zpracováno podle [29]]

Vlastnost link slouží k definování funkce, která se provede po transformaci DOM. Pokud se uživatel bude chtít odhlásit, tak klikne na tlačítko odhlásit a pomocí direktivy ng-click se zavolá tato funkce, která pomocí služby authService odhlásí uživatele. Vlastnost templateUrl odkazuje na HTML šablonu, za kterou se direktiva při použití zamění. [29]

```
<ul class="nav navbar-nav navbar-right">
    <li class="dropdown">
        <a href class="dropdown-toggle" data-toggle="modal" data-target="#addTaskModal">
            <i class="fa fa-plus"></i>
        </a>
    </li>
    <li class="dropdown">
        <a href class="dropdown-toggle" data-toggle="dropdown">
            <auth-user-name/>
            <b class="caret"></b>
        </a>
        <ul class="dropdown-menu">
            <li>
                <a ui-sref="user.tasks"><i class="fa fa-fw fa-tasks"></i> Úkoly</a>
            </li>
            <li>
                <a ui-sref="accountInfo"><i class="fa fa-fw fa-user"></i> Profil</a>
            </li>
            <li class="divider"></li>
            <li>
                <a href ng-click="logout()"><i class="fa fa-fw fa-power-off"></i> Odhlásit se</a>
            </li>
        </ul>
    </li>
</ul>
```

Kód 19 – HTML šablona pro direktivu profile

4.3.2.8 Volání REST služeb

V konceptu REST API je potřeba z klientské části aplikace volat mnoho různých typů RESTových služeb. V AngularJS aplikaci je to možné volat dvěma způsoby. Prvním řešením je $http služba. V kód 20 je zobrazena obecná syntaxe pro znázornění, jak daná služba funguje. Pomocí vlastnosti method se definuje typ HTTP operace, která se bude volat na danou URL adresu, jenž je definována pomocí vlastnosti url. Výsledná odpověď, což je v našem případě objekt response obsahuje:

- data – tělo odpovědi,
- status – HTTP status kód,
- headers – obsah HTTP hlavičky,
- config – konfigurační objekt, který byl použit ke generování požadavku,
- statusText – text HTTP statusu kódu.

```
$http({
    method: 'GET',
    url: '/someUrl'
}).then(function successCallback(response) {
    // success callback
}, function errorCallback(response) {
    // failure call back
});
```

Kód 20 – Obecná syntaxe $http služby [30]

Druhým řešením pro volání REST API je služba $resource, která je použita v implementované aplikaci. Pomocí AngularJS factory se vytvoří tzv.resource objekty které je poté možno použít v jednotlivých controllerech aplikace. V kód 21 je znázorněn jeden z resource objektů, který obsahuje metody pro smazání a načtení úkolu podle daného id úkolu. Povinný parametr $resource služby je URL adresa, která je složena z API_URI a konkrétní URL adresy. API_URI je jedna z konstant aplikace, jenž obsahuje základní URL adresu REST API.

```
TaskId: $resource(API_URI + '/tasks/:id', {}, {
    delete: {
        method: 'DELETE',
        params: {
            id: 'idTask'
        }
    },
    get: {
        method: 'GET',
        isArray: false,
        params: {
            id: 'id'
        }
    }
})
```

Kód 21 – Resource objekt pro provolání konkrétní REST služby

Použití resource objektu je znázorněno v kód 22, přičemž ApiTask je název AngularJS factory, která má na starosti vytváření resource objektů. TaskId už je konkrétní resource objekt, jenž volá delete metodu s daným id úkolu.

```
ApiTask.TaskId.delete({id: idTask});
```

Kód 22 – Volání REST zprávy pomocí resource objektu

Důvody pro použití $resource oproti $http jsou především ve snadnějším a přehlednějším použití, neboť v případě $http služby je nutné vždy definovat znova funkci a všechny parametry volání. To u $resource služby odpadá, neboť se všechny potřebné věci definují v AngularJS factory, které už se potom jenom znovu používají. Pro použití této služby je nutné mít v závislostech aplikace modul ngResource. [31]

4.3.2.9 Zabezpečení aplikace

Zabezpečení aplikace je řešeno pomocí tokenu, který je po úspěšném přihlášení uživatele uložen do tzv. WebStorage, jenž slouží k ukládání dat na straně klienta. Existují dva druhy lokálních uložišť, které se liší pouze perzistencí. Uložiště, které ukládá data v prohlížeči napořád, tedy pokud nejsou smazána, se nazývá LocalStorage a druhý typ uložiště se nazývá SessionStorage, jenž ukládá data jen po dobu trvání sezení (session), což znamená jen po dobu, dokud uživatel neuzavře prohlížeč, nebo záložku s danou stránkou. V našem případě je token uložen do SessionStorage, neboť to je bezpečnější. V kód 23 je zobrazena metoda, která se stará o uložení tokenu do WebStorage a nastavení uživatele do kontextu aplikace. Práce s WebStorage funguje pomocí modulu angular-webstorage, pomocí něhož se token jednoduše uloží. Lokální uložiště fungují na principu asociativního pole. K hodnotám se tedy přistupuje pomocí unikátního klíče, který je reprezentován textovým řetězcem. Pro práci s tokenem v implementované aplikaci se používá klíč cz.toho.auth:token. Jak už bylo zmíněno, tak se využívá JWT token, tudíž je možné z něho získat nějaké informace o uživateli. Pro získání těchto údajů se využívá knihovna angular-jwt.

```
this.setToken = function (token) {
    if (!token) {
        return;
    }

    webStorage.session.add('cz.toho.auth:token', token);

    this.info.token = token;
    this.info.user = jwtHelper.decodeToken(token);
    this.info.expires = jwtHelper.getTokenExpirationDate(token);

    $rootScope.$broadcast(EVENTS.changedState);
};
```

Kód 23 – Metoda pro uložení autorizačního tokenu

Pro autorizaci na straně serveru je nutné pokaždé zasílat token v hlavičce HTTP požadavku. K tomu slouží tzv. Interceptor, který token při každém požadavku na server přidá do HTTP hlavičky. Implementace dané služby je znázorněna v kód 24.

```
function tokenInterceptor(authTokenStore, authServiceSettings) {
    return {
        request: function (config) {
            config.headers = config.headers || {};
            if (authTokenStore.info.token) {
                config.headers[authServiceSettings.tokenHeaderName] = authTokenStore.info.token;
            }
            return config;
        }
    };
}
```

Kód 24 – Token interceptor

Interceptors jsou jen služby, které jsou registrovány pomocí $httpprovider a jsou volány při libovolném HTTP požadavku. Dalším použitím interceptor je pro získání informace o chybě z odpovědi serveru, což je zobrazeno v kód 25. Daný interceptor zachytává určité chybové HTTP stavy, které jsou důležité pro následné chování aplikace. Pokud tedy například nastane případ, kdy odpověď ze serveru přijde s HTTP status kódem 401, což znamená, že uživatel nebyl autorizován, tak bude uživatel přesměrován na přihlašovací obrazovku a uživatelské informace budou odstraněny z kontextu aplikace. Ještě existují dva interceptory, které nebyly představeny. Jeden slouží k manipulaci chyb, které nastaly při vytváření požadavku na server a jmenuje se requestError a druhý je pro zpracování úspěšné odpovědi a nazývá se response. [30]

```
function responseErrorInterceptor($injector) {
    return {
        responseError: function (response) {
            return $injector.invoke(function (authServiceSettings, authService, $q, $state) {
                if (response.status === 401) {

                    if (response.config.url === authServiceSettings.urls.login) {
                        return $q.reject(response);
                    }
                    $state.go('anon.login');
                    authService.clearAuth();
                    return $q.reject(response);
                } else if (response.status === 403) {
                    $state.go('anon.error.403');
                    return $q.reject(response);
                } else if (response.status === 500) {
                    $state.go('anon.error.500');
                    return $q.reject(response);
                }
                return $q.reject(response);
            });
        }
    };
}
```

Kód 25 – Interceptor pro ovládání chybových odpovědí ze serveru

Řízení přístupů na jednotlivé stránky aplikace funguje pomocí knihovny ui.router, která se stará o routování jednotlivých stránek aplikace. Je to hierarchický strom jednotlivých stavů, které při průchodu aplikací mohou nastat. Pokud stav není definován, pak je aplikace přesměrována na definovanou defaultní adresu. Pomocí služby $stateProvider se definují jednotlivé stavy aplikace neboli obrazovky. Pro zabezpečení jednotlivých obrazovek se využívá uživatelských rolí. V kód 26 jsou definovány stavy, které vyžadují mít uživatelskou roli ROLE_USER.

```
$stateProvider.state('user', {
    abstract: true,
    template: "<ui-view/>",
    data: {
        access: 'access.user',
        authRoles: ['ROLE_USER']
    }
}).state('user.tasks', {
    url: '/tasks',
    controllerAs: 'taskListCtrl',
    controller: 'taskListController',
    templateUrl: 'modules/task/taskList.html'
}).state('user.tasks.detail', {
    url: '^/tasks/:id',
    views: {
        '@': {
            templateUrl: 'modules/task/detail/taskDetail.html',
            controller: 'taskDetailController',
            controllerAs: 'taskDetailCtrl'
        }
    }
});
```

Kód 26 – Zabezpečené routování aplikace

Stav user, definuje obecné vlastnosti, které pak dědí ostatní stavy. Vlastnost authRoles definuje role, které uživatel musí mít, aby mohl přejít na danou stránku. Pokud tedy uživatel bude chtít přistoupit na stránku s URL adresou /tasks bude muset mít uživatelskou roli ROLE_USER.

Při každé změně stavu se vytvoří tzv. broadcast s název $stateChangeStart, což znamená, že se vyšle zpráva do aplikace, že došlo ke změně stavu. Tato zpráva se zachytí a spustí se metoda, která zkontroluje oprávnění uživatele viz kód 27. To znamená, že se zkontroluje, jestli uživatel má potřebnou uživatelskou roli pro přístup na danou obrazovku. Pokud nemá, tak bude přesměrován na přihlašovací formulář. [32]

```
this.checkPermissionWhenStateChangeStarted = function (event, toState) {

    if (this.isPublicVisible(toState)) {
        return;
    }

    if (authService.isAuthenticated()) {
        this.checkPermissionsAndBroadcastIfError(toState, event);
    } else {
        event.preventDefault(); // stop routing
        $state.go('anon.login');
    }
};
```

Kód 27 – Metoda kontrolující oprávnění pro vstup na danou obrazovku

5 Výsledky a diskuse

5.1 Potenciální podněty pro vylepšení aplikace

Jelikož se jedná o prototyp aplikace, tak z hlediska funkční logiky byly implementovány jen základní požadavky. Funkčnost aplikace by šla samozřejmě rozšířit o mnoho dalších věcí. Jimiž například jsou časové připomínky (integrace s kalendářem), vytváření podúkolů, seskupování úkolů do různých kategorií, které si sám uživatel definuje atd.

Z hlediska technologického by RESTové rozhraní mohlo lépe umožňovat konzumaci RESTových služeb vývojářům třetích stran, neboť v současném řešení je RESTové rozhraní přizpůsobeno potřebám implementované klientské aplikace. Znamenalo by to vytvoření registračního procesu pro vývojáře třetích stran, který by vygeneroval unikátní klíč, jenž by používali v případě volání RESTových služeb. Tím pádem by pak bylo vždy zřejmé, jaká klientská aplikace provolala danou webovou službu. Stávající řešení by muselo být implementováno více robustněji a to především z hlediska zabezpečení aplikace, neboť to ve stávající implementaci funguje jen pomocí zabezpečovacího tokenu, pomocí něhož se každý klient autentizuje. Pro autorizaci přístupu k REST API by muselo být implementováno řešení na principu OAuth 2.0. Více o tomto principu je možné se dozvědět na následujícím odkazu [33].

V současném řešení chybí i lepší dokumentace, která by vývojářům třetích stran ulehčila vývoj jejich klientské aplikace a především pomohla pochopit význam jednotlivých RESTových služeb.

6 Závěr

Dílčím cílem této diplomové práce bylo přiblížit servisně orientovanou architekturu a webové služby. Hlavním cílem práce byla praktická implementace webové aplikace, která využívá webové služby.

V teoretické části diplomové práce byl přiblížen koncept servisně orientované architektury, který je možno realizovat pomocí webových služeb. Dále byly představeny dostupné architektury webových služeb, jimiž jsou RPC, SOAP a poslední dobou velmi se rozšiřující REST. Krátce byly popsány datové formáty, které používají webové služby pro přenos dat mezi serverem a klientem.

Praktická část diplomové práce se zabývá celým procesem vývoje webové aplikace, která ke svému běhu využívá webové služby. Konkrétně se jedná o aplikaci pro správu úkolů. Prvním krokem při vývoji webové aplikace bylo definování požadavků na aplikaci. Všechny funkční požadavky byly sepsány ve formě případů užití, které jsou zobrazeny na diagramu 1. Na základě těchto případů užití byla vytvořena analýza a následný návrh výsledné aplikace. Byl navrhnut datový model, jenž popisuje základní entity, které se v aplikaci nacházejí. Pro snadnou představu o fungování a rozložení prvků na obrazovce byly realizovány tzv. wireframe, jenž jsou blíže popsány v kapitole 4.2.3. Jelikož výsledkem práce měla být aplikace, která využívá webové služby, tak se pro komunikaci se serverem, navrhlo REST API. Na základě návrhu byla implementována webová aplikace. Implementace se rozdělila na dvě části. V první fázi vývoje byla implementována serverová část aplikace v podobě RESTových služeb. Popis serverové části aplikace je popsán v kapitole 4.3.1. Druhou částí implementace bylo vytvoření klientské části aplikace, která je konzumentem zmíněných webových služeb. Klientská část aplikace se implementovala jako tzv. Single Page Application pomocí frameworku AngularJS. Implementace klientské části aplikace je popsána v kapitole 4.3.2.

Výsledkem diplomové práce je tedy funkční webová aplikace pro správu úkolů, jenž je obsahem digitální přílohy. Ukázky hlavních obrazovek výsledné aplikace se nacházejí v příloze. Potenciální podněty pro vylepšení aplikace jsou zmíněny v kapitole 5.1.

7 Seznam použitých zdrojů

1. Erl, Thomas. *SOA: servisně orientovaná architektura : kompletní průvodce.* Brno : Computer Press, 2009. 978-80-251-1886-3.

2. Kodali, By Raghu R. What is service-oriented architecture? *JavaWorld.* [Online] [Citace: 5. 6 2016.] http://www.javaworld.com/article/2071889/soa/what-is-service-oriented-architecture.html.

3. DAIGNEAU, Robert. *Service design patterns: fundamental design solutions for SOAP/WSDL and RESTful Web services.* Upper Saddle River : Addison-Wesley signature series, 2012. 032154420X.

4. Englander, Robert. *Java and SOAP.* Sebastopol : O'Reilly, 2002. 0596001754.

5. Kosek, Jiří. Využití webových služeb a protokolu SOAP při komunikaci. [Online] [Citace: 6. 5 2016.] http://www.kosek.cz/diplomka/html/websluzby.html.

6. WSDL GetWeather. [Online] [Citace: 5. 6 2016.]. http://www.webservicex.net/globalweather.asmx?wsdl.

7. Burke, Bill. *RESTful Java with JAX-RS 2.0.* Second edition. Beijing : O'Reilly, 2013. 144936134X.

8. Berners-Lee, T. Uniform Resource Identifier (URI): Generic Syntax. [Online] [Citace: 5. 8 2016.] https://tools.ietf.org/html/rfc3986.

9. World Wide Web Consortium (W3C). Method Definitions. [Online] [Citace: 8. 6 2016.] https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

10. Mark Nottingham, Julian Reschke, Jan Algermissen. Link relations. [Online] [Citace: 8. 8 2016.] http://www.iana.org/assignments/link-relations/link-relations.xml.

11. World Wide Web Consortium (W3C). Status Code Definitions. [Online] [Citace: 8. 6 2016.] https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

12. Neustadt, Jim Arlow a Ila. *UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky.* 2, Brno : Computer Press, 2007. 978-80-251-1503-9.

13. Software, Pivotal. Learning Spring. [Online] [Citace: 25. 8 2016.] https://spring.io/docs.

14. Foundation, The Apache Software. Maven Getting Started Guide. [Online] [Citace: 23. 7 2016.] https://maven.apache.org/guides/getting-started/index.html.

15. Sundararajan, Srivatsan. Spring MVC Framework and REST. [Online] [Citace: 6. 8 2016.] https://www.genuitec.com/spring-frameworkrestcontroller-vs-controller/.
16. Pivotal. Web MVC framework. [Online] [Citace: 6. 8 2016.] http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/mvc.html.
17. Foundation, Apache Software. JNDI Resources HOW-TO. [Online] [Citace: 6. 8 2016.] https://tomcat.apache.org/tomcat-7.0-doc/jndi-resources-howto.html.
18. WebSystique. Spring MVC 4 RESTFul Web Services CRUD Example+RestTemplate. [Online] [Citace: 6. 8 2016.] http://websystique.com/springmvc/spring-mvc-4-restful-web-services-crud-example-resttemplate/.
19. Alex, Ben, a další. Spring Security Reference. [Online] [Citace: 6. 9 2016.] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/.
20. JWT. Introduction to JSON Web Tokens. [Online] [Citace: 6. 8 2016.] https://jwt.io/introduction/.
21. Tkalec, Tino. JSON Web Token Tutorial: An Example in Laravel and AngularJS. [Online] [Citace: 6. 8 2016.] https://www.toptal.com/web/cookie-free-authentication-with-json-web-tokens-an-example-in-laravel-and-angularjs.
22. Google. AngularJS. [Online] [Citace: 20. 6 2016.] https://angularjs.org/.
23. Wasson, Mike. Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET. [Online] [Citace: 25. 7 2016.] https://msdn.microsoft.com/en-us/magazine/dn463786.aspx.
24. Mast, Isaac. Bower configuration. [Online] [Citace: 21. 5 2016.] https://github.com/bower/spec/blob/master/config.md.
25. Guarnerio, Matteo. Bower.json specification. [Online] [Citace: 21. 5 2016.] https://github.com/bower/spec/blob/master/json.md.
26. Bower. Bower A package manager for the web. [Online] [Citace: 8. 8 2016.] https://bower.io/.
27. npm, Inc. Npm docs. [Online] [Citace: 20. 7 2016.] https://docs.npmjs.com/.
28. Grunt. Getting started. [Online] [Citace: 21. 7 2016.] http://gruntjs.com/getting-started.

29. Google. Creating Custom Directives. [Online] [Citace: 9. 7 2016.]
https://docs.angularjs.org/guide/directive.
30. Google. $http. [Online] [Citace: 21. 7 2016.]
https://docs.angularjs.org/api/ng/service/$http.
31. Google. $resource. [Online] [Citace: 21. 7 2016.]
https://docs.angularjs.org/api/ngResource/service/$resource.
32. AngularUI. Angular-ui/ui-router. [Online] [Citace: 22. 7 2016.]
https://github.com/angular-ui/ui-router/wiki.
33. D. Hardt, Ed. The Oauth 2.0 Authorization Framework. [Online] [Citace: 25. 10 2016.]
https://tools.ietf.org/html/rfc6749.

8 Přílohy

8.1 Obsah digitální přílohy

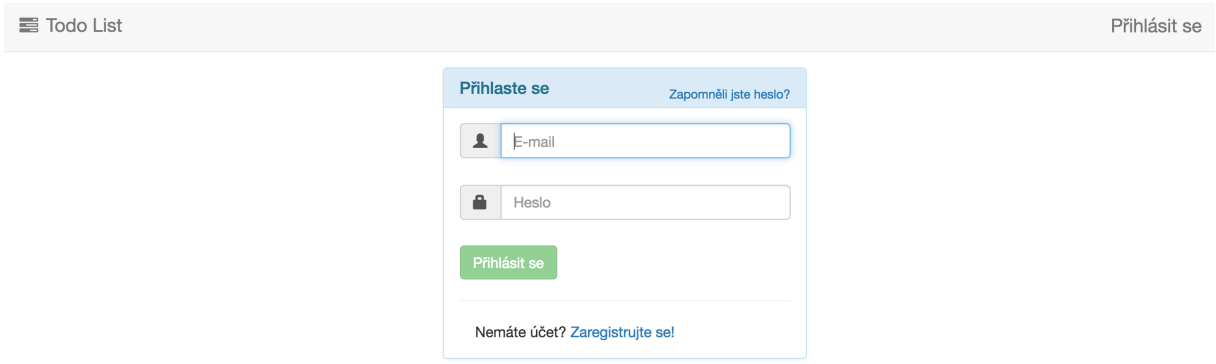
Digitální příloha obsahuje výslednou implementaci webové aplikace, DDL skripty pro vytvoření struktury databáze a DML skripty.

8.2 Seznam zkratek

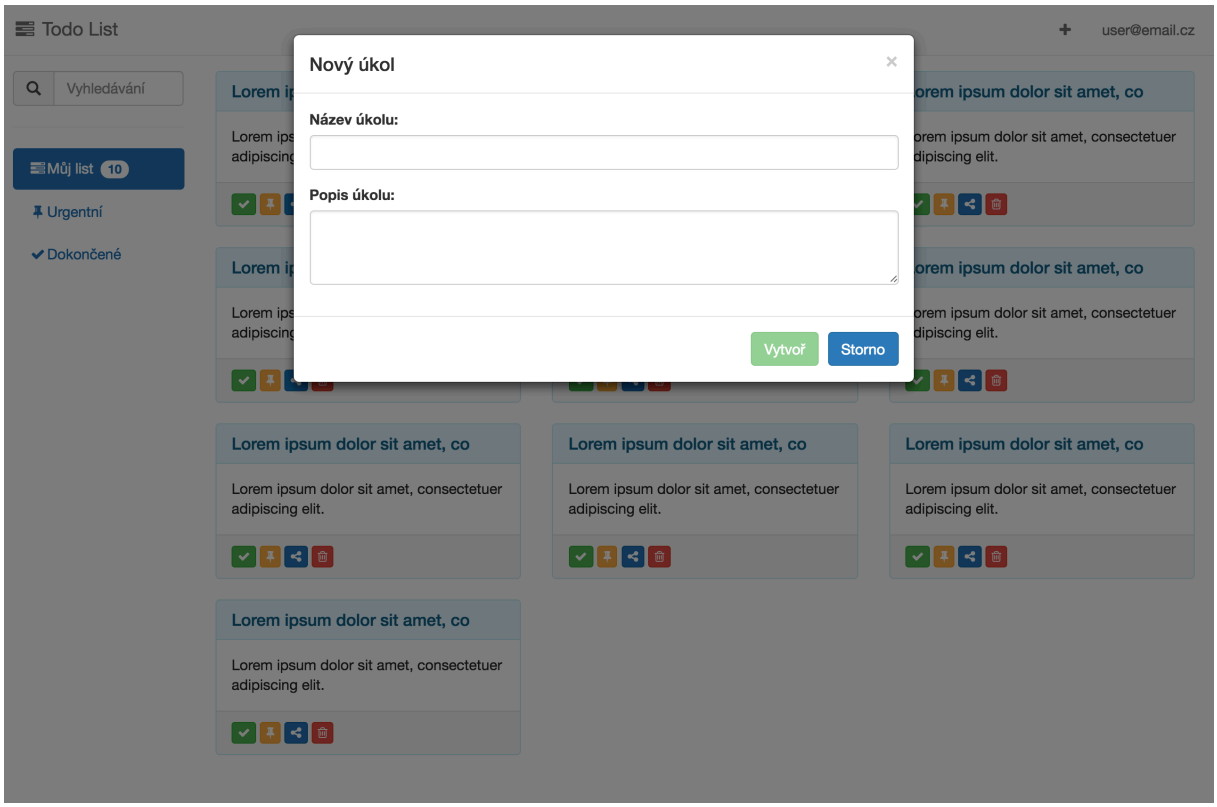
Zkratka	Význam
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CSS	Cascading Style Sheets
DAO	Data Access Object
DDL	Data Definition Language
DML	Data Manipulation Language
DNS	Domain Name System
HATEOAS	Hypermedia As The Engine Of Application State
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JDBC	Java Database Connectivity
JDK	Java Development Kit
JNDI	Java Naming and Directory Interface
JSON	JavaScript Object Notation
JWT	JSON Web Token
MVC	Model-view-controller
MVVN	Model-View-ViewModel
NPM	Node Package Manager
REST	Representational State Transfer
RPC	Remote Procedure Call
SOA	Service Oriented Architecture

SOAP	Simple Object Access Protocol
SPA	Single Page Application
SQL	Structured Query Language
UDDI	Universal Description Discovery and Integration
URI	Uniform Resource Identifier
WSDL	Web Services Description Language
XML	Extensible Markup Language

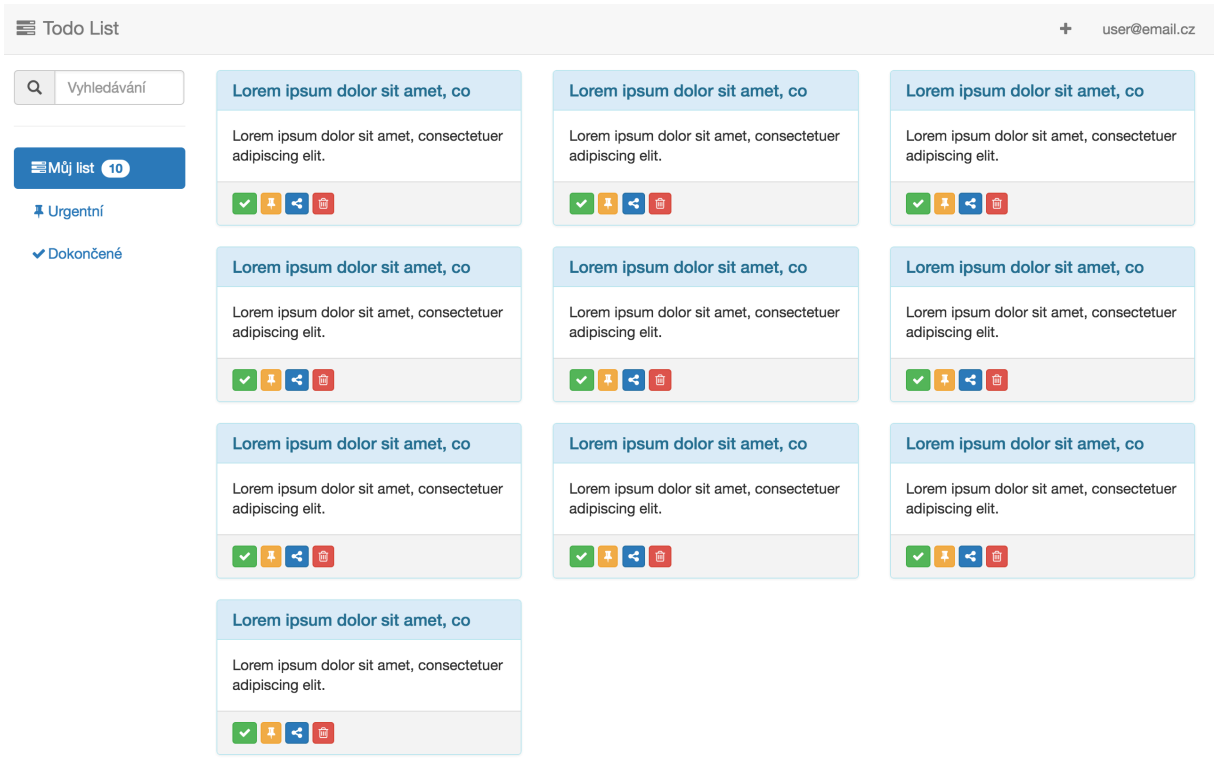
8.3 Obrazovky výsledné implementace aplikace



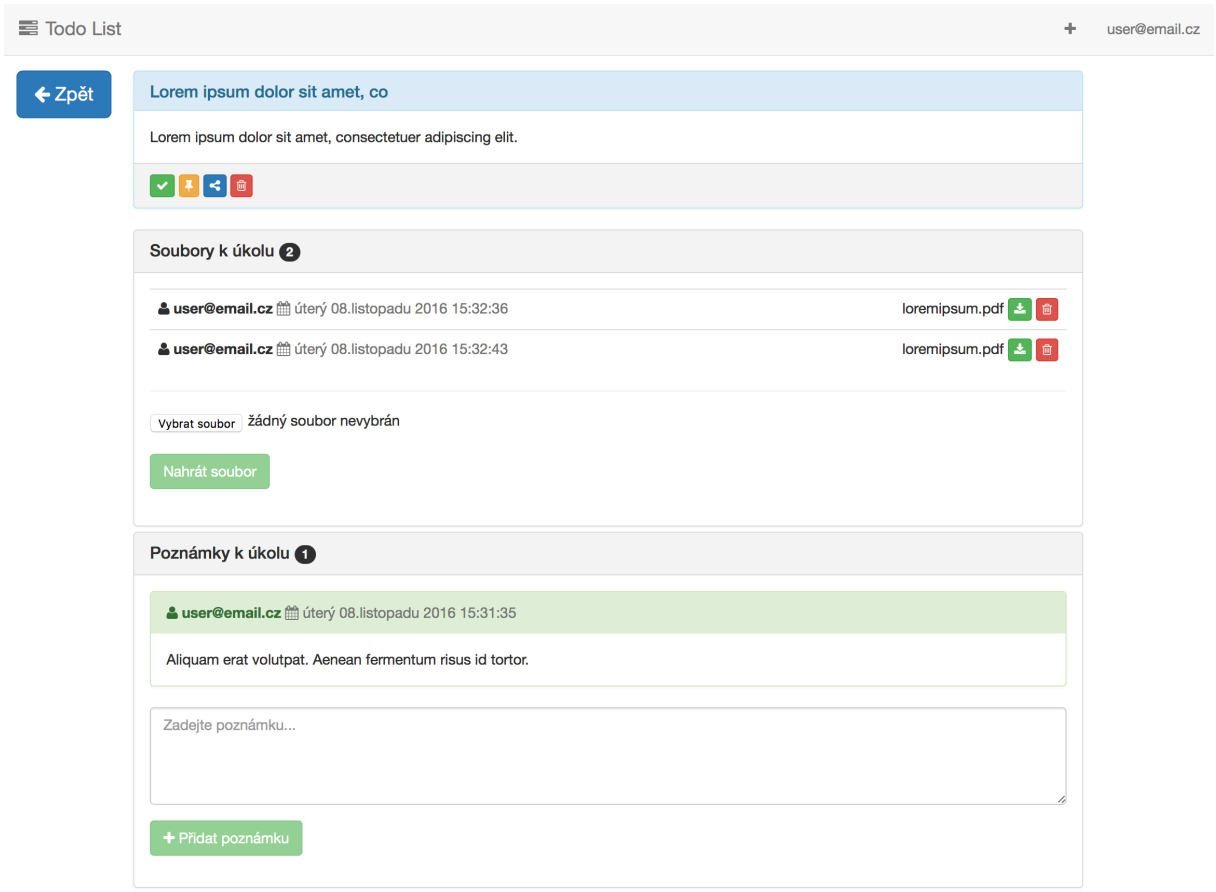
Obrázek 21 – Obrazovka s přihlašovacím formulářem [vlastní zpracování]



Obrázek 22 – Obrazovka pro vytvoření nového úkolu [vlastní zpracování]



Obrázek 23 – Obrazovka se seznamem úkolů [vlastní zpracování]



Obrázek 24 – Obrazovka s detailem úkolu [vlastní zpracování]

9 Seznam tabulek

10 Seznam kódů

11 Seznam obrázků

12 Seznam diagramů