Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Vývoj aplikací v jazyku Kotlin a Spring frameworku

Bakalářská práce

Autor: Marek Bielik
Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Hradec Králové listopad 2021

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

vlastnoruční podpis

V Hronově dne 12.11.2021 Marek Bielik

Poděkování:

Děkuji vedoucímu bakalářské práce panu doc. Mgr. Tomáši Kozlovi, Ph.D. za metodické vedení práce, poznatky k průběhu práce a praktické rady, díky kterým jsem mohl práci úspěšně dokončit. Poděkovat bych chtěl také za samotný návrh a upřesnění tématu bakalářské práce, se kterým mi rovněž pomohl.

Anotace

Název: Vývoj aplikací v jazyku Kotlin a Spring frameworku

Bakalářská práce se zabývá programovacím jazykem Kotlin, frameworkem Spring a tvorbou webových aplikací za využití obou technologií. Jazyk Kotlin je zběžně představen a porovnán s jazykem Java, ze kterého sám vychází. Pro framework Spring jsou zdokumentovány jeho důležité základní moduly nutné pro tvorbu základních webových aplikací. Probrán je návrh ukázkové aplikace a představen je i postup její tvorby se zaměřením na vybranou entitu v průřezu všemi vrstvami aplikace včetně zabezpečovacích mechanismů. V následující části jsou zhodnoceny poznatky z práce na praktické části. V samotné závěru práce je zdůvodněno, nejen proč je Kotlin plnohodnotným náhradníkem Javy i ve spolupráci s frameworkem Spring, ale také proč je možné jej považovat za lepší volbu. Práci je možné využít formou manuálu vývojáři začínajícími s frameworkem Spring.

Annotation

Title: Application development in Kotlin language and Spring framework

This bachelor thesis deals with the Kotlin programming language, Spring framework and the creation of web applications using both technologies. The Kotlin language is briefly introduced and compared with Java based on which it was originally designed. For the Spring framework, the paper documents its important base modules necessary for creating basic web applications. Discussed are the design of the sample application as well as the process of its creation, focusing on a selected entity in a cross-section of all layers of the application, including security mechanisms. The following section evaluates the knowledge gained from the work on the practical part. Lastly at the end of the thesis it is explained why Kotlin is not only a full-fledged replacement for Java even in collaboration with the Spring framework, but also why it can be considered the better choice. The work can be used as a manual by developers starting with the Spring framework.

Obsah

Seznam obrázků

Seznam ukázek kódu

1 Úvod

Webové a mobilní technologie jsou v dnešní době již takřka všudypřítomné. Ať už si chcete přečíst nejnovější zprávy, zkontrolovat bankovní konto, objednat si něco k jídlu či vyhledat jednu z mnoha forem on-line zábavy, budete využívat právě těchto zdrojů. S rostoucím pokrytím celého světa Internetovým připojením se tak jedná o skutečně nedílnou součást životů velmi velké části lidské populace.

Pro tvorbu těchto aplikací je možné využívat nepřeberné množství dostupných technologií. Oblasti tvorby webových aplikací už velmi dlouhou dobu dominuje jazyk PHP, který nabízí mimo jiné například velkou flexibilitu, snadné a nepříliš nákladné hostování a databázové připojení nebo také kvalitní odezvu. Díky těmto a jiným výhodám se stále těší vysoké popularitě, která se zpětně promítá na kvalitě a množství dokumentace spojené s využíváním tohoto jazyka, množství a odbornosti specialistů v oboru a také výběru různých frameworků, mezi něž se řadí například Laravel, Symfony či český Nette. I přes všechny tyto výhody a stále vysokou popularitu však dochází u PHP k poklesu oblíbenosti, neboť často nenabízí řešení pro moderní problémy a je tedy zřejmé, že je nutné ho v těchto případech nahrazovat technologiemi jinými.

Těmito náhradami se rychle stávají především jazyky JavaScript a Python, obzvláště druhý jmenovaný se pak v posledních letech těší velikému růstu v oblíbenosti po celém světě, a to nejen pro své oblíbené backendové frameworky Django a Flask. To ovšem neznamená, že svůj podíl na tvorbě webových aplikací nemají i jiné technologie.

Java se stále řadí mezi jeden z nejpopulárnějších programovacích jazyků vůbec, titul, kterým se může chlubit už dvě desetiletí. Ta také nabízí velké množství výhod, například virtuální stroj JVM (Java Virtual Machine), tedy prostředí, které spouští kód programu a tvoří jej tak platformě nezávislým, velké množství knihoven a frameworků, kvalitní a obsáhlé IDE (Integrated Development Enviroment) programy nebo také integrovanou podporu multi-vláknových operací.

Ani Java však není v každé situaci perfektním řešením a v roce 2011 tak přišla společnost JetBrains s novým jazykem Kotlin, který si za úkol dává zlepšení této velmi oblíbené platformy, ze které sám vychází a se kterou je plně interoperabilní.

Na poli mobilních aplikací se právě Kotlin stal předním jazykem tvorby na platformě Android, ze které Javu postupně odsunuje. Na straně serverové Kotlin využívá několika dostupných frameworků, především pak Spring, který patří mezi nejoblíbenější Java frameworky vůbec a soupeří tak s již zmiňovanými frameworky dostupnými pro ostatní webové technologie. V první řadě se však k porovnání nabízí standard Jakarta EE (dříve Java Enterprise Edition), který vznikl jako originální projekt společnosti Oracle, a přestože Spring samotný není přímou implementací tohoto standardu a je vybudován na platformě Java SE, zpřístupňuje vývojářům i řadu funkcí Jakarta EE bez potřeby speciální serverové infrastruktury implementující celý tento standard.

V tomto velmi rychle se vyvíjejícím prostředí tak Kotlin bojuje o prosazení své pozice jako jeden z lídrů v oboru stejně tak, jako se mu podařilo uspět ve světě mobilních aplikací s platformou Android. Podaří-li se Kotlinu uspět také na trhu webových aplikací, může sám přispět k udržení popularity Springu, který se již těší pozici jednoho z nejoblíbenějších nástrojů pro tvorbu webových aplikací.

2 Cíl práce

Tato práce se zabývá zejména pohledem na tvorbu ukázkové webové aplikace za využití jazyku Kotlin a frameworku Spring. Před jejich praktickým využitím na samotnou aplikaci se zaměřuje také na srovnání jazyku Kotlin s jazykem Java, ze kterého sám vychází, ale také na náhled do frameworku Spring, který je právě Javou tradičně velmi často využíván pro tvorbu těchto aplikací.

Praktickou část práce tvoří návrh a tvorba webové aplikace, v tomto případě sociální síť, která je zaměřená na sdílení kulinářského umění. Jelikož je tento druh aplikace tvořen vrstvami sdílejícími postup tvorby, je pohled práce zacílen především na jednu entitu, která všechny tyto vrstvy znázorňuje.

V konečné části práce shrnuje poznatky z tvorby ukázkové aplikace a hodnotí využité technologie, především pak jejich vzájemnou spolupráci. Nejen závěrečné poznatky, ale také postupy a teoretické shrnutí důležitých částí frameworku lze využít jako manuál vývojářům začínajícím s frameworkem Spring.

3 Srovnání jazyků Kotlin a Java

3.1 Java

Jako jeden z nejvíce historicky zavedených programovacích jazyků se Java řadí mezi nejrozšířenější technologie vůbec. Za 25 let své existence oslovila miliony vývojářů po celém světě a v dnešní době na této platformě běží až 51 miliard virtuálních strojů JVM (Java Virtual Machine). [3] V současnosti se podle TIOBE Indexu řadí na třetí místo nejpoužívanějších programovacích jazyků jen těsně za jazyky Python a C. [4]

Při tvorbě tohoto programovacího jazyku byla hlavním cílem multiplatformní báze, která by umožňovala tvorbu aplikací spotřebovávajících minimum systémových zdrojů a možnost rozšíření na nové platformy. K dosažení tohoto cíle byla Java vytvořena jako jazyk nezávislý na hardwarové architektuře systému, na kterém běží či jeho operačním systému. Touto přenosnou platformou je již výše zmiňovaný virtuální stroj JVM. Místo adaptování jazyku pro rozdílné systémy se tak mění pouze implementace JVM. Platforma je dělena na více částí: Java SE (Standard Edition), která je určena pro desktopové aplikace, Java ME (Micro Edition), určená pro vestavěné systémy a mobilní zařízení, Java Card pro bezpečností karty a jiná čipová zařízení, Java TV založená na Java ME a zaměřená na televizní zařízení a set top boxy s rozšířenou podporou pro práci s videem a Java EE (Enterprise Edition) pro distribuované systémy. [5]

Dalším stavebním pilířem je důraz na výkonnost. Mezi optimalizační prvky patří například velice známý „garbage collector“ („sběrač odpadu“), který se stará o automatické dekonstrukce nepoužívaných tříd a zbavuje tak vývojáře nutnosti dekonstrukce tříd explicitně v kódu. Tato technika uvolňuje operační paměť a předchází tak problémům s jejím přetečením a zajišťuje větší stabilitu aplikací. Dalším prvkem optimalizace je také multithreading (více vláknové operace), který je podporován na úrovni jazyku a v knihovně tříd je pro práci s ním možné nalézt třídu Thread (vlákno).

Neméně důležitou částí je u takto rozšířené technologie i bezpečnost a proto samotný jazyk Java i běhové prostředí obsahují bezpečností prvky, které znemožňují vnější přístup do kompilovaných aplikací.

Jazyk jako takový je objektově orientovaný a s výjimkou několika málo primitivních datových typů tak budou vývojáři vždy pracovat právě s třídami. Na rozdíl od C tak upouští od preprocesorů a hlavičkových souborů, čímž značně ulehčuje práci především na částech kódu, do kterých nahlíží vícero vývojářů a zbavuje je nutnosti znát kontext. Syntaxe však zůstala jazykům C a C++ věrná a umožnila tak lehké sžití s novým jazykem, který byl díky novému, objektovému přístupu mnohem dostupnější především nováčkům. Pro práci je dostupná řada knihoven tříd a API (Application Programming Interface – Rozhraní pro programování aplikací). Tato dodatečná podpora tak velmi ulehčuje další práci při vývoji aplikací. To vše lze vyčíst z představení Javy jejími vývojáři. [6]

3.2 Kotlin

Programovací jazyk Kotlin je zastřešený firmou JetBrains známou především pro tvorbu vývojových prostředí jako jsou IntelliJ IDEA, PhpStorm, PyCharm či WebStorm. [7] Jazyk vznikal původně na platformě JVM, kód je však nyní možné kompilovat i pro Android, JavaScript nebo také přímo pro operační systém. V budoucnu má být rozšířen i o možnosti pro vestavěné systémy, macOS a iOS. Vydán byl původně v roce 2016 a stále dostává pravidelné aktualizace. [8] Řadí se tak mezi další jazyky vzniklé na platformě Java, kterými jsou například Groovy a Scala. V současnosti obsazuje 38. místo v oblíbenosti a je tak předčen oběma těmito jazyky, přičemž právě první jmenovaný si udržuje značnou popularitu a obsazuje 15. místo na žebříčku TIOBE Index. [4]

Navržen byl pro plnou kompatibilitu s Javou. Části kódů těchto dvou jazyků tak podporují bezproblémovou komunikaci a migrace kódu z Javy na Kotlin je ušetřena spousty problémů, neboť není nutné přepisovat celý program. [8] Rozdíly Kotlinu a mnoho dalšího lze mimo knihu [1] studovat i v oficiální internetové dokumentaci. [9]

3.2.1 Obecné změny

Kotlin na rozdíl od Javy umožňuje využití jak objektově orientovaného programování, tak funkcionálního programování. Funkcionální programování je výhodou především pro více vláknové operace, neboť pracuje s neměnnými

(immutable) objekty a nenastávají tak chyby při špatné synchronizaci dat, ke kterým v objektovém programování přistupují třídy z více vláken. [1, s. 6-7] Další výhodou funkčního programování je také snazší testování. Za zmínku stojí i možnost využití takzvaných lambda výrazů, nicméně Java od verze 8 tuto možnost nabízí také.

Primitivní datové typy nejsou explicitně dostupné a uživateli se jeví jako třídy, nad kterými je možné volat základní funkce, nicméně interně je s nimi stále zacházeno jako s primitivními datovými typy. [1, s. 153-154] Třídy v Kotlinu jsou defaultně veřejné a finální a pro vytváření jejich potomků je nutné třídy definovat jako otevřené značkou „open“, naopak abstraktní třídy není nutné deklarovat jako otevřené. [1, s. 70-73] Taktéž metody a vlastnosti jsou bez explicitního označení v Kotlinu privátní. Vlastnosti vytvářené klíčovým slovem „var“ jsou proměnné. Dalším klíčovým slovem pro vytváření vlastností je „val“, které naopak vytváření vlastnosti neproměnné, jedná se tedy o ekvivalent značky „final“ známé z deklarací vlastností v Javě. [1, s. 21] I pole jsou v Kotlinu reprezentována specifickou třídou, jejíž instance se tvoří konstruktorem, další možností je také využití funkce pro vytvoření pole dostupné z knihovny funkcí. [1, s. 167-168]

Změnou prošly i obecné typy, které stačí v Kotlinu deklarovat pouze jednou pro celou třídu nebo rozhraní, což je mnohem elegantnějším řešením. [1, s. 226-227] Java wildcards („divoké karty“) jsou pak nahrazeny specifičtějšími obecnými typy. V Kotlinu je tento modifikátor nazván „use-site variance“. [1, s. 246] S použitím klíčových slov „out“ a „in“ zde lze dále docílit zachování, nebo otočení dědičného vztahu. [1, s. 244-245]

Vývojový tým z JetBrains se také rozhodl pro odstranění takzvaných „checked exceptions“ (kontrolovaných výjimek). Ty totiž v Javě často způsobují pouze tvorbu boilerplate kódu. [1, s. 41]

3.2.2 Novinky

Změnou, která neurazí žádného vývojáře, je redukce boilerplate kódu, tedy kódu, který je nutno často zahrnout, avšak zpravidla v nezměněné formě a stává se tak jakýmsi „nafukujícím“ kódem. Příkladem boilerplate kódu může být třída, která obsahuje pouze data. Tuto třídu je třeba doplnit o konstruktor, funkce pro vkládání a získávání hodnot či další funkce, například převod na řetězec. Pro vytvoření takové

třídy stačí v Kotlinu při deklaraci zahrnout pouze název a vlastnosti třídy spolu se značkou „data“. Taková třída bude při kompilaci kódu automaticky doplněna o všechny potřebné funkce. [1, s. 89-90] [32] Delegované vlastnosti rovněž snižují množství nutného kódu. Pro vlastnosti, které se v kódu často opakují je možné vytvořit takzvaný „delegate“ (delegát), který obsahuje běžné funkce set a get. Vlastnosti zůstanou uloženy v knihovně, případně lze použít delegáty, které se v ní nacházejí defaultně. Je-li vlastnost v její deklaraci odkázána na takového delegáta, bude možné nad ní volat funkce set a get bez nutnosti jejich deklarace. [1, s. 189-190]] Delegované vlastnosti lze kombinovat s „lazy“ (líným) inicializováním, u kterého dochází k inicializaci až při prvním přístupu [1, s. 190-191] nebo také umožní definování „observable“ (pozorovatelných) vlastností, jejichž změny sleduje posluchač. [1, s. 192-195] Jedná se tak v podstatě o jakési rozhraní pro vlastnosti. Delegování je rovněž implementováno pro třídy. [1, s. 91-93] Ani zde však redukce nekončí. Třídy jsou rovněž automaticky vybaveny primárním konstruktorem, který obsahuje všechny vlastnosti třídy. Jelikož přístup ke kódu tohoto konstruktoru není možný, je k těmto účelům využíváno inicializačního bloku, stejně tak, jako jsou například udávány parametry metodám. [1, s. 23]

Nejvíce vyzdvihovanou změnou, kterou Kotlin přináší, je však zcela určitě eliminace výjimek s nulovou (prázdnou) hodnotou známých jako NullPointerException. Je-li při deklaraci proměnné doplněna za datový typ značka „?“ (proměnná je „nullable“), bude této proměnné umožněno obsahovat prázdnou hodnotu i v případě, že není pro datový typ běžná, například u řetězců. Takto změněná hodnota však při kompilaci vytvoří chybovou hlášku, je-li k ní z kódu přistupováno. [1, s. 134-135] Tento problém je řešen bezpečným voláním, pro které je využíván operátor bezpečného volání „?.“ a v případě prázdné proměnné je vrácena prázdná hodnota „null“. [1, s. 137-138] Značkou „!!.“ lze programu naznačit, že hodnota, ke které je přistupováno, určitě nebude prázdná. Lze ji tak využít v případech kdy je prováděn explicitní „null check“ a následně je k ní opětovně přistupováno v jiné části kódu. V případě neprovedení této kontroly a přistoupení programu k prázdné hodnotě nicméně nastává jedna z několika málo možných situací, ve které Kotlin zahlásí výjimku NullPointerException. [1, s. 142-143] Další možností je využití operátoru „?:“ zvaného Elvis Operator. Jedná se v podstatě o

jednoduchou podmínku „if else“, která vrací hodnotu vlevo od operátoru, není-li prázdná, v opačném případě pak vrací hodnotu vpravo. Sloužit tak může třeba k nastavení defaultní hodnoty, dokáže ovšem i vytvořit výjimku. [1, s. 139-140] U proměnných značených jako „nullable“ je nadále možné kontrolovat jejich hodnotu v podmínce pomocí komparátorů „is“ a „!is“. [1, s. 140-141]

Nově je v Kotlinu také možné vytvářet rozšiřující funkce. Jedná se o metody vytvářené vně třídy například v případě potřeby rozšíření tříd nepřístupné knihovny bez nutnosti vytvářet jejich potomky. Při deklaraci této funkce se před její jméno přidá tečkou oddělená třída, nad kterou bude daná metoda volána. Protože i primitivní datové typy jsou v Kotlinu zaobaleny do tříd, je možné vytvářet rozšiřující funkce i pro ně. [1, s. 51-53]

Jedináček, v Kotlinu anglicky zvaný „singleton“, je druh třídy, která může mít pouze jednu instanci. Pro deklaraci takové třídy je místo klasického klíčového slova „class“ využíváno označení „object“. [1, s. 93] Tyto singletony lze také využívat pro tvoření „companion“ objektů (společníků), které umožňují přidávání statických vlastností a funkcí. Ty lze pak využívat i před vytvořením instance třídy, nicméně se nejedná o skutečné statické vlastnosti a funkce, jaké jsou známé z Javy. [1, s. 96-97] V Kotlinu, jak je již z ostatních změn zřejmé, je vše ve skutečnosti instancí nějakého objektu a ani „companion“ objekty nejsou výjimkou. [1, s. 98-99]

Intervaly slouží k vytvoření uzavřeného rozmezí mezi dvěma číselnými hodnotami nebo mezi dvěma písmeny v abecedním pořadí. [1, s. 36] Posloupnosti se liší v možnosti nastavitelného krokování mezi první a poslední hodnotou a slouží tak především pro využití v cyklech. Typický zápis takového cyklu známý z Javy je pak zkrácen na příkladem pouhé „for (i in 0..10)“.

Na rozdíl od Javy umožňuje Kotlin také vlastní implementace funkcí pro operátory. Jedná se o funkčnost známou například z programovacího jazyka C++ jako „operator overloading“ (přetěžování operátorů). [1, s. 173-174]

3.2.3 Syntaxe

Kromě úplných syntaktických novinek popsaných v předešlých sekcích Kotlin prošel oproti Javě i několika jednoduchými změnami. První změnou, kterou postřehne každý programátor, je zcela určitě absence středníků na konci

řádků. Nejedná se ovšem o absenci vynucenou, je-li to nutné pro čitelnost kódu, středníky je možné nadále využívat. Další malou změnou prošly vlastnosti, které mají v Javě datový typ deklarovaný před názvem, v Kotlinu se naopak píše za jméno a je oddělen dvojtečkou a mezerou.

Změněny byly i zápisy funkcí ve kterých je návratová hodnota, která je deklarována až za dvojtečkou následující závorku s parametry. Jednoduché funkce lze také zapsat i jednořádkovým způsobem.

Stejně tak je dvojtečka v pozici za deklarací třídy využívána k rozšířené jiné třídy, ale také k implementaci rozhraní. Závorka s parametry, které třída přijímá, pak slouží zároveň jako její primární konstruktor.

Mezi dalšími změnami si lze všimnout například možnost vynesení návratové značky „return“ před větvící se podmínku. Celkově je tak z Kotlinu zřejmé, že jeho tvůrci upřednostňují především minimalizaci kódu. Přestože těmto změnám není v knize věnována samostatná část, hodně jich lze nalézt v kapitole 9 této dokumentace. [1]

3.3 Využití Kotlinu

Největší využití Kotlin v dnešní době nachází jednoznačně při vývoji mobilních aplikací pro operační systém Android. Android však ani zdaleka není jedinou platformou, na kterou tým z JetBrains se svým programovacím jazykem cílí. Kotlin se stává velmi oblíbeným i pro platformu JVM a míří i na další místa. Se svou podporou pro JavaScript se jeho uživatelé zaměřují i na tvorbu webových aplikací a stránek. S platformou Kotlin/Native je dále možné cílit i přímo na operační systémy. Zde se největší oblibě těší aplikace pro operační systémy Windows a Linux. Dotazovanými vývojáři byl dále označen za jeden z nejpopulárnějších jazyků, který plánují v budoucnu používat. [10]

3.3.1 Mobilní aplikace (Android)

V dnešní době je již Kotlin společností Google upřednostňován a doporučován pro tvorbu mobilních aplikací pro Android. [11] Starší verze Javy, kterou jsou vývojáři mobilních aplikací nuceni využívat z důvodu podpory starších verzí Androidu, má na dnešní dobu značné nedostatky. A právě Kotlin je jejich

řešením, neboť umožňuje značně zjednodušené programování funkčností jako jsou například „listeners“ (posluchače) ovládání. Jeho značná eliminace výjimek s nulovou hodnotou zajistí větší stabilitu, která nebude narušena ani problémy s kompatibilitou, neboť Kotlin nabízí pro Javu plnou interoperabilitu. Kotlin je pak samozřejmě i plně podporován vývojovým prostředím Android Studio společnosti Google.

Každý z vývojářů má pro použití Kotlinu rozdílný důvod. Někteří z dotázaných, jak píšou Martinez a Mateus [48, s. 13-16], uvedli mimo jiné důvody jako právě kompatibilitu napříč verzemi systému Android a eliminaci výjimek, ale dále také uváděli potřebu využití funkcionálního programovacího stylu, kterým Java nedisponuje. Jiní také ocenili modernitu Kotlinu, interoperabilitu s Javou, někteří dokonce sdělili, že pro vlastní vývoj adoptovali Kotlin, protože neradi pracují s Javou. Zmíněna byla také vyhlídka do budoucnosti ve formě multiplatformního vývoje, tedy jednotného vývoje pro Android a iOS, který by měl Kotlin časem přinést.

Průzkumem dále Martinez a Mateus [48, s. 8-9] dokazují, že až 156 z celkem testovaných 374 aplikací, v přepočtu 42 % aplikací, vytvořených dotázanými vývojáři je v jejich poslední verzi psáno již kompletně v Kotlin kódu. Pouze 4 aplikace (1 %) jsou i v nejnovější verzi psány výhradně v Javě. Zbylých 214 (52 %) obsahuje části kódu psaných v obou jazycích. Jako způsoby integrace obou jazyků pak dotázaní uvedli především 2 rozdílné strategie. V první variantě vývojáři migrují kód z Javy do Kotlinu v případech, kdy je nutné provést změny na souboru, provádějí-li refaktorování kódu, ale také pokud je taková migrace prostě jednoduchá a časově nenáročná. Jiní se naopak rozhodli vytvářet všechny nové části kódu v Kotlinu a při změnách kódu původního nadále využívají Javu. [48, s. 18-19]

Nejedná se tak pouze o iniciativu společnosti Google, ale i z přístupu vývojářů samotných je zřejmé, proč se Kotlin stává předním programovacím jazykem pro tvorbu mobilních aplikací.

3.3.2 Programování na straně serveru (webové aplikace)

Jedná se o další populární využití Kotlinu, přestože v tomto ohledu stále zaostává za jazyky se kterými jsou vývojáři zvyklí pracovat již řady let. Díky této podpoře je možné tvořit nejen backendovou stranu pro výše zmiňované mobilní

aplikace, ale i webové aplikace pro internetové prohlížeče. Díky již několikrát zmiňované interoperabilitě s Javou je tyto aplikace možné hostovat na všech serverech, které podporují webové aplikace založené právě na Javě.

Stejně tak jako mobilní aplikace i webové aplikace těží ze změn a novinek, které Kotlin přináší. Oproti Javě dále nabízí například knihovnu funkcí pro generování HTML kódu. Díky standardnímu Kotlin kódu je tak možné vytvářet HTML kód bez nutnosti využití šablonových jazyků, což vývojáře úplně zbavuje přidané práce spojené s používáním dalších tvůrčích prostředků. K dostání je také řada frameworků, mezi nimi i velmi známé frameworky jako jsou Spring (od verze 5.0), který nabízí i generátor projektů, čímž značně urychluje spuštění nových projektů, Javalin, který se zaměřuje na velmi odlehčený přístup k internetovým protokolům či framework Ktor, který vytváří společnost JetBrains přímo pro Kotlin a cílí na vysokou škálovatelnost. Dále je pak možné využívat frameworky Vert.x, kotlinx.html a Micronaut. [12]

Ačkoliv Kotlin stále nachází největší uplatnění při tvorbě mobilních aplikací pro operační systém Android, s podporou, které se mu dostává pro tvorbu webových aplikací, je zřejmé, že své uplatnění nachází právě i na serverech.

3.4 Shrnutí

Jak bylo výše rozebíráno, Kotlin nabízí oproti Javě řadu vylepšení, mezi která patří zejména redukce napsaného kódu a značné snížení výskytu chybových hlášení a tedy i neočekávaných pádů aplikací. Jakožto relativně mladý jazyk také oproti Javě netrpí na problémy spojené se zachováváním původního kódu, neboť právě druhý ze zmiňovaných programovacích jazyků si stále udržuje mnoho původních funkčních prostředků z důvodu podpory velmi starých aplikací a softwarů. Popularitě se v současné době těší především u vývojářů mobilních aplikací na platformě Android, kteří jej hojně adoptují. I přes menší popularitu využití v jiných oblastech se mu dostává značné podpory například pro tvorbu právě webových aplikací a jeví se tak momentálně jako vhodný kandidát pro využití při práci na takových projektech.

Je však třeba zmínit, že například velkého snížení množství boilerplate kódu u jazyku Kotlin (nebo třeba i u jazyku Scala) lze dosáhnout i u starších verzí Javy

díky frameworku Project Lombok, který nabízí řešení v podobě anotací. Ještě důležitější je pak nezapomenout, že i společnost Oracle začala především v posledních letech intenzivněji pracovat na aktualizacích pro Javu a Kotlin v řadě funkcí dohání. Příkladem boilerplate kódu může být typicky datová třída, kterou Kotlin elegantně vyřešil prostým označením třídy klíčovým slovem „data“ a stejně tak její vytvoření zrychlil i Project Lombok vhodnými anotacemi. Od verze Java 14 přichází Oracle se stejným řešením s označením třídy „record“, které dodá jak přístupové funkce k hodnotám a konstruktor, tak i implementace pro typické metody equals, hashCode a toString. U jiných funkcí jako jsou například rozšiřující metody Java zůstává stále pozadu, ovšem i zde může Lombok zastoupit Kotlin, který tuto možnost poskytuje.

Co se výkonu obou jazyků týká, je možné na základě stejné platformy předpokládat velmi podobné výsledky. Potvrzení tohoto předpokladu dokazuje Anioła [42] řadou testů, jejichž výsledky sice naznačují nepatrné zpomalení Kotlinu oproti Javě, nicméně tyto hodnoty jsou v reálném použití neznatelné a jediné skutečné zpomalení vzniká při kompilaci aplikace.

Z výše uvedených informací je tak již na každém vývojáři, pro jakou z možností se rozhodne. Rozhodně ovšem nelze říct, že by nejnovější verze Javy za Kotlinem velmi výrazně zaostávala.

4 Spring

Pro svou popularitu se jazyk Java těší i velkému množství prostředků jako jsou dokumentace a také nejrůznější knihovny a frameworky. Mezi ty nejúspěšnější frameworky patří právě Spring založený na specifikaci Jakarta EE, který se již téměř dvě desetiletí zaměřuje především na tvorbu webových aplikací. [13] Práce na takových aplikacích je jejich vývojářům značně ulehčena díky redukci boilerplate kódu a nastavení nutných pro jejich vývoj. Pominout nelze ani skutečnosti, že Spring nabízí prověřené bezpečností prvky či snadné přístupy k databázovým úložištím. [20]

Framework staví na dílčích aplikacích zvaných „microservices“ (mikro-služby). Jedná se v podstatě o rozdělení kódu do samostatně činných částí. Této dnes již běžné praktiky využívá s příslibem zjednodušení údržby kódu, jeho zefektivnění, lepší stability atd. [14] Ve spojení s přístupem zaměřeným na události v systému jsou pak tyto mikro-služby schopné řídit i neustálé toky dat, což Spring činí vhodným kandidátem i pro aplikace, které využívají právě velký přenos dat či zpráv, například pro platformy poskytující služby, souborové systémy, video služby, komunikační sítě aj. [15]

Tvůrci frameworku lákají také na jeho reaktivnost, tedy soubor vlastností, který systému dodává kvalitní možnosti kontroly toku procesů. Reaktivnost činí systém velmi responzivním a zaručuje rychlou odezvu i přes velkou propustnost, to vše díky vhodnému využití systémových zdrojů a maximalizaci efektivity kódu. Takové aspekty jsou velmi důležité obzvláště pro velké systémy, které řídí velký přenos dat pro velká množství současně připojených uživatelů. V dnešní době již Spring nabízí reaktivní metody i pro přístup k datům díky podpoře mnoha oblíbených relačních databází mezi které patří MySQL, PostgreSQL, Microsoft SQL Server aj. [16]

Dále Spring nabízí i podporu pro distribuované systémy, tedy systémy, které přenášejí určité části systému na jiné platformy, které je poskytují jako služby. V tomto případě se jedná o cloudové služby a právě pro jejich podporu je Spring připraven díky svým řešením pro 12 faktorů programů poskytovaných jako služby, mezi které patří například konfigurace, konektivita, logování aj. [17][18] Rozvinutá

podpora pro cloudové služby dále umožňuje budování „serverless“ aplikací (bez serveru). Vývojáři se tak mohou naplno věnovat samotné aplikaci a nemusí řešit problémy spojené s infrastrukturou, která bude díky této podpoře přesunuta na cloud. Další výhodou tohoto přístupu je také absence problémů se škálovatelností infrastruktury. Například při nečinnosti jsou takovému systému delegovány minimální zdroje, což výrazně snižuje provozní náklady na infrastrukturu. [19]

V neposlední řadě umožňuje Spring i hromadné zpracování dat, tedy takové zpracování, u kterého nemusí docházet k žádným interakcím se systémem. Jedná se tak o zefektivnění pracovních činností, neboť je možné je naplánovat a šetřit tak systémovými prostředky. V kombinaci s delegováním těchto hromadných operací na již zmíněné cloudy tak mohou vývojáři maximalizovat efektivitu jejich systémů a aplikací. [21]

Dohromady všechny tyto prvky v kombinaci s velkým množstvím knihoven třetích stran utváří framework, který je pro vývojáře velice flexibilní. Jeho oblíbenost zaručuje i dostupnou dokumentaci a návody, které pomáhají komunitu okolo tohoto frameworku dále zvětšovat.

Nazvat však Spring frameworkem není možná zcela přesné. V dnešní době se totiž již jedná spíše o kolekci různých frameworků, každý starající se o samostatnou část výsledné aplikace. Tyto části jsou nazývány moduly. Jak tedy vypadá Spring při bližším pohledu? Některé, hlavně ty nejdůležitější části jako jsou Spring Data nebo Spring Security, budou v práci podrobeny samostatným pohledům. Spring je ovšem velice obsáhlým nástrojem, který uspokojí nároky i velkých společností, a není proto možné zabývat se každou z jeho částí. Z tohoto důvodu bude i praktická část práce pojednávat především o modulech a jednotlivých částech těchto modulů, které jsou pro ukázkovou aplikaci nejvíce relevantní. Zároveň bude vzhledem k formě práce diskutována především ta část modulů, se kterou se vývojáři setkávají přímo ve vývoji, protože cílem je spíš přiblížit vývoj se Springem nezasvěceným, než detailně nahlížet do útrob tohoto komplexního nástroje.

4.1 Spring Framework

Hlavním nebo také „core“ (jádro) modulem je Spring Framework. Po předchozím tvrzení, že Spring jako takový vlastně již není frameworkem, ale spíše

kolekcí frameworků, může být tento název částečně matoucí. Jedná se totiž skutečně o ten původní projekt, který stále nese stejné jméno. Mnoho dalších modulů, některé z nich již byly zmíněny, nejsou součástí přímo tohoto „core“ frameworku a jsou přidávány dle potřeby skrze soubor pro připojení projektových závislostí. I proto tvůrci frameworku berou velmi vážně zpětnou kompatibilitu, neboť celý Spring umožňuje vývojářům značnou flexibilitu a vznikají tak velká množství různých konfigurací, často obsahujících i staré moduly, které by se bez takového přístupu jistě neobešly. Mezi základní moduly zcela nepochybně patří konfigurační model a mechanismus vkládání závislostí. [22, s. 2]

Vkládání závislostí, v angličtině známé pod pojmy „dependency injection“ nebo také „inversion of control“ (dále již jen IoC) je princip využívaný při tvorbě aplikací, ve kterém jednotlivé části programu definují, na jakých dalších částech jsou samy závislé. Každá taková část je ve Springu známá jako „bean“ (fazole). Tyto „fazole“ jsou pak spravovány právě IoC kontejnerem, který má na starost jejich konfiguraci a vytváření instancí, ale také skládání programového kontextu, tedy spojování definovaných závislostí. [23, s. 2-3] Ve standardních přístupech je zcela běžné vytvářet instance objektů za pomoci jejich konstruktorů a tyto instance následně vkládat do konstruktorů tříd, které jsou na nich závislé. Toto však může být pro rozsáhlé podnikové aplikace značně nepřehledné a i zbytečné. S přístupem IoC je za delegování tříd pro potřebné závislosti odpovědný právě kontejner. Ten vytvoří potřebnou instanci objektu a následně ji udržuje pro využití v jiných částech programového kontextu. Z toho vyplývá, že při každém volání závislosti není vytvářena instance nová, nýbrž je aplikaci předávána již existující instance objektu. [24] Kontejner je řízen konfiguračním souborem, psaným jazykem XML, avšak pro vývojáře však není nutné tuto konfiguraci zvláště vytvářet, pro Spring stačí pouze označovat objekty pomocí anotací. [23, s. 3-5] Součásti využívané principem IoC budou nyní blíže popsány v následujících kapitolách.

4.1.1 Spring Bean

Spring **Bean** – základní stavební jednotka aplikace, která je řízena IoC kontejnerem. Tato „fazole“ obsahuje vlastní metadata, mezi která patří: třída, jejíž instance bude vytvořena, název, rozsah, defaultně nastaven pro celý aplikační

kontext, argumenty konstruktoru, vlastnosti a vztahy s ostatními objekty. Dále obsahují také informace o inicializačních a destrukčních metodách a o inicializačním módu (defaultně jsou objekty inicializovány při startu aplikace). [23, s. 10-18]

Metadata je možné upravovat i ručně, nicméně je nutné dodržovat určitá pravidla. Doporučené je zcela jistě i dodržování jmenné konvence (stejná jmenná konvence jako ta známá z jazyku Java), avšak sami tvůrci frameworku se snaží vývojáře motivovat proti zásahům do těchto dat příslibem využití například automatického spojování. [23, s. 11-13]

Závislosti jsou do objektů vkládány zejména při jejich inicializaci, nicméně je možné zvolit i možnost jejich vložení za využití setterů, které jsou volány až po inicializaci objektu prázdným konstruktorem. [23, s. 18-24] Vkládání závislostí mimo konstruktor je pak také doporučováno pouze pro volitelné závislosti. [23, s. 24] V takovém případě však není vyloučené celkem běžné vytvoření kruhových závislostí u kterých se třídy v jejich konstruktorech navzájem vyžadují. [23, s. 25] Lehce je možné se do takového cyklu dostat například v servisní vrstvě aplikace.

4.1.2 Anotace

Alternativou pro XML konfiguraci Spring frameworku je využití anotací. Takto vedená konfigurace je blíže spojená s objekty samotnými, poskytuje také kontext pro kohokoliv, kdo bude číst kód, ale na druhou stranu je taková konfigurace také rozmístěná po celém projektu a může tak být značně nepřehledná. Anotace jsou v třídách značeny pomocí znaku „@“.

Anotací je ve Springu velké množství, pouze s těmi nejdůležitějšími z nich se však operuje skoro v každém z projektů na frameworku postavených. Mezi ně patří některé následující:

@Configuration – označuje třídy nahrazující defaultní chování určitých částí aplikace [23, s. 144-145], typicky využívána třeba pro nastavení zabezpečení Spring Security. Konfigurační třídy jsou využívány také k definování Spring Beans.

@Bean – základní anotace, která je součástí konfigurace a vrací instanci objektu. [23, s. 144-145] Spring využívá návrhového vzoru singleton (jedna instance třídy) a tyto singletony řídí IoC kontejnerem, který takto

vytvořené instance drží a dle potřeby deleguje. [23, s. 52-53] **@Primary** rozšiřuje anotaci @Bean pro upřesnění preference vytvoření této instance. [23, s. 97-98]

@Autowired – zajišťuje automatické vkládání existující instance třídy v potřebném místě. Je možné ji přiřadit ke konstruktoru, nicméně v případě konstruktorů již u novějších verzí Springu není nutné tuto anotaci deklarovat nemá-li třída více konstruktorů nežli primární. [23, s. 89]] Alternativou je anotace **@Inject**. [23, s. 88]

@Component – stereotypní anotace která Springu značí, že má být vytvořena instance třídy pro použití v IoC kontejneru, podobně jako u anotace @Bean. Od této abstraktní anotace se odvozují následující tři anotace, které přidávají komponenty do kontextu aplikace. Komponenty jsou automaticky pojmenovávány na základě názvu třídy, nicméně je možné definovat i názvy vlastní, případně vytvořit i vlastní implementaci generátoru jmen. [23, s. 130-132] Anotace **@Scope** umožní změnu rozsahu komponenty z defaultního singletonu. [23, s. 132-134] Tím je možné řídit, kolik instancí třídy má být pro různé části aplikace vytvářeno.

@Controller – prezenční vrstva aplikace. V těchto kontrolérech se nachází metody pro obsluhu koncových bodů aplikace. Varianta **@RestController** je dále rozšířena o funkci anotace **@ResponseBody** [26], která serializuje data odeslaná z HTML stránky do určitého objektu (například JSON). Samotné koncové body jsou pak anotovány pomocí **@RequestMapping** s příslušnou metodou a adresou v parametru, nebo pomocí čtveřice anotací určených pro každou z CRUD metod s adresou v parametru a to **@GetMapping**, **@PostMapping**, **@PutMapping** a **@DeleteMapping**.

@Service – značí servisní (business) vrstvu, se kterou kontroléry spolupracují. Servisní třídy typicky implementují aplikační logiku zpracovávající data.

@Repository – data pro servisní vrstvu poskytuje datová vrstva (Data Access Object) značená touto anotací. Ta přistupuje k databázi a ve Spring lze mimo tvoření kompletně vlastních dotazů využít i jako pouhé rozhraní

příslušné databáze. S dodržením jmenné konvence lze pak tomuto rozhraní deklarovat metody, které budou Springem automaticky implementovány.

@Document – slouží pro označení datové třídy, tedy modelu, který bude vkládán do databázové kolekce MongoDB. Název kolekce bude defaultně stejný, dá se však změnit pomocí parametru anotace. [35, s. 68] Tvoří tak perzistentní vrstvu aplikace a je zpracováván servisní vrstvou, které je předán vrstvou datovou.

@Value – přiřazuje příslušnou hodnotu například z externího souboru. [25] Taková možnost je vhodná třeba pro vložení základní hodnoty, není-li vytvořena při inicializaci.

@Lazy – zajistí vytvoření instance jakékoliv třídy pro aplikační kontext až při jejím prvním volání namísto defaultní inicializace již při startu aplikace, čímž zrychluje její chod. [23, s. 176]

Správné využití anotací je nejen nutné pro funkci aplikace, ale pomáhá také vnitřní logice Springu pracovat rychleji, například při výskytu výjimek. [23, s. 119] Nastavovat jimi lze i všechna potřebná metadata automaticky detekovaných komponent aplikace. Mezi další se řadí také anotace pro práci se Spring MVC (Model-View-Controller) a celkově uživatelskými požadavky a zpětnými odpověďmi, modulem Spring Cloud, plánování operací, pro testování aplikace a projekt Spring Boot. Zcela jistě se tak tento základní výhled nedotýká všeho, co anotace ve Springu nabízejí.

4.1.3 Spring Expression Language

Jednou z dalších fundamentálních součástí Springu je rozšíření o vlastní sadu výrazů k vyhodnocení při běhu programu, které je také možné vkládat do anotačních či XML konfigurací pro získání a automatické vyplnění potřebných hodnot. Svou syntaxí tento slovník vychází z klasického Jakarta Expression Language nebo spíše z jeho vlastního předchůdce. [23, s. 274] Ten je navržen k využití v HTML stránkách a vývojáři pracující s frameworkem Spring se tak setkají

s podobnou syntaxí i u šablonového enginu pro aplikace v prostředí JVM zvaném Thymeleaf.

Na rozdíl od svého předchůdce však Spring Expression Language, zkráceně SpEL, nabízí i možnost volání funkcí šablonování textových řetězců. Přestože se jedná o další projekt tvůrců Springu, je SpEL technologicky nezávislý a je tak nejen možné jej kombinovat s jinými implementacemi Object-Graph Navigation Language společnosti Apache, ale také jeho využití bez závislosti na Spring struktuře. V takovém případě je nutné pouze vytvoření instance syntaktického analyzátoru (parseru) a další infrastruktury potřebné pro běh tohoto analyzátoru. [23, s. 274]

ExpressionParser – rozhraní zodpovědné za analyzování výrazů formovaných textovými řetězci, které pracují ať už s primitivními datovými typy, jejich kolekcemi či mapami a dalšími objekty. [23, s. 274-275] Nad těmi je možné volat a řetězit příslušné funkce a dokonce je možné v těchto řetězcích i vyvolání konstruktoru. [23, s. 275-278] Zde SpEL pracuje s klasickými operátory – aritmetickými, logickými a relačními operátory, regulárními výrazy a ternárními operátory pro zkrácený zápis podmínek, v tomto případě dobře využitelnými například pro nastavení defaultních hodnot.

Analyzátor syntaxe je také možno konfigurovat. Udávaným příkladem je situace, ve které daný výraz sahá pro neexistující položku do pole. V případě neexistence tohoto indexu je příhodné nechat analyzátor automaticky rozšířit pole a vložit defaultní hodnotu s využitím konstruktoru objektu nacházejícího se v poli. Pokud tento objekt konstruktor nemá či vytváření jinak selže, pak vkládá prázdnou hodnotu. [23, s. 280] Další možností je i vkládání hodnoty do indexu pole který není neexistující, ale prázdný.

Všechny tyto výrazy jsou při běhu programu interpretovány, nicméně SpEL disponuje i vlastním kompilátorem, který je v defaultním nastavení vypnut. [23, s. 282] Interpretovaný jazyk je velice flexibilní, pokud u daných výrazů při běhu programu dojde k určitým změnám u dat, se kterými pracují, nedochází k chybám. Kompilovaná podoba ovšem proběhne mnohem rychleji a je tak přínosnější pro výkon. Kompilátor disponuje dvěma nastaveními, „immediate“ (okamžité), tedy režim, při které jsou všechny výrazy kompilovány okamžitě při startu programu a „mixed“ (smíšené) nastavení, u kterého dochází ke kompilaci až po několikátém

volání daného výrazu. U první zmíněné možnosti může docházet k chybám dojde-li ke změnám objektů, se kterými výrazy pracují. V takovém případě je vyvolána výjimka. U druhé možnosti výjimky vyvolávány nejsou, místo toho je v případě chyby zpracování výraz interpretován. Nenastane-li po několika zavoláních u výrazu chyba, je opět kompilován pro zrychlené zpracování dalšími voláními. [23, s. 282-283]

4.1.4 Zdroje

Práce se zdroji je zcela nepochybně důležitou součástí každého moderního dynamického webu či aplikace a samozřejmě i touto problematikou se tvůrci Spring frameworku zabývají. Nedostačující možnosti „holé“ Javy (Standard Edition) sami doplnili o nové funkcionality.

Resource – tedy zdroj, je nové rozhraní, se kterým samotný Spring velmi často pracuje. To navíc rozšiřuje další rozhraní „InputStreamResource“ (zdroj datového proudu) které se stará o předávání informací o datovém proudu. Rozhraní Resource nabízí metody například pro navrácení datového proudu daného zdroje, získání popisu zdroje, jeho URL nebo i typickou reprezentaci File (soubor) známou z Javy. Mimo množství jiných metadat nese také informaci o tom, zda musí být zdroj po prvním načtení uzavřen, či nikoliv. [23, s. 218-220] V zásadě se tak jedná o velmi abstraktní reprezentaci různých druhů zdrojů, ke kterým lze díky tomuto rozhraní přistupovat stejným způsobem.

ResourceLoader – slouží k načítání zdrojů, je automaticky součástí každé Spring aplikace a není tak třeba vytvářet samostatnou instanci tohoto rozhraní pro přístup k jeho metodám. Toto automaticky užívané rozhraní je v tomto případě dále rozšířeno rozhraním pro hledání vzorů v cestách k daným zdrojům a nakonec implementováno třídou PathMatchingResourcePatternResolver, tedy třídou, která dokáže zdroje nalézt ať už podle jejich úplné adresy nebo „classpath“ identifikátoru. [23, s. 225]

V poslední řadě je možné zdroje připojit k projektu jako závislosti skrze konfigurační XML soubor. K načtení zdroje bude dle kontextu aplikace využita jedna z existujících implementací zdrojů, nicméně je možné vybrat jinou implementaci vhodným prefixem. [23, s. 226] Tato možnost urychlí vkládání zdrojů do míst, do

kterých by byly pouze přesouvány třídou pro načítání zdrojů a zase tak ušetří vývojářům několik řádků kódu.

Samotné rozhraní ovšem pro vývojáře nenabízí mnoho. Proto obsahuje Spring i řadu implementací tohoto rozhraní, které poskytují už hotová řešení pro části aplikace, které by mnoho vývojářů mohlo považovat za hraniční boilerplate kód.

ClassPathResource – umožňuje přístup ke všem souborům složek, jejichž cesty jsou připojeny v konfiguračním souboru. Cesta pro vyhledání těchto zdrojů se klasicky nazývá „classpath“. [23, s. 221]

FileSystemResource – alternativní implementace tohoto rozhraní. Ta navíc využívá File API poskytované přímo Javou. Podobným způsobem lze využít i **PathResource**. [23, s. 221] Na rozdíl od předchozího způsobu se neřídí cestami určenými v konfiguračním souboru, ale celou adresou v souborovém systému, případně i adresou relativní k umístění projektu v souborovém systému.

UrlResource – rozhraní nabízející univerzálnější řešení. Tato třída umožní přístup ke zdrojům klasickou adresou v podobě textové řetězce, nicméně obsahuje-li tento řetězec správně formátovanou hlavičku, jako například „file:“, pak je tento požadavek automaticky zpracován jako FileSystemResource. Dalšími detekovatelnými způsoby jsou vyhledávání s hlavičkami „https:“ pro získání dat skrze příslušný hypertextový protokol či „ftp:“ pro zpracování protokolem pro přenos souborů. [23, s. 220-221]

ServletContextResource – řídící se cestami relativními ke kořenové složce projektu na rozdíl od předchozích implementací pracujících i se soubory existujícími mimo projektový kontext umožní přístup pouze k souborům, které se v daném projektu skutečně nacházejí. [23, s. 221] URL adresy nicméně umožní získávat v každém případě.

Dalšími dostupnými implementacemi souborového rozhraní ve Spring frameworku jsou také **ByteArrayResource**, **InputStreamResource**, **FileUrlResource**, **AbstractResource** a několik dalších speciálních tříd. Existuje tak velké množství možností, se kterými by si měli vývojáři v běžných situacích vystačit.

4.1.5 Datový přístup

Práce se zdroji je jedním ze způsobů zpracování externích informací uvnitř aplikace, není však vhodná ani zdaleka pro všechny situace. Dalo by se spíše říct, že má naprosto jiný úkol než databázové systémy. Problematikou spojení s databázemi samotnými se bude zabývat další část práce, nicméně řešením pro přenos dat mezi určitým rozhraním databáze a servisní vrstvou aplikace se zabývá již část původního core modulu Spring Framework.

Tyto transakce obstarává abstrakční model, který je konzistentní mezi různými specifikacemi jako jsou JPA (Java Persistence API), JTA (Java Transaction Api) či JDBC (Java Database Connectivity). [27, s. 2] To znamená, že jediná verze kódu pro řízení transakcí může fungovat nezávisle na tom, zda se jedná o transakce lokální, či globální. [27, s. 2-3] Není tak nutné řešit speciální situace, ve kterých je například použito vícero databázových zdrojů. To je značnou výhodou oproti rozhraním jako například Hibernate (specifikace JPA) zastávající pouze lokální transakce a umožňující mapování modelů pouze pro jeden databázový zdroj.

Přímo tvůrci Springu doporučeným způsobem řízení datových transakcí je způsob ve Springu známý jako deklarativní. [27, s. 3] Tento přístup znamená, že transakce jsou prováděny mimo programovou logiku. Tento způsob je konfigurován, jak je již u Springu zvykem, anotacemi nebo XML soubory. [27, s. 12-13] Opačný způsob, ve kterém vývojáři vytvářejí vlastní logiku transakcí je vhodné využít pouze u menších aplikací. V takovém případě ovšem nevzniká potřeba využití příslušných Spring modulů.

Spring řešení navíc nabízí i více vylepšení. Systém automatických „rollbacků“, tedy návratů dat do stavů před zahájením transakcí, je zde připraven k vlastním specifikacím a mimo systémové výjimky funguje i při výjimkách, které vývojáři sami zachycují v kódu aplikace. [27, s. 11-12]

Další flexibilitu lze zajistit i pomocí anotace @TransactionalEventListener, se kterou je možné vyvolat libovolnou metodu před provedením transakce, po jejím úspěšném ukončení, po neúspěchu a tedy rollbacku nebo po jejím ukončení ať už úspěchem, či neúspěchem. [27, s. 60-61] Je-li tak v aplikaci při transakcích třeba

vykonat určité operace v těchto případech, Spring opět přináší velmi elegantní řešení, se kterým značně ulehčí práci při tvorbě této logiky.

4.1.6 Validace

Přístup webové aplikace k proměnlivým datům v podobě databáze většinou znamená i to, že určitá data jsou do těchto databází vkládána přímo uživateli. Z tohoto důvodu je nutné jimi vložené informace určitým způsobem ošetřit a to nejen kvůli neúmyslným chybám, ale i úmyslnému chování. Takovými ošetření se rozumí záležitosti jako je kontrola správných formátů (e-mailová adresa, telefonní číslo, adresa), ořezávání přebytečných mezer, malá a velká písmena, nepovolené znaky aj.

Validator – rozhraní nabízené Springem, které lze implementovat pro validaci jednotlivých modelů v aplikaci, ale i jejich případných potomků. [23, s. 235-237] K tomuto rozhraní je samozřejmě možné dodat již existující implementace. Mezi ty se řadí například defaultní Spring implementace, ale také validátor Hibernate. Velmi příjemným řešením validace uživatelských vstupů jsou všudypřítomné anotace. S jejich pomocí lze v datových třídách tvořících modely do databází ukládaných tříd přiřadit potřebné anotace přímo k jednotlivým vlastnostem. Tyto anotace upřesňují požadavky, kterými může být například to povinné vyplnění, určitá délka například pro hesla a mnoho dalších. Dále obsahují i chybové hlášky vracené při porušení nastavené konvence.

@Valid – označení v kontroléru třeba pro model odeslaný formulářem, které zprovozní validaci. V případě výskytu vstupu či vstupů které požadavky nesplňují je možné tuto situaci dále řešit, nejčastěji pak návratem k formuláři s chybovým hlášením. [41] K tomu je v kombinaci s anotací používána třída **BindingResult** (nebo **Errors**) obsahující případné chyby.

4.1.7 Spring MVC

Celým názvem Spring Web MVC (Model-View-Controller) je nadstavbou programového rozhraní Servlet API známého z Javy. [28, s. 2] Tento balíček používá třídy jako Jakarta Servlet k obsluze HTTP dotazů a také odpovědí na tyto dotazy. Právě proto je i samotný Spring MVC servletem. Takový servlet sice umí obsluhovat

dotazy a jejich odpovědi, ale samotný přenos řídit neumí. Proto je třeba tento kód spouštět na vhodném serveru, obvykle nazývaném kontejner, který tyto přenosy řídit umí. Takovým serverem je například Apache Tomcat.

DispatcherServlet – zadává obstarání dotazů jiným komponentám deklarovaným v konfiguračním souboru. [28, s. 2] Jedná se o typickou architekturu, kterou se řídí i Spring. Dispečer pracuje se servletovým webovým aplikačním kontextem, který přenáší informace o kontrolérech koncových bodů. Těch může být v jedné aplikaci i více a je možné, aby požadavky na danou aplikaci byly obsluhovány různými způsoby. Tyto kontexty spojené s jednotlivými dispečery jsou však spojeny s aplikačním kontextem, který je pouze jeden a obsahuje služby a také repositáře. [28, s. 4-5]

HandlerMapping – jeden z příkladů komponent, kterým dispečer deleguje práci. Pracuje s dříve zmíněnými anotacemi jako je @RequestMapping označující konkrétní koncové body. V této adrese je pak dále možné označit její část jako proměnnou, která bude přenesena do metody. [28, s. 31-32] To je vhodné v případech, kdy se tato část adresy běžně mění, třeba při přístupu na uživatelské profily, které jsou rozlišeny jistým identifikátorem (uživatelské jméno, unikátní číslo a jiné). V případě shody v adresách je pak vybírána ta delší, detailnější cesta, která s největší pravděpodobností vede dále. Parametry těchto anotací přijímají dále typ požadovaného média a také parametry samotného dotazu, které musí existovat, aby byl takový dotaz vůbec obsloužen. [28, s. 33-34]

ViewResolver – ten rozhoduje o tom, jaká stránka aplikace má být uživateli vykreslena. [28, s. 17-18] Obě komponenty tedy spolupracují. ViewResolver může stránky také přesměrovávat na jiné koncové body, které jsou opět vyhodnoceny komponentou předchozí. [28, s. 18-19] Vykreslení určité stránky může probíhat i konkrétním modelem, jakým je například ModelAndView. Tato návratová hodnota obsahuje nejen webovou stránku, ale také model, který může mít přiřazenou celou řadu atributů, které mohou obsahovat i celé třídy. [30]

Obě komponenty pracují v rámci kontrolérů obsahujících kód obstarávající dotazy na koncových bodech.. Mapování dotazů je docíleno užitím vhodných anotací závislých na typu dotazu (POST, GET, DELETE, PUT a PATCH). [28, s. 31] Kontroléru může být dotazem zaslán i model odpovídající určité třídě, který bude mít přidělen

hodnoty odeslané formulářem (příkladem může být registrace uživatele). Mimo to může takový model metoda zpracování požadavku i vytvářet a odesílat ho v odpovědi. [28, s. 70-72] Pomocí anotací je u kontrolérů možné vytvářet i vlastní logiku pro řešení výjimek. Výjimky lze řešit obecně, nicméně nejlepším řešením je specifikovat je co možná nejpřesněji. I tyto výjimky mohou navracet model s webovou stránku, nejběžněji stránku chybovou. [28, s. 74-79] Řešit je lze mnohými způsoby na úrovních jednotlivých kontrolérů, ale i na úrovni globální.

WebMvcConfigurer – základní konfigurace Spring MVC, kterou lze rozšířit vlastními pravidly, bez kterých se žádná aplikace nejspíše ani neobejde. Zmíněno bylo vytváření koncových bodů pro uživatele definicí obsluhy příslušných adres v kontrolérech. Není však úplnou pravdou, že tyto koncové body v jiném případě nemohou existovat. Lze jim totiž určit velmi základní obsluhu v podobě vykreslení konkrétní webové stránky s využitím právě tohoto konfigurátoru. [28, s. 181-182] Ten otevírá řadu možností pro specifikaci formátů, jako jsou data a čas, nastavení přípustných souborových formátů (.json, .xml, .rss, .atom) či validátorů. [28, s. 171-175] Umožňuje nastavení „interceptorů“ („zachytávačů“), které mohou vykonat určitou operaci před, nebo po vykonání obsluhy dotazu. Dále mohou tuto operaci vykonat i po dokončení celého dotazu. [28, s. 15] Za zmínku určitě stojí také připojení složek se statickými zdroji, jako jsou například kaskádové styly, obrázky, skripty a další. Soubory z těchto zdrojových složek poté nemusejí být linkovány celou cestou, nýbrž cestou, která jim bude nastavena v této konfiguraci. [28, s. 185-187]

U Spring MVC lze již dobře nahlédnout na skutečnou funkčnost anotací. Označování jednotlivých komponent anotacemi @Controller, @Service nebo @Repository nemusí mít z počátku příliš jasnou vypovídající hodnotu. U Spring MVC už ovšem dostávají svůj kontext. Na druhou stranu však Spring také vnitřně řeší problémy, kterými není nutné jeho uživatele zatěžovat. Způsob, jakým dispečer vyhledává jednotlivé komponenty a deleguje jim operace nebo srovnává a správně přiřazuje adresy koncových bodů je na úrovni vnitřní logiky a vývojáře tím není nutné zatěžovat. I zde ovšem nabízejí tvůrci náhled do útrob, má-li o to daný vývojář zájem. Vzniká zde tak kombinace velké abstrakce, pro co nejefektivnější používání, ale také vysoká míra přizpůsobení.

4.2 Spring Data

Jednou z nejpodstatnějších částí každé webové či mobilní aplikace, především pak v případě sociálních sítí, je přístup k informacím, které uživatelé nejen zobrazují, ale také sami vkládají, upravují a jinak s nimi pracují. Informace v podobě proměnlivých dat jsou součástí internetových médií, která je transformuje ze starých, statických webových stránek, tak jak byly známy především dříve, do moderních a interaktivních webových aplikací. Framework určený právě pro tvorbu takovýchto softwarových řešení tak z logiky věci velmi silně staví na podpoře různých databázových systémů. Touto problematikou se zabývá modul zvaný Spring Data.

Na rozdíl od datového přístupu, který pojednává spíše o komunikaci mezi programovou logikou a přístupem k datům se tento nadstavbový modul věnuje vytvoření abstrakce persistentní datové vrstvy v daném projektu. Jinak řečeno se jedná o rozhraní, skrze které se projekt připojuje k databázi, té skutečné persistentní vrstvě. Vrstva pro datový přístup umí v rámci Springu toto rozhraní nezávisle na specifikaci implementovat a data tak předávat pro další zpracování servisní vrstvě.

Repositář – rozhraní ve Springu nazývané anglicky „repository“ a označované stejnojmennou anotací @Repository, která již byla zmíněna. Defaultní implementaci jednoho z dostupných rozhraní závislého na používané databázi aplikace získá díky pouhému přidání tohoto rozhraní do kontextu aplikace. Předáním třídy, která má být do tabulky databáze vkládána (včetně datového typu jejího ID) získává toto rozhraní řadu řešení, aniž by musel být implementován další kód. Mezi operace dostupné pro danou třídu pak samozřejmě patří ukládání záznamů, jejich opětovné získání a mazání pomocí unikátního identifikátoru, dále také navrácení všech záznamů tabulky, počtu záznamů a také zjištění, zda daný záznam existuje. Potomkem takového rozhraní může být také další rozhraní, které umožní výsledky stránkovat pro lepší přehlednost. [31, s. 7-9] Pokud se jedná o dotazy pro vyhledání, mazání či počítání záznamů podle určitého sloupce v tabulce databáze, Spring jimi již defaultně disponuje. [31, s. 9-10] Je-li k rozhraní přidána metoda požadovaného dotazu dodržující jmennou konvenci, bude tento dotaz automaticky implementován bez nutnosti připisování dalšího kódu. [31, s. 18-19]

Pro účely těchto metod je možné i řetězení názvu pro vnořené vlastnosti. [31, s. 19] Ani tím však nekončí možnosti, kterými lze s dotazy manipulovat. Správně psaným názvem lze docílit i určitého řazení výsledků, omezení jejich počtu a dalších. [31, s. 20-21] Při vytváření metod lze využít i vhodných parametrů Pageable“ pro získání stránkovaného seznamu výsledků, který přenáší také informaci o počtu stránek, Slice, tedy stránkovanému seznamu bez údaje o počtu stránek, a Sort pro tříděný seznam. [31, s. 19-20] Pojistit lze i variantu, při které by metoda měla dostat možnost pro vracení prázdných výsledků anotací @Nullable. Toho lze dosáhnou i v Kotlinu, přestože se Kotlin jako takový snaží všem případům výskytu prázdných hodnot vyhnout. [31, s. 11-14] Další užitečnou anotací je @Async, která dotazy přeformuje na dotazy asynchronní. [31, s. 22] To znamená, že metoda nečeká na příchod výsledků a pokračuje dál. Předání výsledku pak proběhne v dalším vlákně aplikace.

Možnosti pro začlenění vlastních dotazů bez nutnosti psaní potřebné logiky těchto dotazů jsou tak ve Springu skutečně bohaté a v případě, že by ani tato škála různých přizpůsobení vývojáře neuspokojila, je samozřejmě možné vytvářet dotazy zcela od základů.

Ta základní rozhraní, nad kterými je možno tvořit rozhraní vlastní jsou Repository, CrudRepository a PagingAndSortingRepository, která (v tomto pořadí) také sama sebe rozšiřují a umožňují tak tvořit vlastní požadovanou úroveň abstrakce nad databází. [31, s. 11] I to poslední ovšem může mít dalšího potomka, například MongoRepository, rozhraní specifické pro práci s NoSQL databází MongoDB, pro kterou má Spring Data vlastní modul.

Užitím určitého modulu budou všechny repositáře projektu laděny pro právě tuto konkrétní databázi, nicméně Spring má pojištěny i případy, ve kterých je nutné použít více různých modulů Spring Data pro rozdílné databáze. V takovém případě je třeba vytvořit potomky rozhraní specifických databází, aby je samotný framework zvládl rozeznat. Alternativně Spring dokáže vyčíst informaci o typu databáze, které je rozhraní určené z vložené třídy, která je doplněna o anotaci, jež může být specifická pro určitý modul, například právě MongoDB. Její speciální anotaci pro třídu vkládanou do tabulky je @Document. [31, s. 14] V poslední řadě je možností i vlastní konfigurace XML souborem, nebo příslušnou anotací. [31, s. 15-17]

Vytváření instancí tříd zamýšlených pro zaplnění tabulek databáze se podle kapitoly 3.1 dokumentace [44] řídí určitými pravidly, která praví, že upřednostňován je konstruktor nepřijímající žádné parametry, následovaný konstruktorem s parametry a v případě vícero konstruktorů to bude ten, který je označen patřičnou anotací. Důvodem k takovému postupu je drobné zrychlení programu při vkládání hodnot do jednotlivých vlastností. U velkých aplikací jsou i takto malé optimalizační prvky důležité pro plynulý chod, který všichni jejich uživatelé zajisté značně ocení. V případě nevyplnění všech hodnot již při vytvoření instance třídy se i nadále postupuje s optimalizačními pravidly, podle kterých je upřednostněno vytvoření nové instance s vkládanou hodnotou skrze vnitřní metodu a až poté využití setterů. Pokud ani jedna z variant není možná, vytváří Spring kopii třídy s novou hodnotou, jak je popsáno v kapitole 3.2 dokumentace. [44] V rámci optimalizace pak tvůrci sami v kapitole 3.3 [44] doporučují postupy, které jdou v souladu s těmito pravidly a doporučují také využití projektu Lombok. [31, s. 47]Lombok, který již byl dříve zmiňován v kapitole o programovacím jazyku Kotlin, vlastně plní stejnou funkci, jako právě Kotlin. Znovu je ovšem dobré připomenout, že elegantním řešením pro tvorbu datových tříd, jejichž výhody byly rovněž dříve popsány, disponují už i novější verze Javy.

Modul Spring Data se dělí do mnoha různých dalších modulů, které se zaměřují na různé databázové technologie. Mezi ty nejznámější patří například Spring Data JDBC, Spring Data JPA a Spring Data MongoDB. [33] **JDBC** (Java Database Connectivity) je tou nejzákladnější vrstvou, která obsahuje v podstatě především ovladače připojení k databázi, avšak samotnou tvorbu dotazů již ponechává na vývojáři. **JPA** lze vnímat jako jakousi nadstavbu, která již zaujímá určitý postoj k tvorbě dotazů jako takových, které jsou v tomto případě už před vývojářem schovány. Jak již bylo zmíněno v předchozí části, implementací specifikace JPA je například velmi oblíbený framework Hibernate. Kromě klasických SQL databází Spring přidává podporu i pro databáze typu NoSQL jako jsou Redis, projekt který je navíc ozvláštněn svou strukturou databáze ukládané v paměti, Apache Cassandra, zmíněnou MongoDB, ale také jiné technologie jako například REST, což není modul databáze jako takové, ale spíše nástroj pro tvorbu REST API a mnoho dalších. [33]

4.2.1 Spring Data MongoDB

Všechny tyto moduly jsou samy o sobě rozsáhlými tématy, nicméně v rámci ukázkové aplikace bude možné se setkat s databází MongoDB a právě proto by bylo vhodné tento konkrétní modul podrobit bližšímu pohledu. Tato databáze se těší vysoké popularitě a není s velkým předstihem jen nejoblíbenější NoSQL databází. Její popularita dosahuje těch nejčastěji užívaných řešení vůbec a v současné době se tak staví na 5. místo v žebříčku popularity vedle databází jako PostgreSQL a Microsoft SQL Server. [34]

Data – v NoSQL databázích jako je právě MongoDB jsou ukládána v dokumentech, v tomto případě ve formátu JSON, bez nařízených strukturovaných relací a vztahy mezi jednotlivými tabulkami jsou tak mnohem flexibilnější a zároveň také netrpí určitou úrovní restrikce. To vše na rozdíl od SQL databází, které jsou známé také jako databáze relační, řídících se určitým schématem, dle kterého jsou data v takové databázi uložena a spojena relacemi, jak už ostatně z názvu vyplívá.

Identifikátory záznamů – v MongoDB jsou reprezentovány vlastností třídy, která je buď pojmenována jako „id“ nebo má také příslušnou anotaci. [35, s. 135] Typická je kombinace obou způsobů. Určitou konvencí se řídí i datové typy, které jsou jako identifikátor používány. Tradičně je tak možné setkat se s ID typu String, Integer nebo také šedesáti čtyř bitovou číselnou hodnotou Long, která pojme několikanásobně více záznamů, než Integer. V případě, že ani takové množství hodnot nedostačuje požadavkům databáze, lze přistoupit i k datovému typu BigInteger. Možností je také použití typu ObjectID, který je výchozí hodnotou při automatickém vytváření identifikátoru (v případě, že žádné ID není modelem poskytnuto) a zároveň je možné vlastní ID na tuto hodnotu do databáze mapovat. [2, s. 315] V případě nutnosti však může být ukládáno i jako textový řetězec. [35, s. 64-65]] Mimo ID je automaticky vkládána i informace o třídě, která záznam v tabulce představuje. Záznamy v databázi je tak možné vyhledávat i podle třídy. [35, s. 65] Nutno podotknou, že u MongoDB neexistuje automatická inkrementace identifikátorů a i z tohoto důvodu je lepší v každém případě zahrnout nastavitelné ID v modelu třídy. Standardním postupem je pak řešení inkrementace programově,

přičemž současná nejvyšší hodnota všech řad se ukládá v samostatné tabulce databáze.

Dotazy – jak již bylo řečeno dříve, moduly projektu Spring Data umožňují tvoření celých dotazů pouhým dodržením konvence názvu metody a ani u modulu zaměřeném na MongoDB tomu není jinak. Vhodná klíčová slova, která lze k tvorbě těchto dotazů využít, tak nabízejí možnosti jako například porovnání nerovností, vyhledávání v číselném rozmezí, vyhledávání prázdných hodnot, regulární výrazy, ale také prostorové dotazy, které vyhledávají dle určitých geometrických dat jako je například vzdálenost dvou míst aj. [35, s. 76-79] Řazení výsledků pak lze dosáhnou stejným způsobem. [35, s. 22-23] Vyhledávat lze dále také s využitím výrazů již diskutovaného jazyku SpEL. [35, s. 142-143] Nechybí ani funkce známá jako fulltextové vyhledávání, které s využitím požadovaného speciálního textového indexu vyhledá shody v celé databázi. [35, s. 82-83] Podporován je také „jazyk" pro práci s dotazy zvaný Query by Example, nabízející možnost tvoření dotazů, které se nezaměřují na specifický sloupec záznamů. [35, s. 86-87] V podstatě se jedná o způsob, díky kterému lze předložením určité šablony (vyhledávaných informací) dosáhnout stejného výsledku jako u klasických dotazů, nicméně bez nutnosti tvoření jejich logiky.

Konfigurace – ve Springu práce s MongoDB začíná začleněním závislosti do konfiguračního souboru, čímž bude projekt automaticky obohacen nejen o modul samotný, ale i jeho další závislosti, jedná-li se o projekt založený pomocí Spring Boot.

AbstractMongoClientConfiguration – ve starších verzích známá jako AbstractMongoConfiguration, je třída, jejímž rozšířením konfigurace probíhá. K databázi jako takové se modul připojí s využitím konfigurační metody vracející třídu **MongoClient**, nebo konfiguračního XML souboru. [35, s. 57] Mimo jiné umožňuje i povoluje indexování dat databáze a vytváří defaultní @Bean pro třídu MongoTemplate.

MongoTemplate – třída obstarávající nejen mapování datových tříd do tabulek databáze, ale poskytující také řadu metod pro databázové dotazy. [35, s. 58-59] Automatické tvoření dotazů je sice skutečně obsáhlé, nicméně i tak může být limitující. Právě proto třída MongoTemplate nabízí operace, které připouštějí určitou volnost dotazování nad databází a zároveň vytvářejí jistou úroveň abstrakce,

neboť není třeba vytvářet dotazy jako takové. I zde lze volat metody také nad celými kolekcemi, lze je vytvářet, mazat a dále je poskytnuta i metoda, která navrátí názvy všech tabulek. [35, s. 114] Stejně jako ke kolekcím pak lze přistupovat i ke všem indexům dané databáze. [35, s. 114]

MongoOperations – rozhraní, které MongoTemplate implementuje. To přijímá také datové třídy, kdežto jeho defaultní implementace přijímá pouze dokumenty, na které již třídy byly namapovány. [35, s. 58-59] Je-li i tato míra abstrakce nežádoucí, vývojářům je umožněno vytvořit vlastní implementaci. Implementace vyžadují třídu MongoClient, která obsahuje údaje potřebné k přihlášení k databázi. V případě XML konfigurace je i třída MongoTemplate konfigurována v tomto XML dokumentu. [35, s. 59-60]

EntityCallback – metoda, která je automaticky volaná v různých částech průběhu databázových dotazů, tedy před, či po uložení a dále také před, nebo po převedení datové třídy na dokument. Pro implementaci těchto metod nabízí Spring již tradičně vlastní rozhraní popsané v kapitole 11.16 dokumentace. [45] Funkci těchto metod lze aplikovat v různých případech, typicky se nabízí například přeformování hesla z čitelného textu pomocí hašovací funkce.

Sessions – rozšiřující funkce MongoDB pro vytvoření konzistentního spojení. U aplikací obecně je velmi důležité vytvořit co možná nejvyšší úroveň komfortu pro jejich uživatele a nezatěžovat je zbytečnými operacemi. Jelikož velké množství funkcí aplikací velmi často vyžaduje určitý stupeň autorizace, uživatelé jsou neustále povinni svou úroveň autorizace vykazovat. Právě proto existují „sessions“ neboli relace, které zajišťují automatizaci procesů, které jsou závislé na procesech jiných. Namísto neustálého vyžadování autentizace skrze přihlašovací údaje, na jejichž základě může aplikace udělit autorizační práva, se tato práva a další informace o uživateli dají na stanovenou dobu uložit do databáze. Záznam relace v databázi má pak vlastní identifikátor, který je u uživatele uchován v cookies v jeho prohlížeči. MongoDB tak nabízí pro tyto relace infrastrukturu, kterou je možné skrze vhodné třídy ve Springu ovládat. Popsány jsou v dokumentaci [45] v kapitole 12.

Nejspíše všechny tyto funkce modulu je možné zařadit do očekávaných. Přestože rozebrány byly opět především části, se jejichž většinou se setká každý vývojář, který se pro MongoDB ve spojení se Spring frameworkem rozhodne,

nabízena jsou i další řešení pro komplexnější úkoly. Nechybí tak hromadné zpracování dat, ukládání skriptů psaných v jazyce JavaScript přímo do databáze a mnoho dalších funkcí. [35, s. 49]

4.3 Spring Security

Žádná aplikace v Internetové síti se neobejde bez určité úrovně ochrany. Důležitá je nejen ochrana samotné aplikace, ale především ochrana uživatelů tuto aplikaci používajících a jejich dat. Tomuto komplexnímu tématu se věnuje modul Spring Security, který nejen že poskytuje infrastrukturu pro autentizaci a autorizaci uživatelů, ale také ochranu proti různým druhům útoků, kterým mohou být v rámci užívání především webových aplikací vystaveni. Vzhledem k popularitě Springu jako takového je i popularita Spring Security značná a je pravděpodobné, že většina vývojářů bude kombinovat hlavní modul frameworku právě i s tímto modulem zabezpečení.

Pomocí zabudovaného systému filtrů modul Spring Security zachytává veškeré příchozí požadavky, které aplikace obdrží. [36, s. 52-53] Mezi ty patří například filtry proti různým druhům útoků ze třetích stran, OAuth filtry (předávání práv k datům uživatele třetím stranám), session filtry, autorizační filtry a mnoho dalších. [36, s. 53-55] V případě příchozího požadavku je nutné ověřit, zda má jeho odesílatel oprávnění potřebná k jeho zpracování. Některé požadavky, mezi které by mohlo patřit například zobrazení úvodní stránky nebo také registrační a přihlašovací stránky, budou s největší pravděpodobností přístupné všem uživatelům nezávisle na tom, zda nějaká práva v dané aplikaci mají, či nemají. V takovém případě se jedná o návštěvníky bez uživatelského účtu, nicméně pro přístup k jiným funkcím a stránkám aplikace je z velké části vyžadováno přihlášení uživatele pro přidělení příslušných práv v rámci webové či mobilní aplikace a jiných softwarových řešení. K tomu je využíváno především dvou hlavních konceptů autentizace a autorizace.

4.3.1 Autentizace

Autentizace je způsob, jakým je ověřována totožnost uživatele, klasicky známý jako přihlášení do aplikace, při kterém se uživatel prokáže určitým

přihlašovacím jménem (například emailovou adresou nebo přezdívkou) a heslem. [36, s. 13] Speciální třídy vyhodnocují požadavek na autentizaci uživatele a také automaticky vyžadují zaslání takového požadavku v případě neoprávněného přístupu ke koncovým bodům aplikace nepřihlášeným uživatelem. [36, s. 68] V případě shody všech vyžadovaných informací o totožnosti se záznamem v databázi je uživatel považován za autentizovaného. Ve Springu pak bývá takto ověřený uživatel většinou znám pod anglickým označením „principal“. [36, s. 65]

SecurityContextHolder – třída ukládající informace o ověřeném uživateli, díky které mají metody vždy k těmto informacím přístup. Pro jejich uložení slouží ve Springu speciální autentizační token. V případě neúspěchu při autentizaci je tato třída vyčištěna. [36, s. 70-73] Takové chování je defaultní, nicméně u zavedení vlastní konfigurace pro webový konfigurační adaptér zabezpečení je třeba jej také specifikovat. [36, s. 78]

Autentizace jako taková je pak ve Springu z velké části automatizována. Vývojářům stačí použít defaultní, nebo vlastní přihlašovací formulář a užívat správnou jmennou konvenci, která je velmi intuitivní. [36, s. 73-76] Poté je nutná, alespoň v kontextu použití společně s databází MongoDB, implementace několika metod rozdělených do dvou připravených rozhraní, které mají zajistit vyhledání konkrétního uživatele a přiřazení jeho rolí, či jednotlivých práv.

UserDetailsService – jedno z těchto rozhraní, pro které existuje i defaultní implementace pro JDBC. [36, s. 90-91] O ostatní se již framework stará sám, což opět vede ke vzbuzování pocitu silné míry abstrakce. Kromě autentizace spojené s databází je nabízena i autentizace uživatelů v paměti, ovšem této možnosti by mělo být využíváno výhradně v rámci ladění aplikace. [36, s. 81-83] I pro tento případ existuje defaultní implementace rozhraní obstarávajícího autentizační objekt uživatele. [36, s. 90-91]

4.3.2 Autorizace

Pod autorizací si lze představit určitá práva, která jsou tímto způsobem ověřenému uživateli přiřazena. Mezi taková práva může patřit třeba přístup k částem aplikace, které jsou nepřihlášeným uživatelům nepřístupné, přidávání a upravování obsahu a další. Udělovat takové množství práv jednotlivě je ovšem

značně nepraktické a proto je často využíváno rolí, které jsou ve svojí podstatě určitou skupinou dostupných práv. Přidělováním rolí tak lze uživatelům uvolňovat celé soubory práv. [37] U rolí zároveň existuje i možnost stanovení hierarchie, ve které nadřazené role, například administrátorské role, dědí i práva z rolí podřazených, například uživatelských. Nabízí se tak jeden z více způsobů jak řešit problém více rolí, jehož alternativou by bylo přidání všech potřebných rolí danému uživateli aplikace. [36, s. 152] Tento přístup je využívaný napříč celým spektrem informačních technologií, od webových aplikací až po informační systémy obecně.

GrantedAuthority – je ve Springu objekt určený k reprezentaci těchto práv, který navrací jednotlivé uživatelské pravomoci v podobě textových řetězců. Jeho Springem nabízenou defaultní implementací která je v aplikacích běžně používána je pak **SimpleGrantedAuthority**. [36, s. 106-107]

S následným využitím výrazů dříve zmiňovaného jazyku SpEL pak lze různými způsoby vyhodnocovat, zda by měly být různé funkce určitému uživateli přístupné, či nepřístupné. Připravené výrazy ve Springu čítají možnosti jako je dotázání, zda je uživatel přihlášený, jaká je jeho úroveň autorizace, jaké jsou jeho role a jiné, mezi něž patří i výrazy, díky kterým lze všechny přístupy povolit, nebo zakázat. [36, s. 156-158] Poslední ze jmenovaných možností bývají často využívané pro velmi základní úroveň nastavení, například přístup k přihlašovací stránce by měl mít dostupný nezávisle na tom, kdo k takovému koncovému bodu přistupuje. Jak je již u tohoto frameworku zvyklostí, samotné nastavení zabezpečení jednotlivých adres lze provádět rozdílnými způsoby a to XML souborem nebo také konfigurační třídou. [36, s. 158-162] Naopak v případě omezení přístupu k jednotlivým metodám jsou tato pravidla řízena anotacemi, do kterých se požadovaný výraz vkládá formou parametru. Rozdíl mezi anotacemi vzniká v době kontroly a to většinou před voláním metody za pomoci anotace @PreAuthorize, ale možností je i provedení kontroly práv uživatele po vykonání volané metody anotací @PostAutorize. Kromě těchto dvou anotací existují i možnosti další a to @PreFilter a @PostFilter sloužící k názvem napovídanému filtrování různých druhů polí. [36, s. 162-168] V kombinaci s vhodnou anotací pro zapnutí ochrany metod třídy stejnému účelu poslouží i anotace @Secured, která ovšem místo SpEL výrazů přijímá jako parametr název určité role v podobě textového řetězce. [36, s. 172-173]

4.3.3 Zabezpečovací mechanismy

Spring Security nabízí velmi široké možnosti zabezpečení aplikace a některé z nich budou v tomto krátkém výpisu popsány. Mimo to, že zvládne pokrýt nároky i velkých aplikací a ulehčuje především malým vývojářům značnou míru práce na složitém zabezpečování jejich softwaru díky své připravenosti za pomoci jednoduchých konfigurací a „out of the box“ (předpřipravených) řešení, které okamžitě nastartují ochranu aplikace. Velmi základním nastavením vyžadujícím autentizaci pro uživatelský přístup tak disponuje už po pouhém přidání modulu Spring Security do závislostí projektu. Praktické přiblížení využití diskutovaných prvků zabezpečení bude stejně jako v případě ostatních modulů Spring frameworku dále probráno v následující kapitole, která je zamřená na tvorbu ukázkové aplikace.

4.3.3.1 Hesla

Způsob, který Spring pro ochranu dat uživatele nabízí je vcelku prostý a standardní. Přestože přihlašovací jména budou ve většině případů nejspíše ukládána v takzvaném „plain textu“ (prostém textu), hesla jsou naopak, alespoň v přáních všech uživatelů, ukládána ve formě šifrované. Doby kdy i hesla samotná byla ukládána v neupravené podobě jsou již naštěstí dávno pryč a přes prvotní šifrovací funkce, které byly později nahrazeny funkcemi využívajícími kromě holého hesla také šifrovací sůl, se tato disciplína dostala až do bodu, kde jsou již nedostačující kryptografické hašovací funkce nahrazovány modernějšími nástupci mezi které patří Argon2, bcrypt, scrypt a další. [36, s. 13-14] Tyto funkce účelně pracují v řadě opakování. To sice zapříčiní vcelku pomalé ověřování hesel uživatelů, nicméně to také učiní hesla dnešní technikou neprolomitelná. Existuje však spousta dalších algoritmů, které jsou sice zastaralé, ale databáze stále mohou obsahovat hesla jimi šifrovaná. Z toho důvodu Spring disponuje třídou speciálního enkodéru, kterému lze přidělit různé implementace hašovacích funkcí. Ten pak deleguje úkol srovnání hesla uživatele se záznamem v databázi vhodné funkci, kterou pozná podle identifikátoru připojeného k šifrovanému heslu. [36, s. 14-16]

4.3.3.2 Ochrana proti útokům

Kromě systému ověřování totožnosti uživatelů a udělování potřebných práv disponuje Spring i tou zmíněnou ochranou proti útokům ze třetích stran. Je zde tak dostupná ochrana proti útokům známým pod anglickým názvem „Cross site request forgery“. Takový útok zneužívá dat uložených v cookies uživatele pro vytvoření falešného dotazu. Vzhledem k tomu, že požadavek ze suverénní, ale také podvodné stránky bude mít stejnou podobu, je nutné je od sebe určitým způsobem rozlišit. Standardním řešením tohoto problému je využití unikátního tokenu a přesně takové řešení volí i tvůrci Springu. Tento token je automaticky vkládán do požadavku odesílaného serveru, který jej porovnává s očekávanou hodnotou, kterou cizí stránka nemůže poskytnout, protože nepochází z tohoto serveru. Dalším způsobem je také využití speciálního atributu pro cookies, který nedovolí své odeslání z jiné stránky. [36, s. 25-28] V kombinaci s velmi populárním šablonovým enginem Thymeleaf je pak implementace ochrany proti tomuto druhu útoku velice jednoduchá, neboť využitím speciální syntaxe tohoto enginu je po správné konfiguraci do všech formulářů (s výjimkou toho přihlašovacího) automaticky vkládán token ve skrytém poli. [38] Tuto ochranu je pak doporučováno použít pro všechny HTTP dotazy.

Dále Spring nabízí i podpůrné funkce pro zabezpečené komunikační protokoly HTTPS. Mimo to však bezpečnostní modul frameworku podporuje i hlavičku typu HSTS (HTTP Strict Transport Security), která brání uživatele před útoky známými jako „Man in the middle“ (muž uprostřed) a je také Springem nastavena i defaultně. [36, s. 39] Takový útok zneužívá vynechání specifikace zabezpečeného protokolu z adresy URL při přístupu na určitou webovou stránku, díky kterému může tento požadavek přesměrovat. Proti stejnému útoku chrání a další podporovaný typ hlavičky HPKP (HTTP Public Key Pinning). Jiné útoky proti kterým Spring Security také chrání čítají metody jako je „clickjacking“, při kterém uživatel kliknutím na zdánlivě neškodný maskovaný element ve skutečnosti kliká na obsah útočníka, a také útoky vedené vložením útočníkova skriptu do jinak neškodné stránky známé jako „cross site scripting“. [36, s. 32-39]

4.3.3.3 OAuth

Ne vždy však musí být přítomnost jakési třetí strany jen bezpečnostním rizikem. Součástí Spring Security je i modul OAuth 2.0, díky kterému je možné registrace a následné přihlášení (tedy autentizaci) delegovat třetí straně, také označované slovem provider. Každý se určitě setkal s nějakou webovou aplikací, která nabízí možnost přihlášení skrze účet Google, Facebook a jiné bez nutnosti vytváření nového účtu.

Právě tento způsob ověřování totožnosti nabízí i framework Spring, který dokáže spolupracovat s již zmíněným ověřováním společnosti Google skrze nadstavbu OAuth 2.0 zvanou OpenID Connect 1.0. Zprovoznění tohoto mechanismu pak probíhá vcelku jednoduchým přesměrováním stránky na požadovanou adresu a nastavením konfiguračního souboru. [36, s. 181-182] V konfiguračním souboru pro časté poskytovatele této funkce jako je například právě Google stačí opravdu jen minimální míra nastavení, protože Spring disponuje defaultními nastaveními. [36, s. 183-184] Jako u spousty systémů tohoto frameworku je tak jeho prvotní spuštění velmi rychlé a efektivní a poskytuje uživatelům aplikace další funkcionalitu, která je velice oblíbená především proto, že urychluje jejich užívání aplikace a také je nenutí vytvářet další z množství různých uživatelských účtů. I zde lze však defaultní nastavení funkčnosti změnit dle potřeby za využití konfigurační třídy, která díky správným anotacím umožní přepsat Springem určený výchozí stav. [36, s. 189-190] V pokročilejším nastavení pak nechybí možnosti jako je změna přihlašovací stránky, přesměrování po úspěšném ověření uživatele, integrace JSON Web Tokenu a také konfigurace přihlašování a odhlašování a další. [36, s. 192-203] Kromě JWT, známého i z velké řady jiných prostředí a programovacích jazyků od C po Python, se nabízí také využití takzvaného „opaque“ (neprůhledného) tokenu. [36, s. 280] Ten se od JWT liší především tím, že je jeho formát neznámý. Nepovolané strany tak nemohou bez informace o formátu tohoto tokenu přečíst jeho obsah. Těmto tokenům je po zpracování přidělena úroveň autorizace a celkový objekt je pak vkládán do třídy SecurityContextHolder stejně tak, jako tomu bylo u autentizace bez JWT. [36, s. 284-285] [36, s. 319-320]

4.3.3.4 Reprezentace uživatele

Uživatel jako takový je reprezentován objektem, který o něm uchovává určité informace, mezi které patří samozřejmě identifikátory, jméno, způsob ověření uživatele a k tomu potřebné údaje jako je například heslo, speciální token a identifikátor takového tokenu a další. Speciální objekt zastupuje nejen registraci uživatele, ale i jeho autentizaci (tedy ověřeného uživatele). Obě třídy samozřejmě disponují vlastními repositáři v databázi. [36, s. 225-227] Způsob řešení je tak v podstatě stejný, jako například u uživatelské relace. Rozdílné řešení požadavků pro autorizaci uživatele Spring nabízí pro dva typy klientů a to klientů důvěrných a veřejných. [36, s. 237-238] Tím vývojářům ušetří práci v podobě ochrany proti útokům, které odchytávají přenos dat, tedy dalšímu z řady nežádoucích zásahů třetích stran do běhu aplikace.

4.4 Spring Boot

Spring Boot – slovní spojení, které v této kapitole zaznělo již několikrát. Spring framework jako takový je pro svou obsáhlost i celkem složitý pro počáteční nastavení a rozběhnutí nového projektu, ať už je to z důvodu připojování závislostí projektu či jiných. Nový vývojář začínající s tímto Java frameworkem tak může nabýt nepříjemného pocitu, že takové spuštění bude velice časově náročné a obtížné. Z tohoto důvodu tvůrci Springu přišli s řešením, které zařídí start nové aplikace s jen velice málo kroky a to díky již připraveným defaultním konfiguracím a implementacím, ale také dalším systémům jako jsou zabezpečení, metrika aplikace, vestavěné servery implementující HTTP protokol jako je například Apache Tomcat a další. [39, s. 6-11] Protože ta defaultní nastavení jsou koncipována určitým způsobem, který samozřejmě nemusí vždy každé aplikaci vyhovovat, je možné výchozí konfigurační třídy pomocí anotací v hlavní třídě projektu vypínat. [39, s. 28-29] Jak již bylo naznačeno v předchozích částech kapitoly, možností je i definice vlastních tříd pro nastavení aplikace s využitím anotace @Configuration. [39, s. 27-28]

Stejně jako všechny ostatní moduly a rozšíření projektu je Spring Boot k aplikaci připojen jakožto závislost zapsaná v konfiguračním souboru správcovských nástrojů jakými jsou Apache Maven a Gradle. S inicializací celého

projektu pomůže speciálně vytvořená webová aplikace Spring Initializr. V té se pomocí jednoduchého formuláře zvolí výběr právě správcovského systému (Maven, nebo Gradle), verze Spring frameworku, metadata projektu a požadované rozšiřující moduly v podobě konfigurace závislostí. [40] Naprosto stejnou funkcí disponují i některá vývojová prostředí jako například IntelliJ IDEA společnosti Jetbrains, která stojí za vznikem Kotlinu. S již vloženou anotací hlavní třídy @SpringBootApplication je automaticky zapnuta auto-konfigurace, vyhledávání komponent (jakými jsou například repositáře, služby a další) a na závěr také vyhledávání konfiguračních tříd. [39, s. 30-31] Takto založená Spring Boot aplikace je pak lehce spustitelná především přímo ve vývojovém prostředí, ale může být také kompilována do spustitelného souboru. [39, s. 32]

Naprostou samozřejmostí je i logování průběhu programu, které je možné sledovat v terminálu přímo ve vývojovém prostředí, ale také jej lze exportovat do externího souboru. Zobrazování informací, varování, chyb a dalších náležitostí je velmi užitečné především při ladění softwaru a i Spring Boot má svůj vlastní logovací modul, který množství obdržených informací dále rozšiřuje. [39, s. 96-98] Přestože tato funkcionalita jistě nikoho nepřekvapí, výhodou je opět její připravenost od samotného založení Spring Boot projektu bez nutnosti jakéhokoliv dalšího nastavení, přestože i tuto možnost také nabízí.

Automatickým nastavením díky Spring Boot disponují i další části jakými jsou například podpora asynchronních funkcí využívající proměnlivý počet vláken nebo také dříve zmíněné uživatelské relace pod taktovkou určitého databázového systému, v tomto případě by se jednalo o MongoDB. [39, s. 225-226] [39, s. 228-229] I ty použitelné webové servery jako je Tomcat dostanou potřebná automatická nastavení. [39, s. 271-272]

Modul určený k co nejrychlejšímu zprovoznění nového Spring projektu tak očividně potvrzuje svůj účel výčtem automatických konfigurací spousty dalších modulů, včetně i těch nezmíněných, nebo naopak probraných detailněji, jako jsou Spring Data a Spring Security. Velká většina vývojářů již v dnešní době k vývoji Spring aplikací přistupuje právě skrze modul Spring Boot, který pomohl již tak velmi populárnímu frameworku ještě zvýšit svou popularitu díky ulehčenému přístupu ke všemu, co může jeho uživatelům nabídnout.

5 Návrh a tvorba ukázkové webové aplikace

Přiblížena již byla témata Kotlin a Spring. Obě tyto technologie budou využity pro tvorbu ukázkové aplikace, na které by mělo být vyzkoušeno, zda je jejich kombinace vhodná. Užití Springu v oblasti webových aplikací jistě není ničím neobvyklým, nicméně většinou se tak děje za kombinace s jazykem Java. Z informací, které jsou již o Kotlinu dostupné, a byly částečně rozebrány v této práci, je možné očekávat, že by ani zde neměly vznikat výrazné problémy při tvorbě ukázkového projektu. Jak se programovací jazyk společnosti Jetbrains projeví v implementaci kódu přiblíží právě tato kapitola.

V úvodní části předchozí kapitoly bylo zmíněno, že díky rozsáhlosti a značné modularitě Spring frameworku není nutné a je spíše nemožné využití veškerých dostupných modulů pro účely ukázkové aplikace. Proto je testování v tomto projektu zaměřeno především na ty části, které budou používány nejběžněji, kromě samotného jádra frameworku tedy také moduly pro práci s databázemi, velmi oceňovaný modul pro zabezpečení aplikací Spring Security a bootstrapovací modul Spring Boot (nástroj sloužící pro nastartování projektu).

5.1 Podoba aplikace

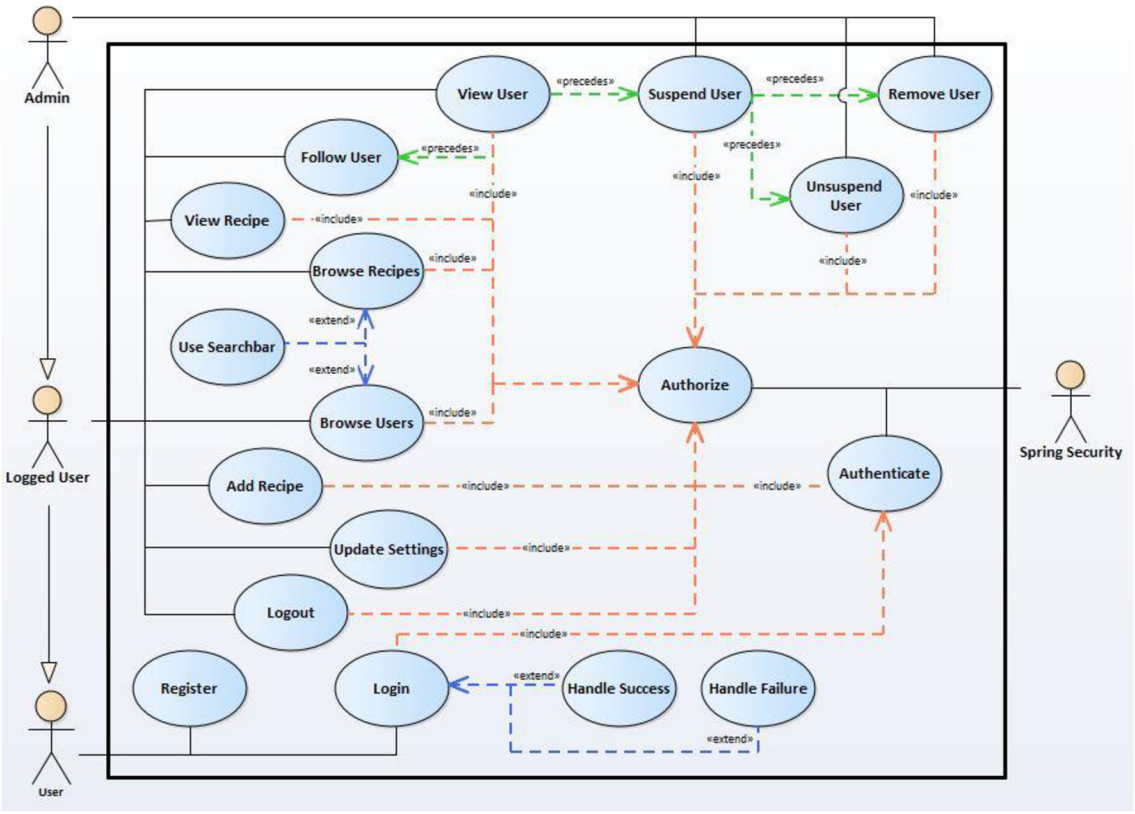
Dnešní doba je se sociálními sítěmi spojena na všech úrovních a svým rozsahem tyto sítě zahrnují vše od osobních profilů jednotlivců až po marketingové kampaně mezinárodních korporací a politických organizací. Jejich podoby se různí a uživatelům je tak nabízena značná variace serverů pro tvorbu jejich internetových profilů. Není určitě výjimkou, že lidé často využívají sociální sítě i k zálibě, která spojuje, troufalým tvrzením, úplně všechny. Touto zálibou je samozřejmě jídlo. A přestože uživatelé sociálních sítí hojně sdílejí své kulinářské zážitky na odiv ostatním, zatím nejspíše nevznikla sociální síť s tímto konkrétním zaměřením. V některých případech lidé vytvářejí vlastní kulinářské blogy, nicméně z již zmíněných důvodů se jako zajímavé a možná i zábavné téma k prozkoumání při tvorbě ukázkové aplikace k této práci jeví právě tato tématika.

Ukázková webová aplikace pro tuto práci tedy bude mít podobu webu, který by měl splňovat základní náležitosti sociální sítě, v tomto konkrétním případě se zaměřením ne tak úplně na její uživatele, ale spíše na to, čím se tito uživatelé stravují.

5.1.1 Funkce

Těmi nejzákladnějšími funkcemi, které by uživatel takové sítě mohl očekávat je vytváření a správa uživatelských profilů a samozřejmě i přidávání a mazání příspěvků, v tomto případě receptů. Nechybí ani možnost sledování uživatelů, jejichž recepty se pak zobrazují na hlavní stránce aplikace. V seznamech receptů i uživatelů lze také vyhledávat.

Protože aplikace mají jistý uživatelský vstup, bude ve většině případů nutná i možnost jej nějakým způsobem spravovat. K takovému druhu zásahu je potřeba nějaký důvod a jednat se může třeba o porušení zásad aplikace nebo také o přístup nechtěným typem „uživatelů“, jakými jsou automatizovaní internetoví boti. Z tohoto důvodu je vhodné vybudování i nadstavbové vrstvy aplikace, která bude zaměřená na administrátorské úkony. Tímto způsobem je případná starost o provádění určitých akcí nad informacemi vloženými uživateli aplikace převedena na dedikovaného správce webu, namísto toho, aby tuto funkci musel zastávat správce databázového systému. K takovému rozdělení mezi uživatele a administrátory dojde díky zmíněným pomocným rolím a vše bude implementováno za pomoci probranému modulu Spring Security. Pro rychlé znázornění těchto funkcí slouží následující diagram užití.



Obrázek 1: Diagram užití aplikace

Zdroj: Autor

5.1.2 Architektura

Z funkcí vyplívá, že aplikace bude zcela určitě disponovat databází, nad kterou bude mít každý návštěvník aplikace do jisté míry možnost provádět CRUD operace (tedy Create – vytváření obsahu a jeho ukládání do databáze, Read – čtení obsahu z databáze a jeho zobrazení na frontendové části aplikace, Update – úprava již existujícího obsahu v databázi a Delete – mazání obsahu z databáze). Tyto operace budou poskytovány skrze API (Application Programming Interface). Takové API je většinou stavěno dle určité architektury a ani zde tomu nebude jinak, nebude to však úplně přesně zmiňovaný CRUD, ale formát jemu velice podobný.

Rozhraní se tak bude v tomto případě řídit architekturou REST (Representational State Transfer), která je také zaměřená na datovou komunikaci. To znamená, že v případě požadavku na určitý přístupový bod aplikace bude komunikace mezi klientem a serverem probíhat pomocí dat uspořádaných do nějakého formátu. Architektura se vyznačuje klíčovými metodami GET a POST dříve

zmiňovanými v sekci 4.1.2 o anotacích Spring frameworku. Dalšími dvěma HTTP metodami REST architektury jsou DELETE pro mazání dat a také PUT sloužící pro úpravu dat, která ale na rozdíl od ekvivalentní operace architektury CRUD také vkládá nová data, není-li dotazovaný záznam v databázi dostupný. Oproti obyčejnému CRUD schématu má REST i další požadavky, například „statelessness“ (požadavky jsou bez stavů).

Kromě záznamů o uživatelích a jejich příspěvcích bude databáze obsahovat i pomocné tabulky, které mohou uchovávat různé informace, například role, díky kterým mohou být jednotlivým uživatelům přiřazena jejich autorizační práva v celých balíčcích, ale také hodnoty sekvencí pro všechny ostatní záznamy databáze. Určité části záznamů je dále možné rozložit na další menší části.

Databáze samotná bude implementována pomocí NoSQL databáze MongoDB, jak je již zřejmé z předešlé kapitoly. Důvodem k výběru tohoto přístupu namísto klasických SQL databází je především velmi snadná výstavba databáze, která není omezena žádnými složitými relacemi, které mohou značně komplikovat případné potřebné úpravy na sktruktuře databáze. Takové úpravy je jednoduché zavést a případné problémy se objeví na straně aplikace, kde jsou řešeny v kódu programátorem na rozdíl od úprav SQL databáze, ve které by se změny zaváděly úpravou databázového skriptu. Takové řešení se tedy zdá být přijatelnější s ohledem na zaměření této bakalářské práce. Ve skutečné aplikaci tohoto formátu pak NoSQL databáze často nabízejí i vyšší výkon při práci s velkým objemem dat.

Vynechat nelze ani nějakou grafickou podobu celého projektu. Protože se jedná o projekt zaměřený především na přiblížení backendové části aplikace, bude frontend řešen jednodušším řešením, které nebude zavádět moderní způsoby, jakými by mohly být například knihovny Angluar nebo React. Vzniklá webová stránka tak bude pro ukázkové účely vystavěna za využití klasického HTML obohaceného pouze o šablonový engine Thymeleaf, kaskádové styly přenesené z frameworku Bootstrap a pro skriptovací potřeby poslouží JavaScript rozšířený o knihovnu jQuery.

V takovém obecnějším pohledu by se pak dalo říct, že aplikace tohoto typu bude rozdělena do tří hlavních částí, také zvaných vrstev. Mezi ty patří vrstva datová, vrstva aplikační a vrstva prezenční. Datová vrstva je úplným počátkem

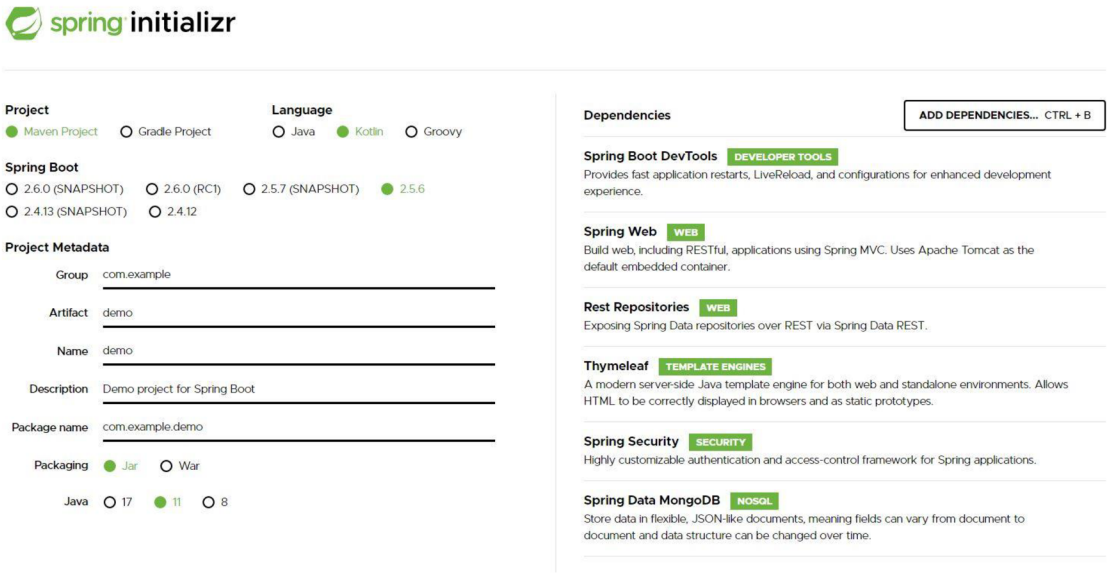
aplikace a ve své podstatě se dělí na dvě části. V první řadě se jedná hlavně o problém dat, tedy databázi samotnou a přístup k ní. Tato část aplikace tak bude obsahovat metody pro CRUD operace a další, obsažené v repositářích. V rámci té druhé části jsou do této vrstvy zahrnuty i samotné modely jednotlivých objektů, se kterými tato persistentní vrstva pracuje. Nad touto vrstvou stojí vrstva aplikační, známá též jako servisní či pod anglickým výrazem známá jako „business logic“. Zde jsou prováděny veškeré operace s daty, které předává datová vrstva a tedy implementuje funkce poskytované uživatelům ve vrstvě poslední. Touto vrstvou je prezenční nebo také webová vrstva, která obsahuje obsluhu všeho, co je nabízeno uživatelům skrze uživatelské rozhraní a design aplikace a dále obsahuje samozřejmě i prezentaci výsledků uživatelských akcí. Jak již bylo zmíněno, z pravidla zde neprobíhá žádná práce s daty, neboť se o tuto funkci stará prostřední z trojice částí aplikace.

5.2 Tvorba aplikace

Po velké části teorie je důležité také přiblížit praktické využití všech probraných témat a proto bude v této sekci nabídnut pohled do psaní kódu ukázkové webové aplikace. Od počátečního založení projektu s využitím rozhraní Spring Initializr (případně rozhraním poskytnutým vývojovým prostředím IntelliJ IDEA) se kapitola postupně přesune k prvním částem kódu, který bude postupem času vypracován až do konečné aplikace s názornými ukázkami z vývojového prostředí.

5.2.1 Založení projektu

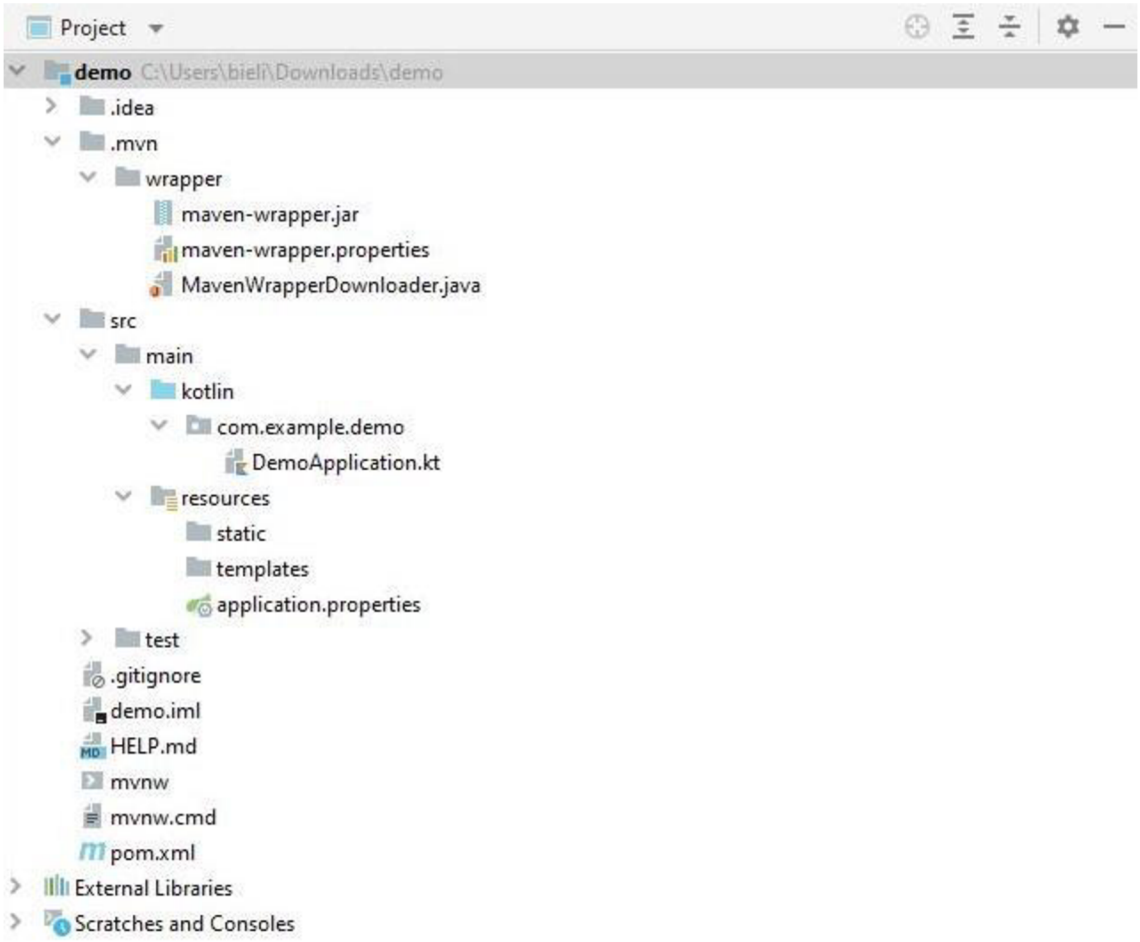
K založení projektu samotného lze zvolit možnost přímo v softwaru IntelliJ IDEA, nicméně pro názornou ukázku je zde z důvodu přehlednosti použito zmíněného webu Spring Initializr, neboť nastavení nerozděluje do více formulářů.



Obrázek 2: Nastavení projektu v rozhraní Spring Initializr

Zdroj: https://start.spring.io/

Takto vytvořený projekt po stažení a otevření ve vhodném IDE obsahuje základní strukturu s potřebnými soubory pro projekt a také pro správcovský nástroj Maven. Zdrojová složka je dále rozdělena do hlavní programové větve, testovací větve a nachází se zde i další soubory, mezi kterými se nachází například i velmi důležitý soubor pom.xml, který obsahuje všechny rozšiřující moduly projektu a s jeho rozšířením lze přidávat k celé práci další potřebná rozšíření. V hlavní větvi lze pak nalézt dvě složky a ta první bude místem, ve kterém budou programové soubory a to buď soubory Javy, Kotlinu nebo i obou najednou. Po založení projektu se zde nachází pouze hlavní třída obsahující spouštěcí metodu. Druhá ze složek hlavní větve slouží k uložení ostatních typů souborů jakými jsou webové stránky ve formátu HTML či jiném, různé šablony a statické materiály, které obsahují soubory s kaskádovými styly pro webové stránky a stejně tak jejich skripty (typicky JavaScript), ale také obrázky, ikony a další.



Obrázek 3: Základní struktura generovaného projektu

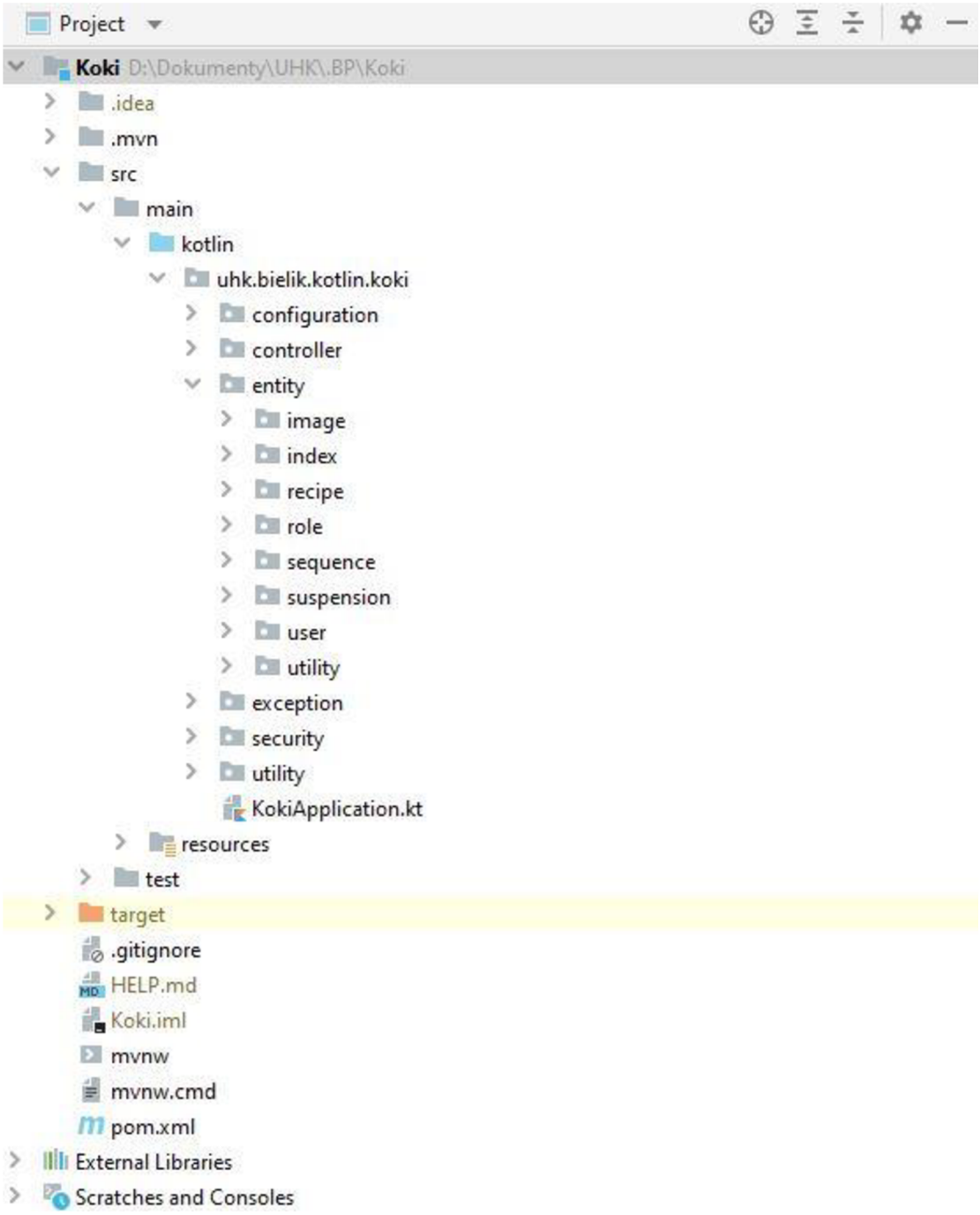
Zdroj: Autor

U takto založeného projektu je již možné začít tvořit aplikaci samotnou a to v první řadě rozdělením do vhodných balíčků. Balíčky, do kterých budou soubory Kotlinu rozděleny se nachází ve stejné složce, jako připravený spouštěcí soubor. Pro způsob rozdělení tříd do balíčků lze zvolit jeden z více způsobů. První je rozdělení dle vrstev aplikace, ve kterém by balíčky tvořily jednotlivé části aplikace a třídy jsou tak seskupeny podle jejich funkce. V balíčcích by tak spolu byly k nalezení například všechny třídy ovládající koncové body aplikace, všechny třídy obstarávající obsluhu databáze, třídy programové logiky a další. Dalším způsobem je namísto rozdělování aplikace podle vrstev seskupení tříd aplikace na základě prvků. U tohoto postupu jsou vytvářeny balíčky podle entit aplikace, což by v případě této ukázkové aplikace mohl být například uživatel nebo recept. V takovém balíčku pak budou třídy důležité pro obsluhu této entity, tedy její model, spojení s databázovou tabulkou,

programová logika a jiné. Tento způsob je i doporučený oficiální dokumentací. [39, s. 26-27]

5.2.1.1 Struktura ukázkové aplikace

Nyní se již popis tvorby odpoutá od zobecněných popisů a přenese se na konkrétní aplikaci. Celková struktura projektu vychází až z dokončené aplikace, nicméně v zájmu návaznosti na předchozí podkapitolu bude znázorněna již nyní. Strukturu výsledné aplikace znázorní následující obrázek.



Obrázek 4: Struktura ukázkové aplikace

Zdroj: Autor

V základní struktuře se backendová část aplikace dělí do více balíčků. Jak již bylo zmíněno toto rozdělení je možné provést různými způsoby. V úvahu přichází i první probraná možnost, taková struktura se však může začít brzy jevit značně nepřehlednou. Druhou možností rozdělení balíčků podle entit, v tomto případě v základu uživatelé a recepty, vzniká poměrně přehledná struktura. Každý takový

balíček pak bude obsahovat třídy tuto entitu obstarávající. Dále budou oba systémy dle potřeby obsahovat balíčky pro další třídy aplikace jako například konfigurační třídy aj. Ze zmíněného důvodu byl pro tuto aplikaci zvolen druhý způsob rozdělení, ovšem s vyjmutím kontrolérů do samostatného balíčku.

5.2.2 Vrstvy aplikace

Po struktuře projektu bude dále popsána datová třída představující entitu aplikace a všechny třídy, které jsou potřeba k její obsluze. Ostatní entitní balíčky vytvořené stejným principem nebudou podrobněji rozebrány. V této sekci se tak budou nacházet bližší informace k perzistentní, datové, servisní a prezenční vrstvě aplikace.

5.2.2.1 Datové třídy

Pro vytvoření persistentní vrstvy aplikace slouží definice datových tříd. Jedná se v podstatě o model záznamu v databázi a pro jeho vytvoření tak stačí pouze definice požadovaných vlastností. Pro ukázku bude v této práci použita třída uživatele. Ta bude obsahovat identifikátor, přezdívku, emailovou adresu, šifrované heslo, jméno, příjmení, ale také další informace, mezi které patří například seznamy přidaných receptů a sledovaných uživatelů, popisek a další.

```kotlin
@Document(collection = "users")
data class User(

    @MongoId
    var id: Long = 0,

    @Indexed(unique = true)
    var username: String = "",

    @Indexed(unique = true)
    var email: String = "",

    var password: String = "",

    var name: String = "",

    var surname: String = "",

    var description: String = "",

    var recipes: ArrayList<Long> = arrayListOf(),

    var following: ArrayList<Long> = arrayListOf(),

    var imageIdRef: Long = (1 + Random().nextInt(11)).toLong(),

    @DBRef
    var roles: Set<Role> = HashSet(),

    var suspended: Boolean = false,

    var suspensionId: Long = 0,

    var totalSuspensions: Int = 0
)
```

Ukázka kódu 1: Definice datové třídy

Zdroj: Autor

V první řadě je dobré si všimnou, jak je tato třída v Kotlinu definována. Označení „data“ dává Kotlinu najevo, jaký je účel třídy. Té jsou automaticky doplněny metody pro přístup a nastavování vlastností, ale také defaultní implementace metod equals, toString a hashCode. Stejně tak Springu udává účel třídy anotace @Document, díky které je v kontextu aplikace jasné, že třída modeluje záznam databáze, konkrétně pro databázi MongoDB. Použitý parametr pak pouze udává jméno, pod kterým bude kolekce těchto záznamů v databázi uložena. Defaultně by toto jméno bylo „User“ stejně tak jako název třídy.

Vlastnosti třídy – jejich definice probíhá přímo v parametrech třídy a blok kódu je zde tak zcela vynechán. [1, s. 90] Protože se jedná právě o Kotlin, musí být

každé vlastnosti buď explicitně nastaveno, že může být prázdná, nebo je nutné přiřadit defaultní hodnotu.

Zvolení defaultních hodnot namísto explicitního označení je vhodnější jak již z principů Kotlinu, tak z praktického hlediska, neboť vytvoření instance třídy je potřebné i pro relevantní formulář se třídou pracující (např. registrační formulář), čímž je Thymeleafu umožněno vyplnit potřebná pole a celou upravenou třídu předat kontroléru koncových bodů. Alternativním řešením by pak mohla být také pomocná třída, která obsahuje jen vlastnosti formuláře, nicméně v takovém případě je zase nutné hodnoty přeřazovat z jedné třídy do druhé.

V tomto případě je zvolena varianta defaultních hodnot, které jsou ovšem v podstatě prázdné (např. prázdný textový řetězec). Ty základní budou vyplněny při registraci uživatele a se zbytkem je pracováno dle potřeby užíváním aplikace.

Z ukázky kódu 1 lze vidět, že je každému uživateli při vytvoření přidělen náhodně defaultní obrázek. Obrázky jsou v databázi uloženy stejně tak jako všechny ostatní entity a mají svoji datovou třídu, která obsahuje vlastnost s hodnotou Binary, pro kterou MongoDB disponuje speciálním serializačním formátem. [46]

Konstruktor – ten je v Kotlinu rovněž vytvářen automaticky. Díky defaultním hodnotám je docíleno primárního konstruktoru bez parametrů [1, s. 80], což je také jedním ze záměrů Kotlinu. Zároveň je díky tomu možné vytvářet instance třídy s jakoukoliv kombinací vstupních parametrů – v aplikaci existuje celá množina konstruktorů.

Referenční vlastnosti – odkazují na jiné dokumenty v databázi, ale nejedná se o skutečné omezující propojení, které je známé z SQL databází jako cizí klíč. Není tak nutné řešit konflikty např. při smazání záznamů.

MongoDB má pro tento účel vlastní anotaci @DBRef, ta je ovšem spíše jakýmsi pomocným objektem a k jeho užívání je třeba dodatečných dotazů na databázi. [43] To způsobuje, zvláště pak u velkých aplikací, značnou neefektivitu práce s databází. Typickým řešením, které není nijak limitujícím a nezatěžuje navíc databázi, je prosté vkládání unikátního identifikátoru potřebného záznamu přímo do třídy, jako u již zmíněného obrázku, seznamu receptů a dalších, jak lze také vyčíst z ukázky kódu.

Validace – ta je u vlastností nastavena pomocí anotací. Kromě anotací naznačujících, že by pole mělo obsahovat unikátní hodnotu (přezdívka a email), zde přibývají i další anotace. S jejich pomocí se dá zajistit formát textových řetězců včetně znaků, délek aj. Pro vlastnosti které jsou reprezentovány polem je možné nastavit pravidla jak pro celé pole, tak i pro jednotlivé hodnoty v něm uložené. Mezi anotace validace patří například:

@Indexed – V MongoDB bude tato vlastnost indexována společně s id záznamu. To je potřeba pokud aplikace vyžaduje kontrolu unikátní hodnoty pole.

@TextIndexed – Přidá pole do textového indexu kolekce. Ten je využíván pro textové vyhledávání v této kolekci. Parametrem lze poli nastavit váhu, která je následně využita při řazení výsledků vyhledávání.

@NotNull – Vlastnosti musí být přiřazena hodnota. Ta může být prázdná, například prázdný textový řetězec.

@NotEmpty – Musí být přiřazena hodnota, která zároveň není prázdná.

@NotBlank – Musí být přiřazena hodnota, která po odstranění přebytečných mezer není prázdná.

@Size – Určuje délku řetězce.

@Min a **@Max** – Ekvivalent předchozí anotace pro číselné hodnoty, nastavují minimální a maximální hodnoty.

@Pattern – Pomocí regulárních výrazů může určovat vzor vkládané hodnoty a povolené znaky.

@Email – Stereotypní @Pattern pro emailové adresy.

Všem anotacím lze také nastavit chybová hláška, která je navracena při porušení konvence. Některé z těchto anotací zmiňuje i Walls. [2, s. 46-48] V Kotlinu je ovšem pro vlastnost generováno několik metod. Anotace je proto nutné zapsat trochu jiným formátem, zároveň však nelze použít vnořené anotace pro objekty uvnitř polí. U indexovaných omezení toto nutné není, protože jejich vyhodnocení probíhá až v databázi. Funkční využití anotací lze vidět na rozšíření původní třídy uživatele v následující ukázce kódu.

```
@Indexed(unique = true)
@TextIndexed(weight = 4F)
@field: NotNull
@field: Size(min = 3, max = 30)
@field: Pattern(regexp = "\\w+")
var username: String = "",

@Indexed(unique = true)
@TextIndexed(weight = 2F)
@field: NotBlank
@field: Size(max = 50)
@field: Email
var email: String = "",

@field: NotBlank
var password: String = "",

@TextIndexed(weight = 3F)
@field: NotBlank
@field: Size(max = 50)
@field: Pattern(regexp = "^[A-z]+[ \\-\\w]*$")
var name: String = "",

@TextIndexed(weight = 3F)
@field: NotNull
@field: Size(max = 50)
@field: Pattern(regexp = "^[A-z]*[ \\-\\w]*$")
var surname: String = "",

@TextIndexed(weight = 1F)
@field: NotNull
@field: Size(max = 250)
@field: Pattern(regexp = "[\\w\\s.,+;&\\-]*")
var description: String = "",
```

Ukázka kódu 2: Definice validačních pravidel anotacemi

Zdroj: Autor

Zde je rovněž možné vidět neaplikování určitého vzoru pro heslo přímo v datové třídě. Validace je totiž provedena dvakrát, při příchodu uživatelského dotazu do kontroléru a při ukládání záznamu do databáze. V druhém případě je heslo již přeformováno hashovacím algoritmem a došlo by tak k chybě.

5.2.2.2 Repositáře

Nad vrstvou perzistence stojí datová vrstva tvořená rozhraními, která přistupují k databázi. Každé rozhraní rozšiřuje jedno z těch defaultních, v případě této aplikace se jedná o MongoRepository. Zápisy „extends“ a „implements“ známé z Javy pro rozšíření a implementaci třídy jsou v Kotlinu nahrazeny dvojtečkou. Rozhraní přijímají objekt se kterým pracují a datový typ identifikátoru, zde tedy

třídu User a ID datového typu Long. Aby tato komponenta byla považována za repositář a byla přidána do kontextu aplikace je použita anotace @Repository.

Rozhraní disponuje základními dotazy pro databázi a přidat další je možné pouhým založením funkce s dodržením jmenné konvence metody, jak již bylo dříve popsáno. V případě UserRepository této aplikace je to pak například metoda „findAllBySuspended(suspended: Boolean)“. Z tohoto názvu Spring pozná, že jsou hledány všechny záznamy v tabulce uživatelů, u kterých je vlastnost „suspended“ naplněna hledanou hodnotou.

```
@Repository
interface UserRepository : MongoRepository<User, Long> {

    fun findByUsername(username: String): Collection<User>

    fun findAllBySuspended(suspended: Boolean): List<User>
}
```

Ukázka kódu 3: Repositář uživatelů

Zdroj: Autor

Výsledky lze vracet různými způsoby a to samotnou třídu, případně měnné, či neměnné kolekce. K vidění je navrácení jednotlivého výsledku v kolekci. Důvodem je způsob zpracování v servisní vrstvě. V té lze pak jednoduše zjistit, zda byl výsledek nalezen, nebo je naopak prázdný a umožní jednoduše tuto situaci ošetřit.

5.2.2.3 Servisní třídy

Data z datové vrstvy, ale také z kontrolérů v prezenční vrstvě, jsou zpracovávána servisními třídami servisní vrstvy. Využita zde bude nejen anotace @Service, ale také @Autowired, s jejíž pomocí je možné vložit existující instanci repositáře a jiných potřebných tříd do parametru a tím pádem i primárního konstruktoru třídy.

```
@Service
class UserService(
    @Autowired private val userRepository: UserRepository,
    @Autowired private val sequenceService: SequenceService,
    @Autowired private val roleService: RoleService,
    @Autowired private val imageService: ImageService,
    @Autowired private val mongoTemplate: MongoTemplate
)
```

Ukázka kódu 4: Definice servisní třídy uživatelů

Zdroj: Autor

Tato servisní třída pak disponuje mnoha funkcemi. V první řadě se zde nacházejí funkce využívající repositář pro získání záznamů z databáze. Zde lze vidět, proč je i jednotlivý uživatel ve vlastním databázovém dotazu navracen v kolekci pro jednoduché ošetření výjimkou. Zápis funkce je dále v Kotlinu od pohledu jiný – návratová hodnota se nachází až za funkcí a je oddělena dvojtečkou. Jednoduché funkce lze také zapisovat do jednořádkového formátu tak jako u následujících.

```
fun findUserForAuthentication(username: String): User =
    userRepository.findByUsername(username).firstOrNull { it.username
== username }
        ?: throw BadCredentialsException("Incorrect credentials")

fun findUserByUsername(username: String): User =
    userRepository.findByUsername(username).firstOrNull { it.username
== username }
        ?: throw UsernameNotFoundException("Sorry, it appears the user
$username does not exist")

fun findUserById(id: Long): User = userRepository.findByIdOrNull(id)
    ?: throw NoSuchElementException("Sorry, it appears the user of id
$id does not exist")

fun findAllSuspendedUsers(): List<User> =
userRepository.findAllBySuspended(true).sortedByDescending {
it.username }

fun findAllUsers(): List<User> {
    val users = userRepository.findAll(Sort.by(Sort.Direction.DESC,
"_id"))
    users.removeIf { it.suspended }
    return users
}

fun findUsersByTextSearch(search: String): List<User> {
    val criteria =
TextCriteria.forDefaultLanguage().matchingPhrase(search)
    val query = TextQuery.queryText(criteria).sortByScore()
    return mongoTemplate.find(query, User::class.java)
}
```

Ukázka kódu 5: Metody servisní třídy pracující s repositářem

Zdroj: Autor

V poslední metodě je k vidění i vlastní dotaz pro vyhledávání textem za využití MongoTemplate. Pouze pro ukázku, na dalším obrázku je možné vidět také dvě řešení pro dosažení stejného výsledku databázového dotazu použitím programového řazení a použití komplexnějšího automaticky odvozeného dotazu ze jména metody.

```
fun findAllSuspendedUsers(): List<User> =
userRepository.findAllBySuspended(true).sortedByDescending {
it.username }

fun findAllSuspendedUsersInferred(): List<User> =
userRepository.findAllBySuspendedOrderByUsernameDesc(true)
```

Ukázka kódu 6: Automaticky odvozený databázový dotaz

Zdroj: Autor

Mezi ostatní funkce patří vytváření, úpravy a mazání uživatelů a všechny ostatní metody pracující se zbylými vlastnostmi uživatele, mezi které patří i seznamy receptů a sledovaných uživatelů. Ve třídách se může nalézat také inicializační blok, který proběhne při vytvoření třídy při spouštění aplikace. [1, s. 79] V servisní třídě je u této aplikace využit pro naplnění databáze základními ukázkovými záznamy.

```
init {
    if (userRepository.findAll().isEmpty()) {
        …
    }
}
```

Ukázka kódu 7: Inicializační blok (minimalizovaný)

Zdroj: Autor

V celku zde tak probíhají veškeré operace, které vyžaduje prezenční vrstva aplikace ve svých kontrolérech.

5.2.2.4 Kontroléry

Nejvýše postavená vrstva aplikace, která má za úkol především obsluhu koncových bodů aplikace, ale může dále obsluhovat i kontrolu chybových hlášení. Neprobíhá zde žádná logika aplikace – všechny operace jsou vykonávány servisními třídami, kterým je jejich provedení delegováno kontrolérem. Jedná se o jedinou vrstvu se kterou uživatel interaguje skrze frontendovou část aplikace. Deklarace třídy probíhá stejným způsobem jako v minulém případě, je zde ovšem použita anotace @RestController.

Metody pro obsluhu koncových bodů jsou dále doplněny o anotaci mapující přístupovou URI. Vypadat mohou následujícím způsobem.

```
@GetMapping("/users")
fun users(): ModelAndView {
    val users = userService.findAllUsers()
    return prepareUsersPage(users)
}

@PostMapping("/users")
fun userSearch(@RequestParam(required = false, name = "search")
search: String): ModelAndView {
    val users: List<User> = if (search.isNotEmpty()) {
        userService.findUsersByTextSearch(search)
    } else {
        userService.findAllUsers()
    }
    val mv = prepareUsersPage(users)
    if (users.isEmpty()) mv.addObject("searchEmpty", true)
    return mv
}

private fun prepareUsersPage(users: List<User>): ModelAndView {
    val pictures = imageService.mapProfilePictures(users)

    val mv = ModelAndView("users")
    mv.addObject("title", "Users || Koki")
    mv.addObject("users", users)
    mv.addObject("pictures", pictures)

    mv.addObject("isAdmin", principalUtility.isAdmin())
    mv.addObject("principalPfp", principalUtility.getProfilePicture())
    return mv
}
```

Ukázka kódu 8: Obsluha koncových bodů v kontroléru

Zdroj: Autor

První metoda je jen velmi jednoduchou obsluhou načtení stránky uživatelů, která zobrazí jejich kompletní seznam. Návratovou hodnotou je třída ModelAndView, přičemž View je požadovanou HTML stránkou a do modelu je možné přidávat atributy pro přenos všech potřebných dat, které následně zpracuje engine Thymeleaf.

Ve druhé metodě je možné vidět zpracování stejné stránky, ve které byl ovšem pomocí vyhledávače zadán výraz pro textové vyhledávání v kolekci uživatelů. Přenesení vyhledávání z formuláře je docíleno anotací @RequestParam, které je nastaveno jméno pole z tabulky a zároveň není vyžadováno jeho vyplnění. Díky tomu lze ošetřit případ vyhledání bez zadání jakéhokoliv textu. V takovém případě jsou opět navráceni všichni uživatelé.

V poslední řadě je v ukázce kódu zobrazeno i využití pomocné třídy PrincipalUtility. Ta není servisní třídou a nepracuje tak s repositářem a databází, pracuje pouze s informacemi o právě přihlášeném uživateli, které jsou dostupné ze třídy SecurityContextHolder. Protože se nejedná o servisní třídu, bylo by možné tyto operace vykonávat přímo v kontroléru, zase ovšem platí, že v kontroléru by neměla být vykonávána aplikační logika. Pomocná třída bude později rovněž přiblížena.

```
@PostMapping("/update/info")
fun infoUpdate(
    @RequestParam("file") file: MultipartFile,
    @Valid @ModelAttribute("userForm") userUpdate: UserUpdateForm,
    bindingResult: BindingResult
): ModelAndView {
    val mv = ModelAndView()

    if (!bindingResult.hasErrors()) {
        userService.updateUser(userUpdate, file)
        mv.viewName = "redirect:/settings"
    } else {
        mv.viewName = "settings"
        mv.addObject("passwordForm", PasswordUpdateForm())
        mv.addObject("isAdmin", principalUtility.isAdmin())
        mv.addObject("principalPfp",
principalUtility.getProfilePicture())
    }

    return mv
}

@GetMapping("/{username}/follow")
fun followUser(@PathVariable("username") username: String):
ModelAndView {
    val user = userService.findUserByUsername(username)
    if (!principalUtility.isFollowing(user.id)) {
        userService.addFollowing(user.id,
principalUtility.getUsername())
    }
    return ModelAndView("redirect:/$username")
}
```

Ukázka kódu 9: Obsluha dalších koncových bodů v kontroléru

Zdroj: Autor

Na ostatních obsluhujících metodách z předchozí ukázky lze pozorovat další možnosti Springu. Soubor je v té první sloužící pro úpravu uživatelských informací získáván z formuláře opět stejnou anotací zaměřující pole formuláře dle jeho jména. Dále je zde však anotace @ModelAttribute, která získává z formuláře celou třídu s již vyplněnými hodnotami a lze ji tak pouze odeslat dále bez nutnosti přiřazování jednotlivých hodnot. [2, s. 70-71]

Je důležité podotknout, že přeposílání instance databázového modelu pro účely formulářů je sice vhodným řešením pro takovýto typ ukázkové aplikace (a v některých částech je tak aplikováno), nicméně v reálné aplikaci je flexibilnější a také bezpečnější použití speciální účelové třídy, která pouze přenáší potřebné vlastnosti a neobsahuje všechny náležitosti celé entity. Takové třídy jsou známé pod anglickou zkratkou DTO (Data Transfer Object). Takovými třídami jsou například i UserUpdateForm a PasswordUpdateForm, které je možné vidět v předchozí ukázce kódu a slouží k přenosu aktualizovaných informací o uživateli.

Mimo jiné jsou i tyto třídy v parametrech metod označeny anotací @Valid, díky které Spring provede validaci zadaných informací podle toho, jak byla omezení nastavena v těchto třídách. [2, s. 48-49] BindingResult je třída, která musí následovat hned za třídou z formuláře. Do té jsou přidány všechny chyby, které při validaci proběhnou. Je-li třída prázdná, aktualizace profilu se provede standardním způsobem, obsahuje-li chyby může být uživatel například vrácen zpět. Zaznamenané chyby jsou pak automaticky odeslány pro zobrazení chybových hlášek na stránce. Alternativně je možné použít i třídu Errors. [2, s. 49]

V poslední řadě stojí metoda sloužící pro přidání uživatele do seznamu sledovaných uživatelů. Z této ukázky lze vidět využití anotace @PathVariable, ve které je proměnná z URI obklopena složenými závorkami. Obsah těchto závorek je přenesen do stejnojmenného textového řetězce pro další zpracování. [2, s. 143-144]

5.2.2.5 Výjimky

Přestože obsluhu výjimek lze řešit i na lokální bázi přímo v kontrolérech, ale i mnoha jinými způsoby, v této aplikaci je pro tyto účely využito anotace @ControllerAdvice. Ta z třídy učiní globální kontrolér pro definované výjimky, v tomto případě se jedná o výjimky, které mohou vzniknout třeba u vyhledání uživatele. Vypadat mohou následovně.

```
@ControllerAdvice
class CustomResponseEntityExceptionHandler(
    @Autowired private val principalUtility: PrincipalUtility
) : ResponseEntityExceptionHandler() {

    @ExceptionHandler(NoSuchElementException::class)
    fun handleNoSuchElementException(e: NoSuchElementException):
ModelAndView {
        return prepareModelAndView(e, HttpStatus.NOT_FOUND)
    }

    @ExceptionHandler(UsernameNotFoundException::class)
    fun handleUsernameNotFoundException(e: UsernameNotFoundException):
ModelAndView {
        return prepareModelAndView(e, HttpStatus.NOT_FOUND)
    }
```

Ukázka kódu 10: Obsluha výjimek s využitím @ControllerAdvice

Zdroj: Autor

V aplikaci ovšem mohou nastat i výjimky, které se nedostávají postupným vybubláním přes vrstvy aplikace do kontrolérů. Jednat se může například o nepovolený přístup ke stránce, pro kterou nemá uživatel autorizační práva. Tento případ je spravován modulem Spring Security, který umožní přidání vlastní obsluhy výjimky pomocí vhodné třídy. Zde je příklad jen velmi jednoduchého přesměrování na koncový bod, který vrátí stránku s chybovou hláškou. Takovým způsobem je možné zakrýt, o jakou výjimku se jedná. Chyba o odepření přístupu ke stránce, o které by uživatel neměl vědět se tak může tvářit stejně, jako kdyby stránka neexistovala.

```
class CustomAccessDeniedHandler : AccessDeniedHandler {
    override fun handle(
        request: HttpServletRequest,
        response: HttpServletResponse,
        exception: AccessDeniedException
    ) {
        response.sendRedirect("/error")
    }
}
```

Ukázka kódu 11: Obsluha výjimek ve Spring Security

Zdroj: Autor

5.2.2.6 Pomocné třídy

Všechny další třídy které vykonávají pomocné funkce a měly by být součástí aplikačního kontextu je možné označit anotací @Component. Ukázková aplikace

disponuje dvěma takovými třídami, tou první je již zmíněná třída pracující s informacemi o přihlášeném uživateli, ta druhá pak slouží k práci s tokeny JWT.

```
@Component
class PrincipalUtility(
    @Autowired private val userService: UserService,
    @Autowired private val imageService: ImageService
) {

    fun getId(): Long =
userService.findUserByUsername(SecurityContextHolder.getContext().auth
entication.name).id

    fun getUsername(): String =
SecurityContextHolder.getContext().authentication.name

    fun isAdmin(): Boolean =
SecurityContextHolder.getContext().authentication.authorities.any { a
-> a == SimpleGrantedAuthority("ADMIN") }

    fun isLogged(): Boolean =
!SecurityContextHolder.getContext().authentication.authorities.any { a
-> a == SimpleGrantedAuthority("ROLE_ANONYMOUS") }

    fun getProfilePicture(): String {
        if (isLogged()) {
            val user =
userService.findUserByUsername(SecurityContextHolder.getContext().auth
entication.name)
            return imageService.getImageString(user.imageIdRef)
        }
        return ""
    }

    fun isFollowing(userId: Long): Boolean {
        if (isLogged()) {
            val user =
userService.findUserByUsername(SecurityContextHolder.getContext().auth
entication.name)
            return user.following.any { id -> id == userId }
        }
        return false
    }
}
```

Ukázka kódu 12: Pomocná třída PrincipalUtility

Zdroj: Autor

PrincipalUtility – nahrazuje nutnost pracovat se třídou SecurityContextHolder v ostatních částech aplikace. Díky té se dá zjistit ID uživatele, jeho přezdívka, zda má určitou roli (v tomto případě zda se jedná o admina aplikace) a také zda je nějaký uživatel vůbec přihlášen, nebo se jedná o anonymního návštěvníka. [2, s. 112-113] Dalšími rozšiřujícími funkcemi je zjištění, zda uživatel

sleduje jiného uživatele, jehož stránku navštěvuje a také získání profilového obrázku, který je vkládán do navigační lišty.

```
@Component
class JwtUtility {
    companion object Constants {
        private const val TOKEN_LIFETIME = 1000 * 60 * 60 * 24 * 14
        private const val COOKIE_LIFETIME = 60 * 60 * 24 * 14
        private const val TOKEN_SECRET = "thisshouldbesecret"
    }

    fun createNewJwtCookie(userDetails: UserDetails): Cookie {
        val jwt = createNewJwt(userDetails)
        val cookie = Cookie(HttpHeaders.AUTHORIZATION, jwt)
        cookie.maxAge = COOKIE_LIFETIME
        cookie.path = "/"
        cookie.isHttpOnly = true
        cookie.secure = true
        return cookie
    }

    fun createNewJwt(userDetails: UserDetails): String {
        val username = userDetails.username
        val roles = arrayListOf<String>()
        userDetails.authorities.stream().forEach { authority ->
            roles.add(authority.toString())
        }
        val claims = Jwts.claims()
        claims["roles"] = roles
        val expirationDate = Date(System.currentTimeMillis() +
TOKEN_LIFETIME)

        return
Jwts.builder().setClaims(claims).setSubject(username).setExpiration(ex
pirationDate)
            .signWith(SignatureAlgorithm.HS256,
TOKEN_SECRET).compact()
    }

    fun getTokenClaims(token: String): Claims {
        return
Jwts.parser().setSigningKey(TOKEN_SECRET).parseClaimsJws(token).body
    }
}
```

Ukázka kódu 13: Pomocná třída JwtUtility

Zdroj: Autor

JwtUtility – možná zajímavější třídou může být právě ta pracující s tokeny. Ta v první řadě disponuje funkcí pro vytvoření nového tokenu, ten vzniká po přihlášení do aplikace. Dále je zde i funkce která dokáže z tokenu extrahovat uložené hodnoty známé jako „claims“ a zároveň je při průběhu této funkce automaticky ověřena platnost tokenu. Tokeny jsou vhodné pro využití v REST aplikacích, neboť

tokeny jsou uloženy u uživatele a jejich ověření provádí aplikace. Nevzniká tak požadavek na porovnání se záznamem databáze, jako je tomu u uživatelských relací. V případě této aplikace se zde nachází i vytvoření cookie prohlížeče, která token uchovává.

Je důležité zdůraznit, že pro skutečnou aplikaci není tento způsob užití tokenů doporučen. Pro autorizační tokeny vznikají v reálných aplikacích často složité struktury. Jedním způsobem může být například vytvoření dvou tokenů. První token je autorizační, ten má životnost v řádech minut a je uložen u uživatele, jehož všechny požadavky jsou tímto tokenem doplněny. Vyprší-li životnost tokenu, je využito druhého obnovovacího tokenu, který může mít mnohem delší životnost, v některých případech i měsíce a roky. Po ověření obnovovacího tokenu aplikací je uživateli zaslán nový autorizační token. Obnovovací token by měl být ideálně uložen ve správně nastavené cookie.

Tokeny také obsahují určitý řetězec, díky kterému mohou být podepsány. Pouze tento podpis je tedy šifrovaný, ostatní informace jako jméno uživatele je z tokenu možné získat i „překladačem“. Tento řetězec je v ukázkové aplikaci pouhou konstantní hodnotou, nicméně v reálné aplikaci by se také mělo jednat o unikátní hodnotu uloženou ve speciálním úložišti v zakódované podobě.

Z důvodů limitací je tak v ukázkové aplikaci vytvořena jednodušší struktura, která by sice v reálné aplikaci nezajistila nejlepší možnou ochranu aplikace, ale dovoluje využít funkce Spring Security. V tomto případě je dobré alespoň nastavit pro cookie hodnoty „httpOnly“ a „secure“. Tím je docíleno zasílání pouze zabezpečeným HTTP protokolem a nepřístupnost cookie skrze JavaScript.

V poslední řadě je v této třídě také companion objekt, který ukládá konstantní hodnoty.

5.2.3 Konfigurace

Nastavení potřebných částí aplikace může probíhat s využitím XML souborů, nebo konfiguračních tříd. V projektu je zvolena druhá varianta a celkem obsahuje 4 konfigurační třídy.

5.2.3.1 Nastavení aplikace

Jakákoliv třída může být označena anotací @Configuration za účelem deklarace nových tříd označených pomocí @Bean. V této práci jsou nastaveny čtyři dodatečné třídy, především pak třída pro kódování hesel a validátor.

```
@Configuration
class ApplicationConfiguration {

    @Bean
    fun bCryptPasswordEncoder(): BCryptPasswordEncoder =
BCryptPasswordEncoder()

    @Bean
    fun validatingMongoEventListener(): ValidatingMongoEventListener =
        ValidatingMongoEventListener(localValidatorFactoryBean())

    @Bean
    fun localValidatorFactoryBean(): LocalValidatorFactoryBean =
LocalValidatorFactoryBean()

    @Bean
    fun loginAndRegisterPageFilter(): LoginAndRegisterPageFilter =
LoginAndRegisterPageFilter()
}
```

Ukázka kódu 14: Deklarace dodatečných tříd @Bean

Zdroj: Autor

5.2.3.2 Nastavení databáze

Rozšířením abstraktní třídy AbstractMongoClientConfiguration je možné nastavit MongoDB použitou v tomto projektu, samozřejmě za doprovodu anotace @Configuration. Na následují ukázce kódu lze vidět nastavení připojení k MongoDB Atlas [47], kde lze zdarma vytvořit základní databázi. K připojení je poskytnut speciální řetězec. Kromě připojení je potřeba také zvolit název databáze, která má být zaměřena, protože na MongoDB Atlas lze vytvořit i více databází. Posledním zde použitým nastavením je zapnutí indexování pro účely textového vyhledávání a kontrolu unikátních hodnot, neboť v defaultním nastavení je indexování vypnuto.

```
@Configuration
class MongoDBConfiguration : AbstractMongoClientConfiguration() {
    override fun getDatabaseName(): String {
        return "Kokidb"
    }

    override fun mongoClient(): MongoClient {
        val connectionString =
            ConnectionString(
"mongodb+srv://<NAME>:<PASSWORD>@kluster.efqun.mongodb.net/Kokidb?retr
yWrites=true&w=majority"
            )
        val settings = MongoClientSettings.builder()
            .applyConnectionString(connectionString)
            .build()
        return MongoClients.create(settings)
    }

    override fun autoIndexCreation(): Boolean {
        return true
    }
}
```

Ukázka kódu 15: Konfigurační třída databáze

Zdroj: Autor

5.2.3.3 Nastavení Spring MVC

Tohoto nastavení lze docílit rozšířením třídy WebMvcConfigurer s anotací @Configuration, která umožní konfigurací kontrolérů pomocí anotací namísto XML. Nastavit v ní lze například nějaké základní obsluhování koncových bodů aplikace, které pouze vykreslí požadovanou stránku. Dále je možné připojení statických zdrojů vyhledaných podle jejich cesty a nastavit jim jinou, kratší přístupovou cestu pro využití ve frontendové části bez nutnosti přístupu skrze adresu vyplývající z pozice výchozího souboru. Toho je využito v aplikaci. S anotací @EnableWebMvc lze také potlačit defaultní Spring Boot nastavení této třídy.

```
@Configuration
class WebMvcConfiguration : WebMvcConfigurer {
    override fun addResourceHandlers(registry:
ResourceHandlerRegistry) {
        registry
            .addResourceHandler("/webjars/**")
            .addResourceLocations("/webjars/")
        registry
            .addResourceHandler("/static/**")
            .addResourceLocations("/static/", "classpath:/static/")
        registry
            .addResourceHandler("/css/**")
            .addResourceLocations("/static/css/",
"classpath:/static/css/")
        registry
            .addResourceHandler("/favicon/**")
            .addResourceLocations("/static/favicon/",
"classpath:/static/favicon/")
        registry
            .addResourceHandler("/images/**")
            .addResourceLocations("/static/images/",
"classpath:/static/images/")
        registry
            .addResourceHandler("/js/**")
            .addResourceLocations("/static/js/",
"classpath:/static/js/")
    }
}
```

Ukázka kódu 16: Konfigurace Spring MVC

Zdroj: Autor

5.2.3.4 Nastavení zabezpečení

V poslední řadě disponuje aplikace nastavením webového zabezpečení. K tomu je určena třída WebSecurityConfigurerAdapter, jejíž rozšíření je opět doplněno dvojí anotací @Configuration a @EnableWebSecurity. V první metodě je aplikaci přiřazena služba starající se o získání přihlašovaného uživatele z databáze, kterou je třeba nejprve implementovat. Ve stejné metodě je přidána i třída pro kódování hesel, kterou aplikace využívá.

```
@Configuration
@EnableWebSecurity
class SecurityConfiguration(
    @Autowired private val userDetailsService:
CustomUserDetailsService
) : WebSecurityConfigurerAdapter() {

    @Autowired
    private val bCryptPasswordEncoder = BCryptPasswordEncoder()

    @Autowired
    private val loginAndRegisterPageFilter =
LoginAndRegisterPageFilter()

    override fun configure(auth: AuthenticationManagerBuilder) {
        auth
            .userDetailsService(userDetailsService)
            .passwordEncoder(bCryptPasswordEncoder)
    }
```

Ukázka kódu 17: Deklarace třídy zabezpečení

Zdroj: Autor

Další metoda přidává řadu nastavení třídě HttpSecurity. V první řadě je vypnuta ochrana proti útokům typu CSRF z důvodu použití autorizace pomocí tokenů u kterých není tato ochrana potřeba. Dalším nastavením je přístup ke koncovým bodům. Ty je třeba řadit od nejpovolanější role shora dolů, v tomto případě je nejvýše postavena role admin. Následuje nastavení přihlašování, u kterého jde mimo základní nastavení přidat také vlastní logiku při selhaném, nebo naopak provedeném přihlášení. Podobná nastavení existují i pro odhlášení uživatele, u kterého dochází i k vymazání souborů cookie z prohlížeče. V neposlední řadě jsou přidány i vlastní filtry, které budou přiblíženy v další části a je vypnuto tvoření relací, opět z důvodu používání tokenů. Přidána je i třída pro obsluhu zamítnutí přístupu.

```
override fun configure(http: HttpSecurity) {
    http
        .csrf().disable()
        .authorizeRequests()
        .antMatchers("/login", "/register", "/credits", "/error",
"/bad-credentials",
            "/recipes", "/{username}/recipe/{recipeId}").permitAll()
        .antMatchers(
            "/", "/index", "/home", "/feed", "/logout", "/users",
"/{username}", "/add-recipe",
            "/delete-recipe/{username}/recipe/{recipeId}/{imageId}",
            "/settings", "/update/**", "/{username}/follow",
"/{username}/unfollow"
        ).hasAuthority("USER")
        .antMatchers("/admin/**").hasAuthority("ADMIN")
        .anyRequest()
        .authenticated()
        .and()
        .formLogin()
        .loginPage("/login")
        .permitAll()
        .usernameParameter("username")
        .passwordParameter("password")
        .and()
        .logout()
        .permitAll()
        .logoutUrl("/logout")
        .deleteCookies("JSESSIONID")
        .deleteCookies("Authorization")
        .logoutSuccessUrl("/login")
        .and()
        .exceptionHandling()
        .accessDeniedHandler(CustomAccessDeniedHandler())
        .and()

.addFilter(CustomAuthenticationFilter(authenticationManager()))
        .addFilter(CustomAuthorizationFilter(authenticationManager()))
        .addFilterAfter(loginAndRegisterPageFilter,
CustomAuthenticationFilter::class.java)

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS)
}

override fun configure(web: WebSecurity) {
    web
        .ignoring()
        .antMatchers("/webjars/**")
        .antMatchers("/static/**")
        .antMatchers("/css/**")
        .antMatchers("/favicon/**")
        .antMatchers("/images/**")
        .antMatchers("/js/**")
        .antMatchers("/templates/**")
}
```

Ukázka kódu 18: Konfigurace zabezpečení

Zdroj: Autor

V poslední funkci jsou nastaveny cesty, které mají být zabezpečením ignorovány. Toto nastavení je důležité pro zdrojové složky, které by jinak byly nedostupné.

5.2.4 Spring Security

K předchozímu nastavení zabezpečení se váže také řada tříd, pro které má aplikace vlastní implementace. Mezi ty patří zmíněná služba UserDetailsService a následné filtry, které mají na starost autentizaci a autorizaci uživatele. Do této části aplikace patří i další třídy, jako například obsluha úspěšného, či neúspěšného přihlášení.

5.2.4.1 UserDetailsService

Tato služba má za úkol nalézt přihlašovaného uživatele v databázi a vytvořit novou instanci třídy User pro autentizaci. Je důležité si uvědomit, že v tomto případě se jedná o třídu User, která je součástí modulu Spring Security, ne o entitní model vytvořený pro tuto aplikaci a databázi. [2, s. 98-99] Do objektu pro autentizaci je předáno uživatelské jméno, šifrované heslo a role uživatele. V případě, že uživatel není v databázi nalezen je navrácena příslušná výjimka.

```
@Service
class CustomUserDetailsService(
    @Autowired private val userService: UserService
) : UserDetailsService {

    @Throws(BadCredentialsException::class)
    override fun loadUserByUsername(username: String): UserDetails {
        try {
            val user = userService.findUserForAuthentication(username)
            val roles = HashSet<GrantedAuthority>()
            user.roles.forEach { role: Role ->
                roles.add(SimpleGrantedAuthority(role.role))
            }
            val authorities = ArrayList(roles)

            return User(user.username, user.password, authorities)
        } catch (e: BadCredentialsException) {
            throw Exception(e)
        }
    }
}
```

Ukázka kódu 19: Implementace UserDetailsService

Zdroj: Autor

5.2.4.2 Autentizační filtr

V první řadě je dobré si u definice třídy povšimnout chyby s inspekcí tříd vložených pomocí anotace @Autowired, která může v některých případech nastat. Ta zamezí sestavení aplikace, nicméně je-li s jistotou možné říct, že tyto komponenty v kontextu aplikace existují, lze toto varování potlačit anotací @SupressWarning s náležitým parametrem specifikujícím potlačovanou chybovou hlášku.

```
class CustomAuthenticationFilter(
    @Autowired private val authManager: AuthenticationManager
) : UsernamePasswordAuthenticationFilter() {

    @Autowired
    @SuppressWarnings("SpringJavaAutowiringInspection")
    private val jwtUtility = JwtUtility()

    @Autowired
    @SuppressWarnings("SpringJavaAutowiringInspection")
    private val successHandler = CustomAuthenticationSuccessHandler()

    @Autowired
    @SuppressWarnings("SpringJavaAutowiringInspection")
    private val failureHandler = CustomAuthenticationFailureHandler()
```

Ukázka kódu 20: Definice autentizačního filtru

Zdroj: Autor

Co se týká metod samotných, jedním z dalších kroků k úspěšnému přihlášení uživatele je srovnání přihlašovacích údajů s údaji, které předchozí služba získala z databáze. Jelikož uživatelské jméno již bylo nalezeno v databázi a tedy odpovídá, je dále potřeba porovnání hesel. Heslo zadávané uživatelem při přihlášení má podobu prostého textu a je srovnáváno se zašifrovaným heslem z databáze. To ovšem není problém, přesně z tohoto důvodu byl aplikaci v konfigurační třídě nastaven hašovací algoritmus. Srovnání je provedeno rozhraním AuthenticationManager, respektive jeho defaultní implementací. [36, s. 65]

Druhou metodou je v tomto filtru řešena situace po úspěšném přihlášení. V takovém případě je uživateli vytvořen token JWT společně s cookie ve které je uložen a ta je následně přidána do servletu. Filtrování následně pokračuje dále.

```
@Throws(AuthenticationException::class)
override fun attemptAuthentication(request: HttpServletRequest,
response: HttpServletResponse): Authentication {
    try {
        return authManager.authenticate(
            UsernamePasswordAuthenticationToken(
                obtainUsername(request),
                obtainPassword(request),
                arrayListOf<GrantedAuthority>()
            )
        )
    } catch (e: AuthenticationException) {
        throw e
    }
}

override fun successfulAuthentication(
    request: HttpServletRequest,
    response: HttpServletResponse,
    chain: FilterChain,
    authResult: Authentication
) {
    if (authResult.principal != null) {
        val user = authResult.principal as
org.springframework.security.core.userdetails.User
        if (user.username != null && user.username.isNotEmpty()) {
            val cookie = jwtUtility.createNewJwtCookie(user)
            response.addCookie(cookie)
            successHandler.onAuthenticationSuccess(request, response,
authResult)
        }
    }
}

override fun unsuccessfulAuthentication(
    request: HttpServletRequest,
    response: HttpServletResponse,
    failed: AuthenticationException
) {
    failureHandler.onAuthenticationFailure(request, response, failed)
}
```

Ukázka kódu 21: Metody autentizačního filtru

Zdroj: Autor

5.2.4.3 Autorizační filtr

Následujícím filtrem je filtr autorizace, kterým požadavek na aplikaci projde nejen po úspěšném přihlášení, ale při každém požadavku. [36, s. 153] Kdykoliv tak uživatel například přejde na jinou stránku, aplikace ověří jeho práva tímto filtrem.

V případě, že se uživatel snaží zobrazit stránku přihlášení nebo registrace, je krok ověření rolí přeskočen, neboť žádnou nevyžaduje. Stejně tak v případě kdy

uživatel nedisponuje tokenem, je jeho požadavek přeskočen a dále přesměrován na přihlašovací stránku.

Má-li uživatel token, je zkontrolováno jméno uživatele a také životnost tokenu, obě tyto hodnoty jsou v tokenu uloženy. Následně je vytvořen token používaný třídou SecurityContextHolder přenášející jméno a práva uživatele. Vložit zde lze i heslo, nicméně je to pro funkčnost aplikace zbytečné a možná dokonce i nežádoucí.

```
class CustomAuthorizationFilter(
    @Autowired private val authManager: AuthenticationManager
) : BasicAuthenticationFilter(authManager) {

    @Autowired
    @SuppressWarnings("SpringJavaAutowiringInspection")
    private val jwtUtility: JwtUtility = JwtUtility()

    override fun doFilterInternal(request: HttpServletRequest,
response: HttpServletResponse, chain: FilterChain) {
        if (request.servletPath.equals("/login") ||
request.servletPath.equals("/register")) {
            chain.doFilter(request, response)
            return
        }

        val jwtCookie = WebUtils.getCookie(request,
HttpHeaders.AUTHORIZATION)

        if (jwtCookie == null) {
            chain.doFilter(request, response)
            return
        } else {
            try {
                val claims =
jwtUtility.getTokenClaims(jwtCookie.value)
                val username = claims.subject
                val roles = claims["roles"] as ArrayList<*>
                val authorities = arrayListOf<GrantedAuthority>()

                if (username != null && username.isNotEmpty()) {
                    for (role in roles) {
                        val authority = SimpleGrantedAuthority(role as
String)
                        authorities.add(authority)
                    }
                    val authToken =
UsernamePasswordAuthenticationToken(username, null, authorities)
                    SecurityContextHolder.getContext().authentication
= authToken
                } else {
                    SecurityContextHolder.getContext().authentication
= null
                }

                chain.doFilter(request, response)
            } catch (e: ExpiredJwtException) {
                throw Exception(e)
            }
        }
    }
}
```

Ukázka kódu 22: Implementace autorizačního filtru

Zdroj: Autor

5.2.4.4 Filtr přesměrování

Jedním z vlastních filtrů, které jsou běžně využívány, může být prostý přesměrovávající filtr. Zde se jedná o případ, ve kterém se již přihlášený uživatel snaží zobrazit stránku přihlášení, nebo registrace. To je z pochopitelných důvodů nevhodné, v této aplikaci se na těchto stránkách nenachází žádné rozšiřující funkce. Dostane-li aplikace požadavek na tyto stránky a autorizační filtr zaznamená přihlášeného uživatele, bude tento požadavek přesměrován zpět na domovskou stránku.

```
class LoginAndRegisterPageFilter : GenericFilterBean() {
    override fun doFilter(request: ServletRequest, response:
ServletResponse, chain: FilterChain) {
        val requestServlet = request as HttpServletRequest
        val responseServlet = response as HttpServletResponse

        if ((requestServlet.requestURI.equals("/login") ||
requestServlet.requestURI.equals("/register"))
            && WebUtils.getCookie(request, HttpHeaders.AUTHORIZATION)
!= null
        ) {
            val encodedRedirectURL =
response.encodeRedirectURL(requestServlet.contextPath.toString() +
"/")
            responseServlet.status =
HttpStatus.TEMPORARY_REDIRECT.value()
            responseServlet.setHeader("Location", encodedRedirectURL)
        }

        chain.doFilter(requestServlet, responseServlet)
    }
}
```

Ukázka kódu 23: Filtr přesměrování

Zdroj: Autor

5.2.4.5 Obsluha po přihlášení

Spring nabízí i vlastní logiku postupu po úspěšném, nebo naopak neúspěšném přihlášení. Zde je pouze jednoduchým způsobem nastavena cílová adresa koncového bodu podle toho, jaké oprávnění uživatel má. V případě běžného uživatele je přesměrování cíleno na domovskou stránku, u administrátorů je to pak pouze jimi přístupná stránka umožňující revizi pozastavených účtů. Přeskočení kroku přesměrování je možné z následujícího obrázku vidět na začátku metody. To proběhne například v případě výjimky, která zapříčiní nemožnost provádění změn na servletech.

```
@Component
class CustomAuthenticationSuccessHandler :
SavedRequestAwareAuthenticationSuccessHandler() {

    private val redirect = DefaultRedirectStrategy()

    @Throws(IOException::class, ServletException::class)
    override fun onAuthenticationSuccess(
        request: HttpServletRequest,
        response: HttpServletResponse,
        authentication: Authentication
    ) {
        if (response.isCommitted) {
            return
        }

        var url = "/"
        if (authentication.authorities.any { a -> a ==
SimpleGrantedAuthority("ADMIN") }) {
            url = "/admin/tools"
        }
        redirect.sendRedirect(request, response, url)
    }
}
```

Ukázka kódu 24: Obsluha úspěšného přihlášení

Zdroj: Autor

6 Shrnutí výsledků

Na začátek shrnutí by bylo vhodné po představení backendové části aplikace a tedy spolupráce Springu s Kotlinem nahlédnout také na frontendovou podobu výsledné aplikace. V práci lze na tuto část nahlédnout v příloze 1. Zároveň je celý projekt dostupný v internetovém repositáři zde: https://gitlab.com/bielima1/koki.

Dále je na řadě k probrání hlavní téma práce a to praktické využití obou technologií pro tvorbu webových aplikací.

Kotlin má jakožto jazyk snažící se o vytvoření „lepší Javy“ výhodu v tom, že není prvním takovým jazykem. Inovace zavedené jazyky Groovy a Scala, které existovaly již řadu let před prvním představením Kotlinu, tak budou pro řadu nových zájemců již známé. Ovšem v případě této práce jsou osobní zkušenosti s Kotlinem porovnávány pouze s obyčejnou Javou SE. A ani v takovém případě nevznikají při práci nováčka žádné zásadní problémy. Práce s Kotlinem je pro vývojáře Javy intuitivní a dokumentace dostupná i v podobě knihy [1] je velmi obsáhlá.

Záměr Kotlinu a jeho vývojářů je velmi praktický, z čehož lze těžit již od samotného začátku práce. Jak bylo možné vidět v předchozích částech, vytváření perzistentní vrstvy aplikace s využitím datových tříd bylo velmi efektivní a kompletně eliminovalo boilerplate kód běžný pro starší verze Javy, které jsou i z důvodu zpětné podpory existujícího softwaru nadále používány. Nové aktualizace pro tyto softwary by ovšem mohly být klidně vytvořeny v Kotlinu díky podpoře interoperability.

Způsob, jakým se Kotlin rozhodl vyřešit problém s nulovou referencí, nutí řešit základní věci lehce odlišným způsobem, než jak by mohlo být zvykem u Javy. Příkladem z této aplikace může být registrace nového uživatele, při které je formuláři pro vyplnění dat odeslána prázdná třída, na kterou mají být pole mapována. Místo vytvoření prázdné instance třídy uživatele je v Kotlinu nutné buď nastavit výchozí hodnoty přímo v modelu, nebo nějaké doplnit při vytváření instance namísto vyplnění těchto hodnot až při skutečném odeslání tohoto formuláře a nutnosti záznam uložit do databáze. Na první pohled by se takový způsob mohl zdát nepohodlným, nicméně toto řešení se naopak projevuje jako velmi

příjemné při další práci s objektem, neboť eliminuje přítomnost lidské chyby, která by jinak mohla nastav v mnoha místech aplikace. Kotlin se nenechá odbýt ani v případech, ve kterých je nějaká hodnota explicitně kontrolována a vždy žádá řádné ošetření. V celku je velmi jednoduché těmto změnám přivyknout a jejich absence je u Javy velmi citelná.

Změn má Kotlin mnoho a o žádné z nich se nedá říct, že by byla na škodu. Deklarace tříd a funkcí je zase o něco rychlejší, v druhém případě lze u těch jednodušších zvolit i zápis jednořádkový. Dále práci zlepší i vylepšená podoba Java Stream pro práci s kolekcemi, práce s lambda výrazy nebo rozšiřující funkce, zkrátka zřejmě každý u Kotlinu najde něco, co bude vedle Javy příjemnější.

Spring na rozdíl od Kotlinu nemusí na první pohled působit zdaleka tak intuitivním dojmem. Spring Boot je skutečně velkým ulehčením pro start nového projektu, protože obsahuje potřebné základní závislosti a především pak při přidávání dalších modulů automaticky připojí i jejich ostatní závislosti. Stále se však jedná o velice rozsáhlý framework, který může být pro začínající vývojáře imponující.

Běžné součásti Spring aplikací které jsou hojně využívány byly natolik zjednodušeny, že se mohou stávat až matoucími. Z dokumentace lze totiž jednoduše vyčíst například to, kde, proč a jakým způsobem použít danou anotaci, například @Component nebo jednu z anotací od této odvozených, ale do útrob systému na jehož pozadí všechny části aplikace zapadají do sebe již vidět nelze. Díky tomu aplikace sice vzniká rychle a efektivně, zároveň se však vývojář nachází v situaci kdy aplikace „prostě funguje“, aniž by musel tušit, proč tomu tak je. Podobná situace je pak ještě citelnější v případě řetězce filtrů zabezpečení. Zde je nutné implementovat vlastní třídu pro získání uživatele z databáze a autentizační a autorizační filtry, jak již bylo přiblíženo dříve. Takový postup se může zdát kompletní, jednalo by se však o velmi naivní implementaci zabezpečení. Celkový řetězec filtrů obsahuje kroky, které jsou sice pro vývojáře nedůležité a z toho důvodu i skryté, při řešení chyb v programu ovšem nezkušenému vývojáři práci opět spíše komplikují. Stejně tak i ošetření výjimek může svým rozdělením na rozdílné způsoby mást.

Celkově by se tak dalo říct, že je Spring zpočátku možná trochu náročným nástrojem pro začátečníka, především pak ve srovnání s Kotlinem, ovšem po

seznámení se s jeho funkcemi se stává nástrojem skutečně silným a vybaveným pro spoustu případů. Disponuje mnoha řešeními která jsou velice rychlá na implementaci a také často nabízejí několik možných způsobů, jak požadovaného výsledku docílit. Není proto překvapivé, že se mezi Java vývojáři těší tak velké a rozhodně i zasloužené popularitě, především pak díky projektu Spring Boot, který počáteční problémy redukuje na minimum.

7 Závěry a doporučení

Ze zkušeností získaných prací na ukázkovém projektu je také důležité vyhodnotit spolupráci jazyku Kotlin a frameworku Spring dohromady. Hlavní myšlenkou Kotlinu je totiž vytvoření „lepší Javy“ která je s tou skutečnou Javou plně interoperabilní a z její podstaty by tak Kotlin měl bezproblémově fungovat i ve spolupráci se Springem.

Pravdou je, že interoperabilita Kotlinu a Javy funguje opravdu velmi dobře. Spring byl navržen pro Javu a dlouho si udržoval podporu i pro velmi staré verze kvůli zpětné kompatibilitě a je možné, že by u těchto verzí teoreticky mohly vznikat určité problémy, nicméně nejnovější verze Spring 5.0 již počítá s novými verzemi Javy a především i se samotným Kotlinem, který je nyní koneckonců možné zvolit i jako výchozí jazyk při zakládání projektu.

V ukázkové aplikaci v žádném případě nevznikla situace, ve které by bylo použití Kotlinu výraznější překážkou. Až na jednu výjimku nebyl příčinou žádných vzniklých problémů, kterým by bylo možné se použitím Javy vyhnout. Touto výjimkou je validace uživatelských vstupů s využitím anotací, která v kombinace s Kotlinem nefunguje úplně bezchybně a zároveň není jednoduše možné vkládat validační pravidla pomocí anotací například pro datové typy v seznamech (stále je ovšem možné takovouto validaci řešit programově). Celkově je ovšem možné říct, že nové metody Kotlinu jako například jeho řešení prázdných hodnot působí jako velmi dobrá prevence proti lidské chybě a oproti Javě tak případným problémům spíše zabraňuje.

Na mysl by také mohla přijít myšlenka výkonu Springu postaveného na Javě a aplikace vybudované s pomocí Kotlinu. Jak již ovšem bylo probráno v kapitole o Kotlinu, rozdíl v rychlosti zpracování obou jazyků se liší tak minimálními hodnotami, že se tento rozdíl nedá považovat za důležitý a zřejmě by byl i v rozsáhlé reálné aplikaci velmi těžko pozorovatelný.

Z těchto zjištění lze tedy usoudit, že je Kotlin nejen skoro plnohodnotnou náhradou Javy pro Spring, ale je možné ho považovat i za tu lepší variantu. Z předložených výhod Kotlinu navíc nemusí nutně profitovat pouze nové Spring aplikace, ale i ty existující. U případu s vývojem mobilních aplikací byla zmíněna

postupná migrace z Javy 6 na Kotlin, u které byly nové části vytvářeny v Kotlinu, a dále vývojáři zmiňovali i postupné přepisování existujících tříd do Kotlinu. Stejný postoj lze tak zaujmout i v případě webových aplikací, které je možné s Kotlinem rozšiřovat a postupně staré třídy i nahrazovat. V poslední řadě je dobré zmínit, že Kotlin přinesl řadu změn a inovací, díky kterým Javu nejen předčil, ale možná také donutil k vlastnímu pokroku a přestože se stále jeví díky osobním preferencím tou lepší variantou, i Java se postupem času přibližuje jeho novým možnostem. Jediný drobný nedostatek Kotlinu je tak převažován pozitivy a spíše než na jazyku samotném tak záleží především na preferenci vývojáře. Doufat lze také v to, že i tento drobný problém vývojáři společnosti JetBrains časem vyřeší.

8 Seznam použité literatury

8.1 Tištěné zdroje

[1] JEMEROV, Dmitry., ISAKOVA, Svetlana. *Kotlin in Action*. Shelter Island: Manning Publications Co., 2017. 360 s. ISBN 9781617293290.

[2] WALLS, Craig. *Spring in Action, Fifth Edition*. Shelter Island: Manning Publications Co., 2018. 520 s. ISBN 9781617294945.

8.2 Internetové zdroje

[3] ORACLE CORPORATION. Oracle Java. *Oracle.* [online]. 2021 [cit. 2021-10-27]. Dostupné z: https://www.oracle.com/java.

[4] TIOBE SOFTWARE BV. TIOBE Index. *TIOBE – The Software Quality Company.* [online]. Říjen 2021 [cit. 2021-10-27]. Dostupné z: https://www.tiobe.com/tiobe-index.

[5] ORACLE CORPORATION. Oracle Java Technologies. *Oracle*. [online]. 2021 [cit. 2021-01-24]. Dostupné z: https://www.oracle.com/java/technologies.

[6] ORACLE CORPORATION. The Java Language Environment. *Oracle*. [online]. 2021 [cit. 2021-01-24]. Dostupné z: https://www.oracle.com/java/technologies/introduction-to-Java.html.

[7] JETBRAINS S.R.O. All Developer Tools and Products by JetBrains. *JetBrains.* [online]. 2021 [cit. 2021-01-24]. Dostupné z: https://www.jetbrains.com/products.html.

[8] JETBRAINS S.R.O. FAQ. *Kotlin.* [online]. 2021 [cit. 2021-01-24]. Dostupné z: https://www.kotlinlang.org/docs/reference/faq.html.

[9] JETBRAINS S.R.O. Learn Kotlin. *Kotlin.* [online]. 2021 [cit. 2021-01-25]. Dostupné z: https://www.kotlinlang.org/docs.

[10] JETBRAINS S.R.O. The State of Developer Ecosystem 2020. *Kotlin.* [online]. 2021 [cit. 2021-01-25]. Dostupné z: https://www.jetbrains.com/lp/devecosystem-2020.

[11] GOOGLE LLC. Kotlin on Android FAQ. *Android Developers.* [online]. 2021 [cit. 2021-01-25]. Dostupné z: https://developer.android.com/kotlin/faq.

[12] JETBRAINS S.R.O. Using Kotlin for Server-side Development. *Kotlin.* [online]. 2021 [cit. 2021-01-25]. Dostupné z: https://www.kotlinlang.org/docs/reference/server-overview.html.

[13] MAPLE, S., BINSTOCK, A. JVM Ecosystem report 2018 – About your Platform and Application. Snyk. [online]. 2018 [cit. 2021-01-28]. Dostupné z: https://snyk.io/blog/jvm-ecosystem-report-2018-platform-application.

[14] WMWARE, INC. Microservices. *Spring*. [online]. 2021 [cit. 2021-01-28]. Dostupné z: https://www.spring.io/microservices.

[15] WMWARE, INC. Event Driven. *Spring*. [online]. 2021 [cit. 2021-01-28]. Dostupné z: https://www.spring.io/event-driven.

[16] WMWARE, INC. Reactive. *Spring*. [online]. 2021 [cit. 2021-01-29]. Dostupné z: https://www.spring.io/reactive.

[17] WIGGINS, A. The Twelve Factor App. [online]. 2017 [cit. 2021-01-29]. Dostupné z: https://www.12factor.net.

[18] WMWARE, INC. Cloud. *Spring*. [online]. 2021 [cit. 2021-01-29]. Dostupné z: https://www.spring.io/cloud.

[19] WMWARE, INC. Serverless. *Spring*. [online]. 2021 [cit. 2021-01-29]. Dostupné z: https://www.spring.io/serverless.

[20] WMWARE, INC. Web Applications. *Spring*. [online]. 2021 [cit. 2021-01-29]. Dostupné z: https://www.spring.io/web-applications.

[21] WMWARE, INC. Batch. *Spring*. [online]. 2021 [cit. 2021-01-29]. Dostupné z: https://www.spring.io/batch.

[22] WMWARE, INC. Spring Framework Overview. *Docs.spring.* [online]. 2021-10-21 [cit. 2021-10-25]. Dostupné z: https://docs.spring.io/spring-framework/docs/5.3.12/reference/pdf/overview.pdf.

[23] WMWARE, INC. Core Technologies. *Docs.spring.* [online]. 2021-10-21 [cit. 2021-10-25]. Dostupné z: https://docs.spring.io/spring-framework/docs/5.3.12/reference/pdf/core.pdf.

[24] NGUYEN, N.T. What is a Spring Bean. *Baeldung*. [online]. 2019-12-7 [cit. 2021-06-06]. Dostupné z: https://www.baeldung.com/spring-bean.

[25] FEJÉR, A. Spring Core Annotations. Baeldung. [online]. 2019-02-19 [cit. 2021-06-06]. Dostupné z: https://www.baeldung.com/spring-core-annotations.

[26] BAELDUNG. The Spring @Controller and @RestController Annotations. Baeldung. [online]. 2021-18-03 [cit. 2021-06-07]. Dostupné z: https://www.baeldung.com/spring-controller-vs-restcontroller.

[27] WMWARE, INC. Data Access. *Docs.spring.* [online]. 2021-10-21 [cit. 2021-10-26]. Dostupné z: https://docs.spring.io/spring-framework/docs/5.3.12/reference/pdf/data-access.pdf.

[28] WMWARE, INC. Web on Servlet Stack. *Docs.spring.* [online]. 2021-10-21 [cit. 2021-10-26]. Dostupné z: https://docs.spring.io/spring-framework/docs/5.3.12/reference/pdf/web.pdf.

[29] BAELDUNG. Servlet Redirect vs Forward. Baeldung. [online]. 2020-23-12 [cit. 2021-06-12]. Dostupné z: https://www.baeldung.com/servlet-redirect-forward.

[30] BAELDUNG. Model, ModelMap and ModelAndView in Spring MVC. Baeldung. [online]. 2020-26-12 [cit. 2021-06-12]. Dostupné z: https://www.baeldung.com/spring-mvc-model-model-map-model-view.

[31] GIERKE, O., DARIMONT, T., STROBL, C., POLLACK, M., RISBERG, T., PALUCH, M. Spring Data Commons. *Docs.spring.* [online]. 2017-10-27 [cit. 2021-10-26]. Dostupné z: https://docs.spring.io/spring-data/data-commons/docs/2.0.1.RELEASE/reference/pdf/spring-data-commons-reference.pdf.

[32] JETBRAINS S.R.O. Data classes. *Kotlin.* [online]. 2021-02-11 [cit. 2021-06-13]. Dostupné z: https://kotlinlang.org/docs/data-classes.html.

[33] WMWARE, INC. Spring Data. [online]. 2021 [cit. 2021-06-14]. Dostupné z: https://spring.io/projects/spring-data.

[34] SOLID IT. DB-Engines Ranking – popularity ranking of database management systems. [online]. Říjen 2021 [cit. 2021-10-27]. Dostupné z: https://db-engines.com/en/ranking.

[35] POLLACK, M., RISBERG, T., GIERKE, O., LEAU, C., BRISBIN, J., DARIMONT, T., STROBL, C., PALUCH, M. Spring Data MongoDB. *Docs.spring.* [online]. 2017-

10-27 [cit. 2021-10-26]. Dostupné z: https://docs.spring.io/spring-data/data-mongo/docs/2.0.1.RELEASE/reference/pdf/spring-data-mongodb-reference.pdf.

[36] ALEX, B., TAYLOR, L., WINCH, R., HILLERT, G., GRANDJA, J., BRYANT, J., MELÉNDEZ, E., CUMMINGS, J., SYER, D., STEIN, E. Spring Security Reference. *Docs.spring.* [online]. 2021-10-18 [cit. 2021-10-26]. Dostupné z: https://docs.spring.io/spring-security/site/docs/current/reference/pdf/spring-security-reference.pdf.

[37] BAELDUNG. Granted Authority Versus Role in Spring Security. Baeldung. [online]. 2020-15-08 [cit. 2021-06-21]. Dostupné z: https://www.baeldung.com/spring-security-granted-authority-vs-role.

[38] THE THYMELEAF TEAM. Tutorial: Thymeleaf + Spring. [online]. 2018-10-29 [cit. 2021-06-21]. Dostupné z: https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html#integration-with-requestdatavalueprocessor.

[39] WEBB, P., SYER, D., LONG, J., NICOLL, S., WINCH, R., WILKINSON, A., OVERDIJK, M., DUPUIS, C., DELEUZE, S., SIMONS, M., PAVIĆ, V., BRYANT, J., BHAVE, M., MELÉNDEZ, E., FREDERICK, S. Spring Boot Reference Documentation. *Docs.spring.* [online]. 2021-10-21 [cit. 2021-10-26]. Dostupné z: https://docs.spring.io/spring-boot/docs/current/reference/pdf/spring-boot-reference.pdf.

[40] WMWARE, INC. Spring Initializr. [online] 2021 [cit. 2021-06-26]. Dostupné z: https://start.spring.io.

[41] WMWARE, INC. Getting Started | Validating Form input. *Spring.* [online]. 2021 [cit. 2021-06-28]. Dostupné z: https://spring.io/guides/gs/validating-form-input.

[42] ANIOŁA, J. Java vs. Kotlin – Part 1: Performance. *Java vs. Kotlin — Part 1: Performance | by Jakub Anioła | RSQ Technologies | Medium.com.* [online]. 2019-08-13 [cit. 2021-10-01]. Dostupné z: https://medium.com/rsq-technologies/comparative-evaluation-of-selected-constructs-in-java-and-kotlin-part-1-dynamic-metrics-2592820ce80.

[43] MONGODB, INC. Database References – MongoDB Manual. *Docs.mongodb*. [online]. 2021 [cit. 2021-10-12]. Dostupné z: https://docs.mongodb.com/manual/reference/database-references.

[44] GIERKE, O., DARIMONT, T., STROBL, C., POLLAK, M., RISBERG, T., PALLUCH, M., BRYANT, J. Spring Data Commons – Reference Documentation. *Docs.spring*. [online] 2021-05-14 [cit. 2021-10-26]. Dostupné z: https://docs.spring.io/spring-data/commons/docs/2.5.1/reference/html.

[45] POLLACK, M., RISBERG, T., GIERKE, O., LEAU, C., BRISBIN, J., DARIMONT, T., STROBL, C., PALUCH, M., BRYANT, J. Spring Data MongoDB – Reference Documentation. *Docs.spring*. [online]. 2021-05-14 [cit. 2021-10-26]. Dostupné z: https://docs.spring.io/spring-data/mongodb/docs/3.2.1/reference/html.

[46] MONGODB, INC. BSON Types – MongoDB Manual. *Docs.mongodb*. [online]. 2021 [cit. 2021-10-28]. Dostupné z: https://docs.mongodb.com/manual/reference/bson-types.

[47] MONGODB, INC. MongoDB Atlas Database | Multi-Cloud Database Service. *Mongodb*. [online]. 2021 [cit. 2021-10-28]. Dostupné z: https://www.mongodb.com/atlas/database.

8.3 Elektronické dokumenty

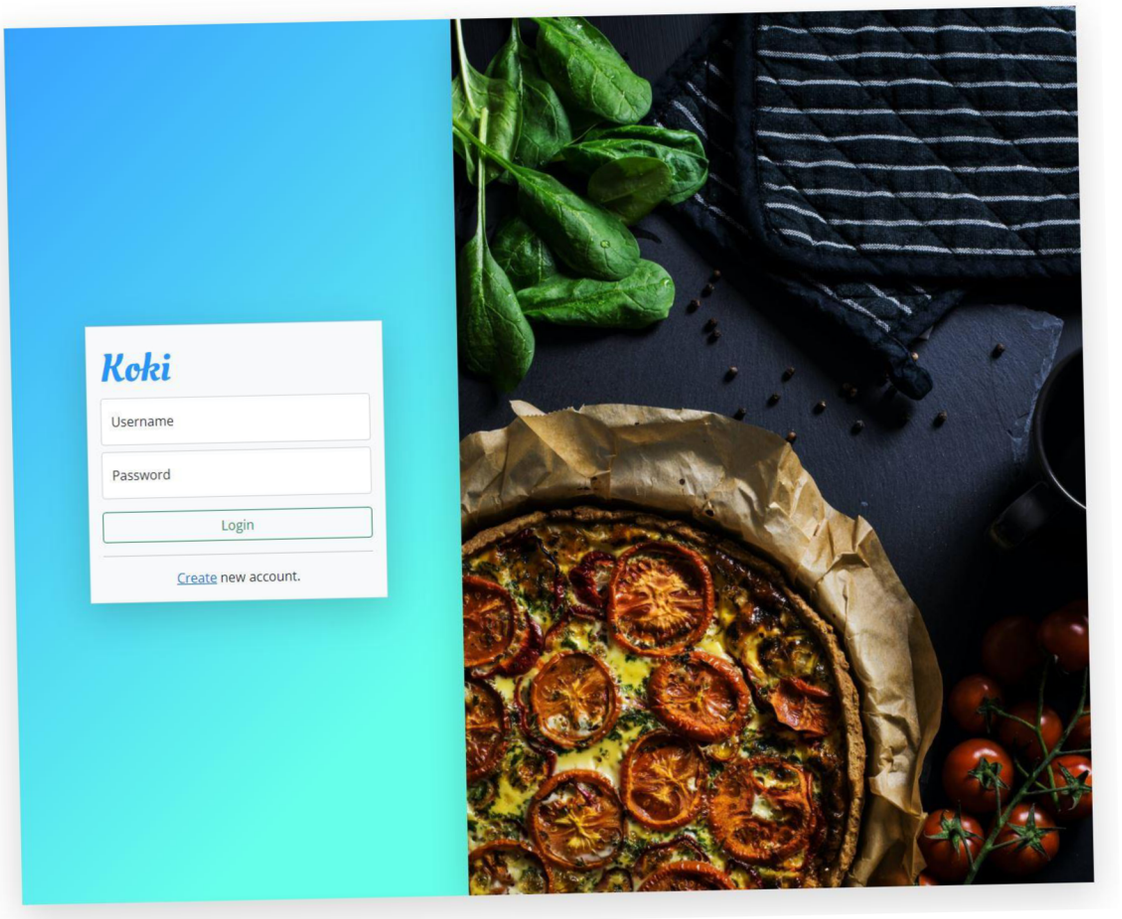
[48] MARTINEZ, M., MATEUS, B.G. *How and Why did developers migrate Android Applications from Java to Kotlin? A study based on code analysis and interviews with developers.* [online]. France: Université Polytechnique Hauts-de-France. 2020-03-28. [cit. 2021-01-25]. 29 s. Dostupné z: https://arxiv.org/abs/2003.12730

9 Přílohy

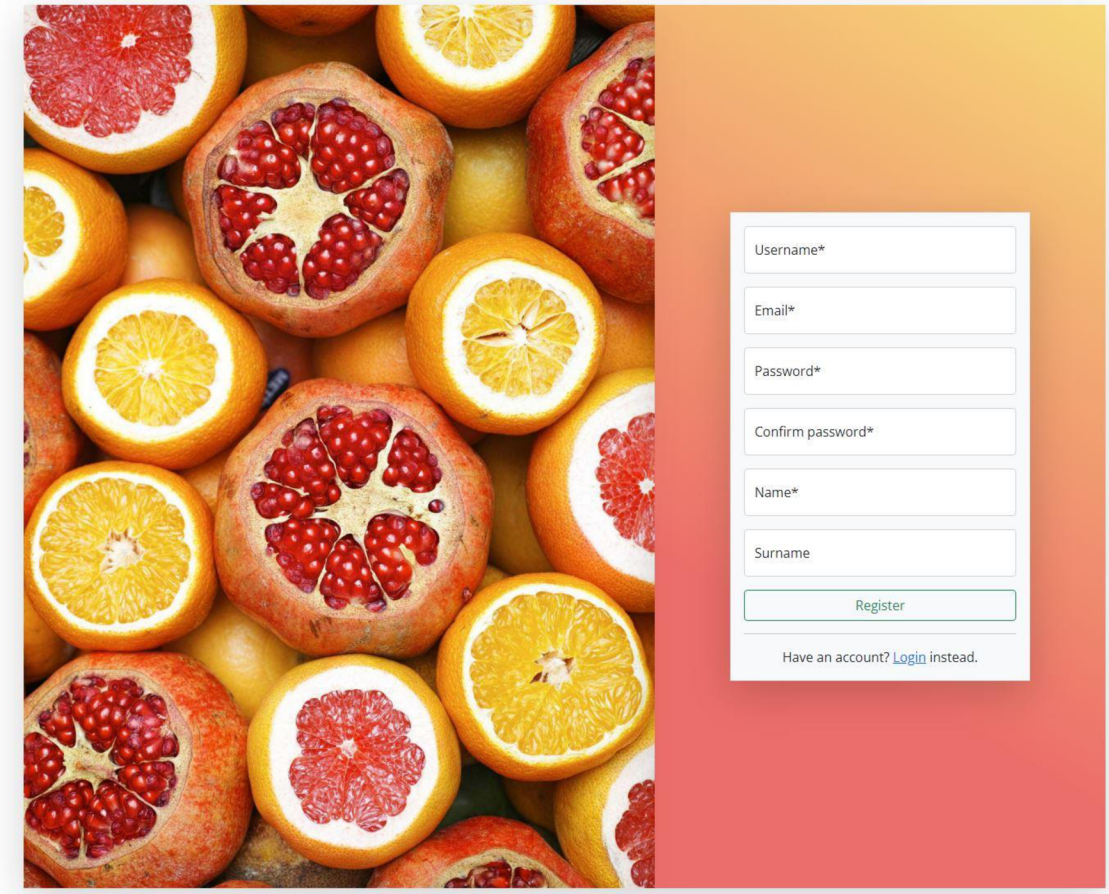
1. Podoba výsledné aplikace
2. Obrázky použité v aplikaci
3. Internetové recepty s obrázky použité v aplikaci
4. Zadání práce

9.1 Podoba výsledné aplikace

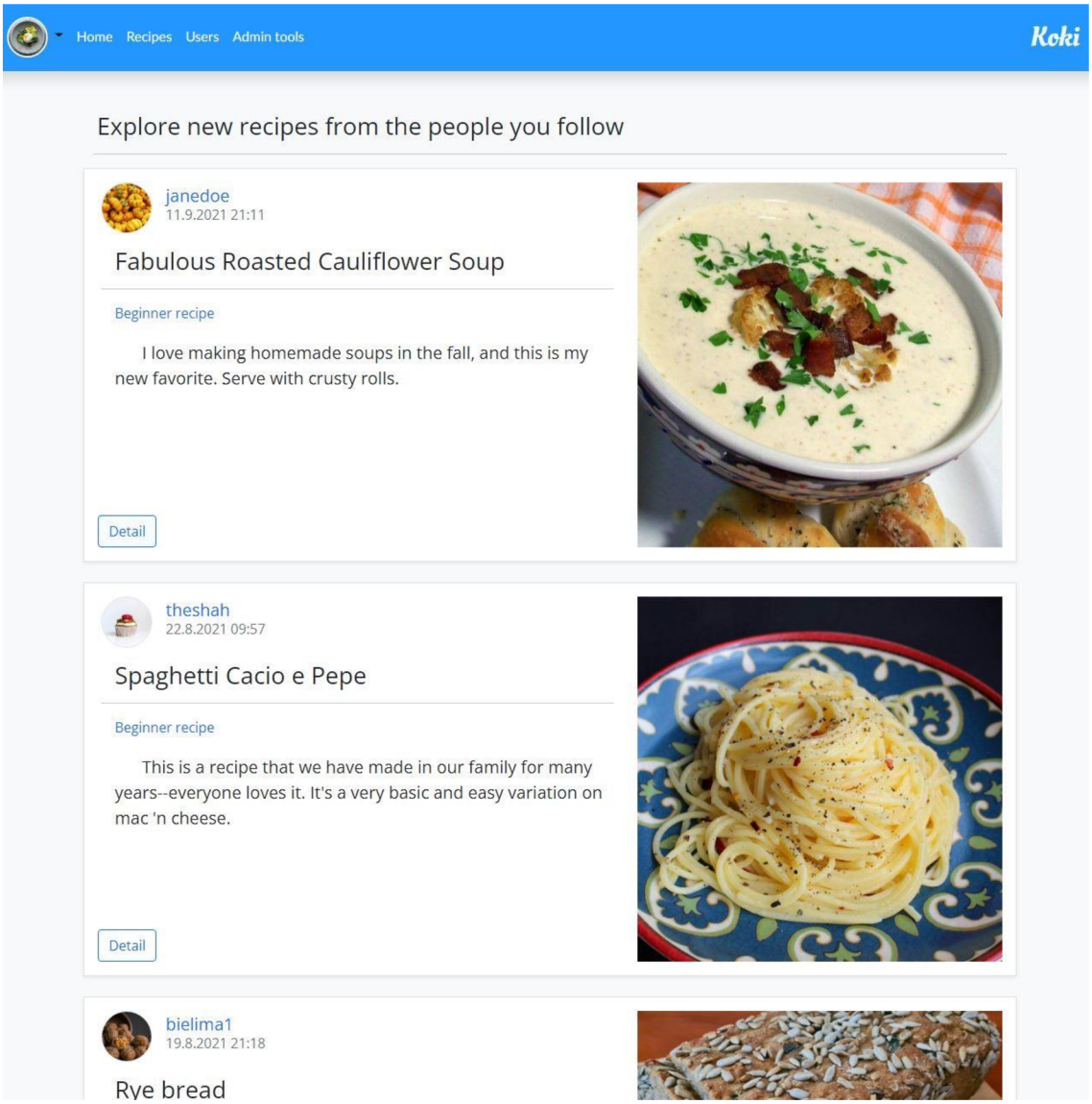
Zdroj: Autor



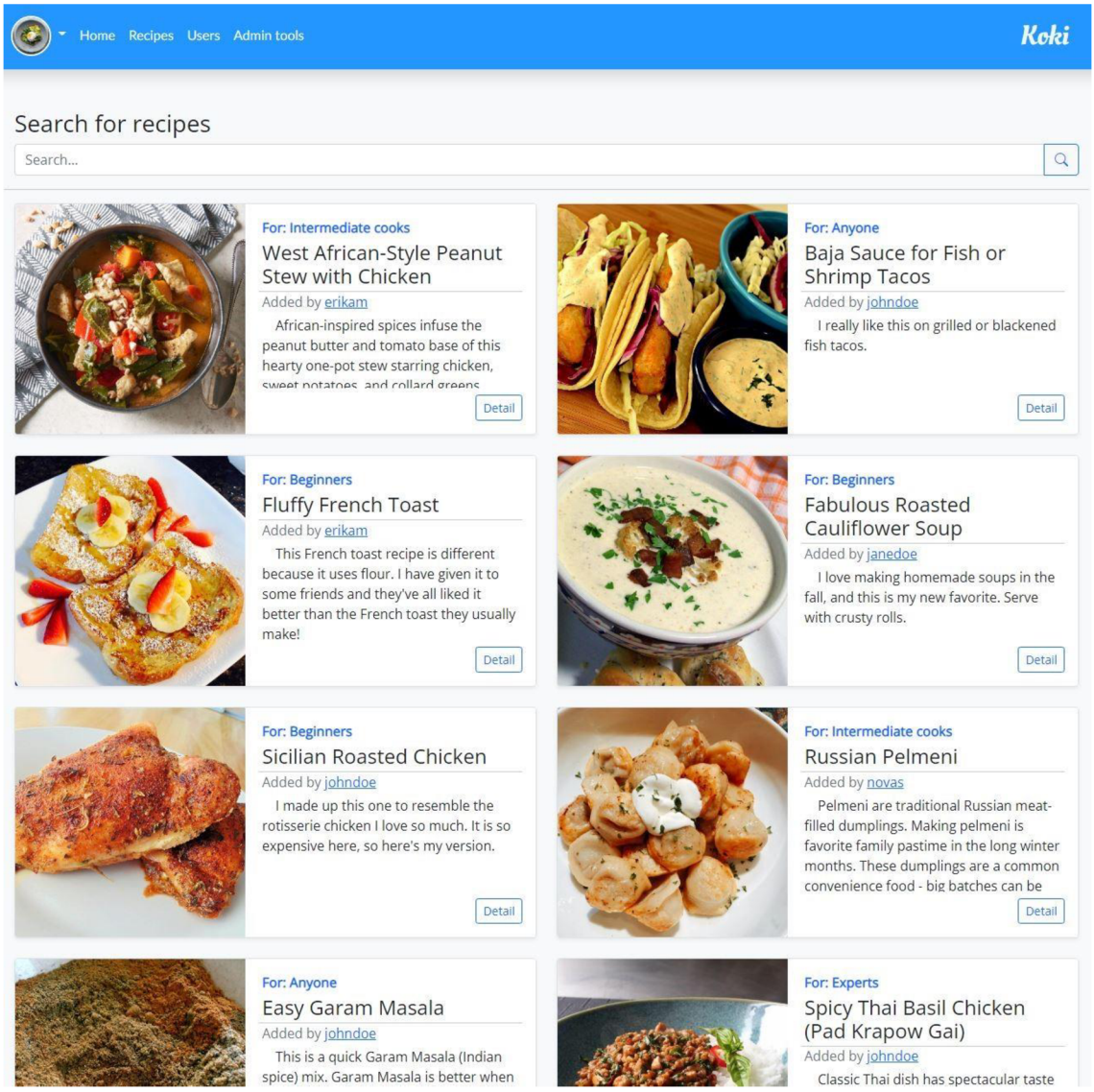
Obrázek 5: Stránka přihlášení



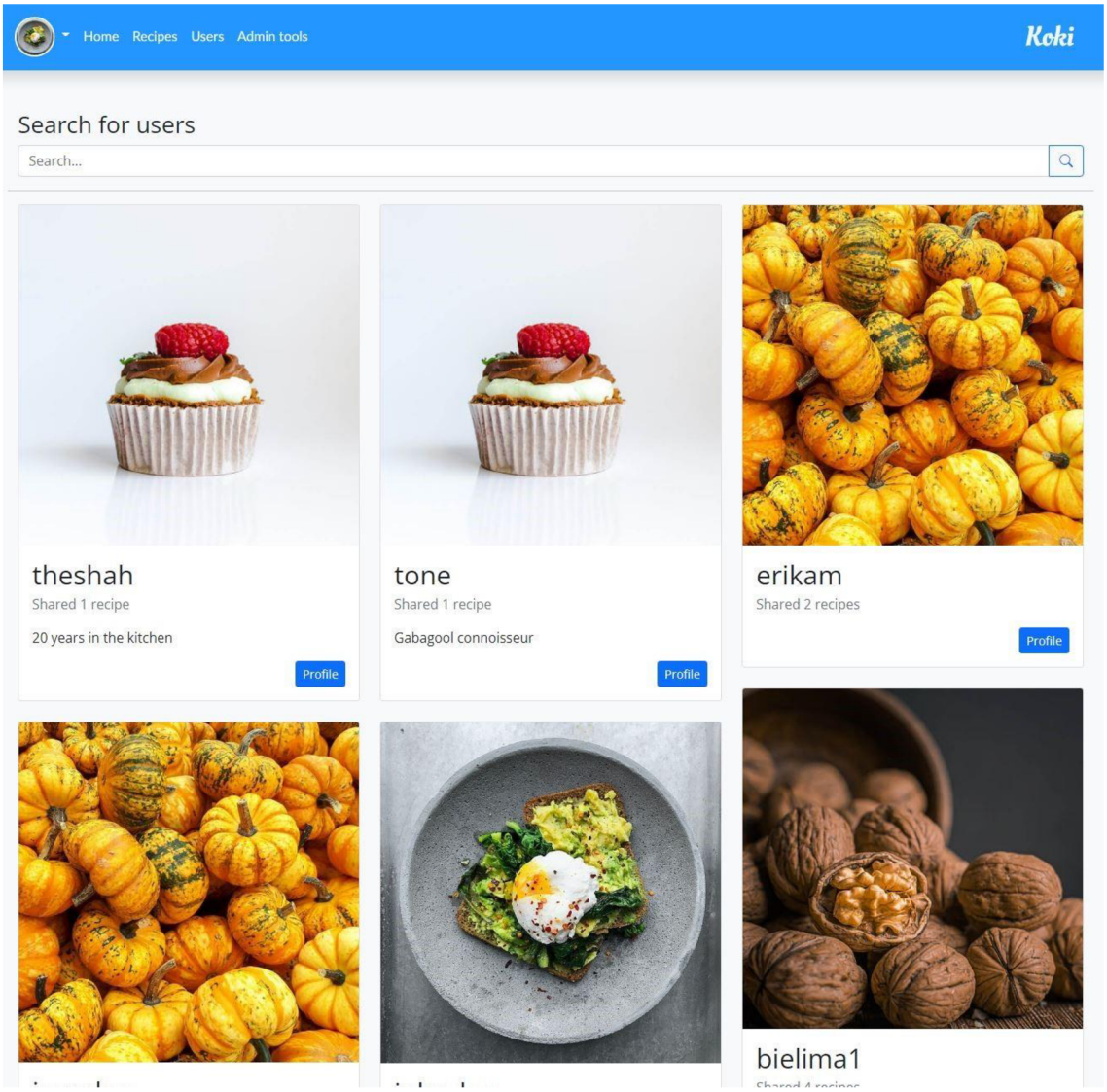
Obrázek 6: Stránka registrace



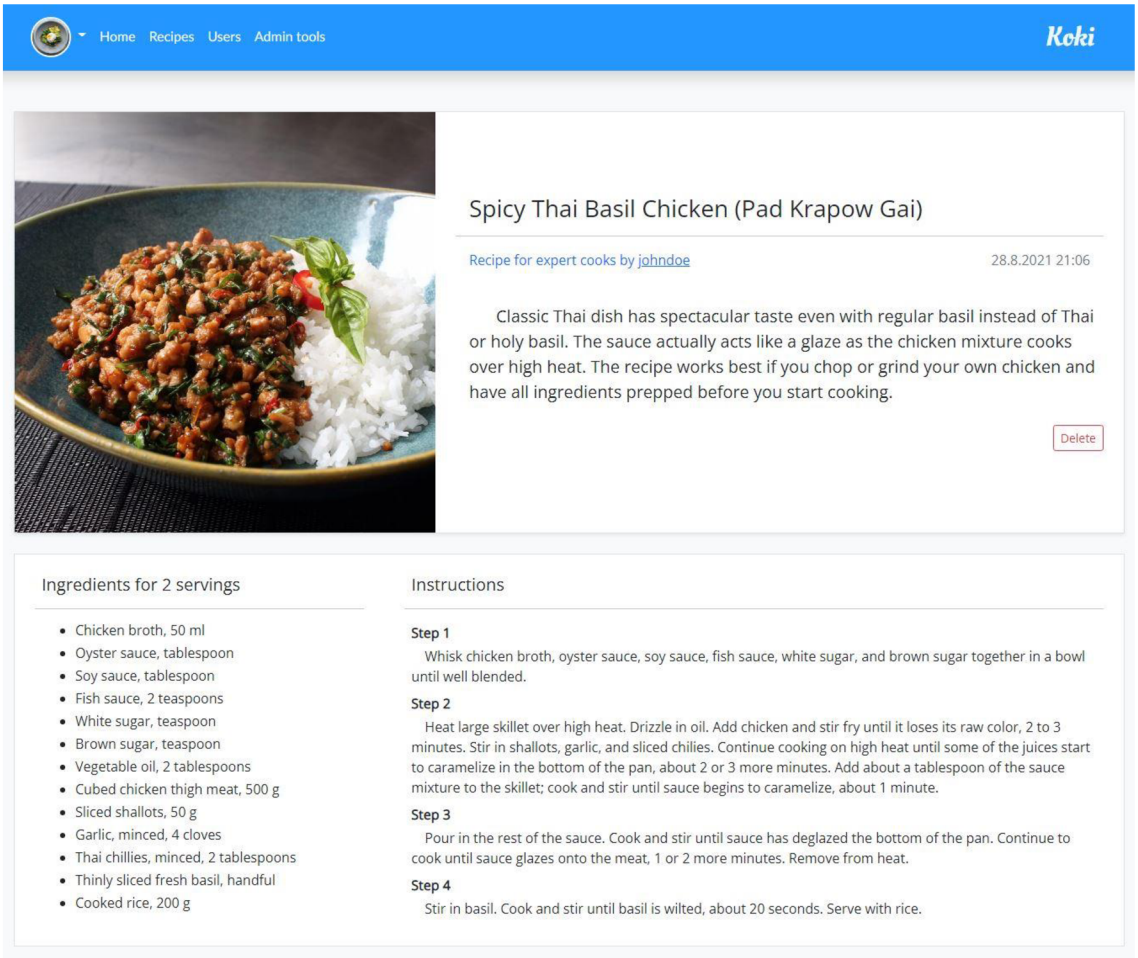
Obrázek 7: Domovská stránka



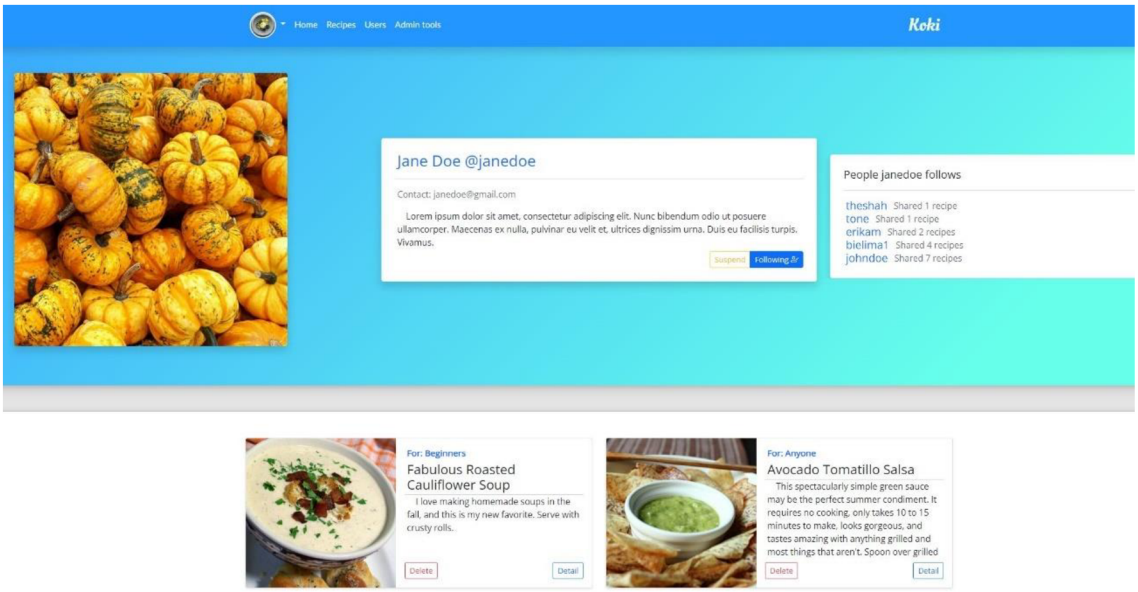
Obrázek 8: Přehled receptů



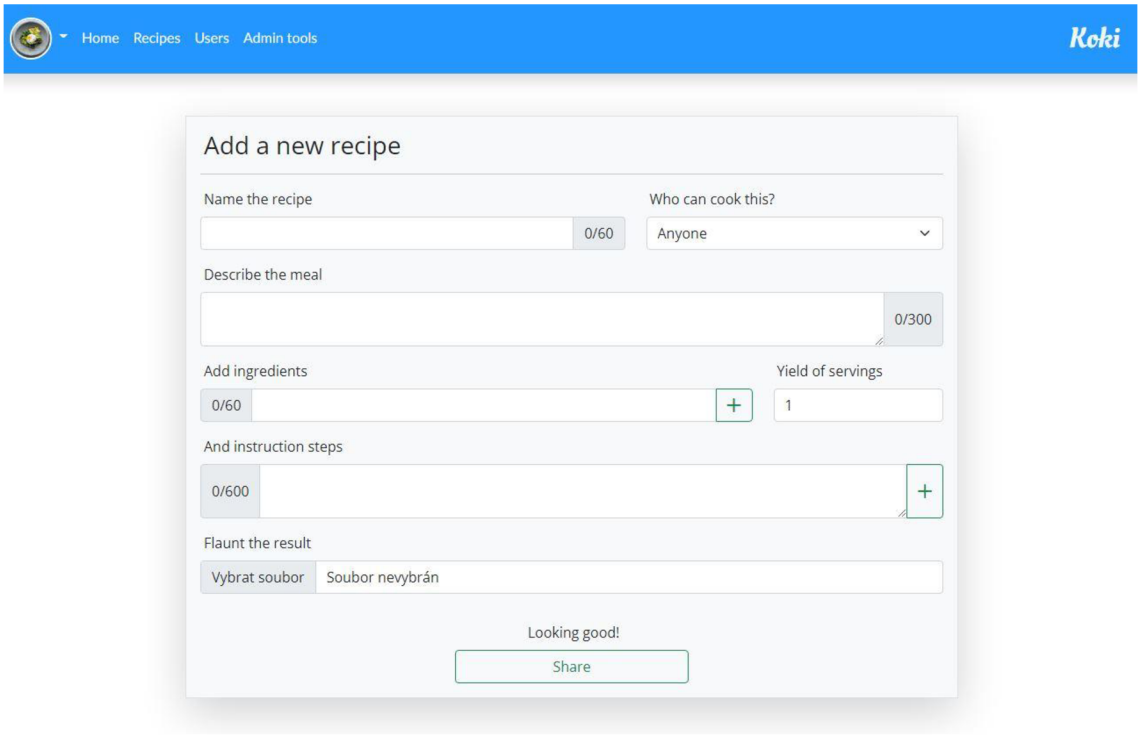
Obrázek 9: Přehled uživatelů



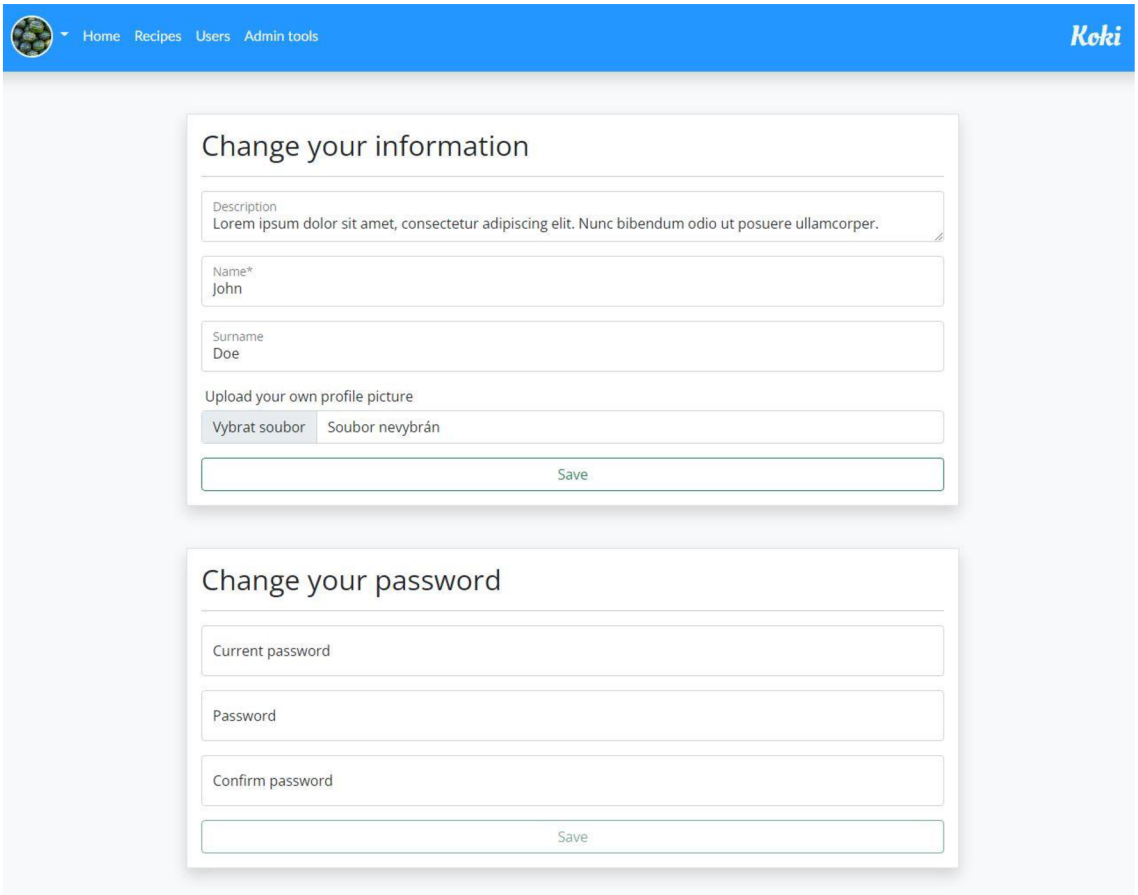
Obrázek 10: Detail receptu



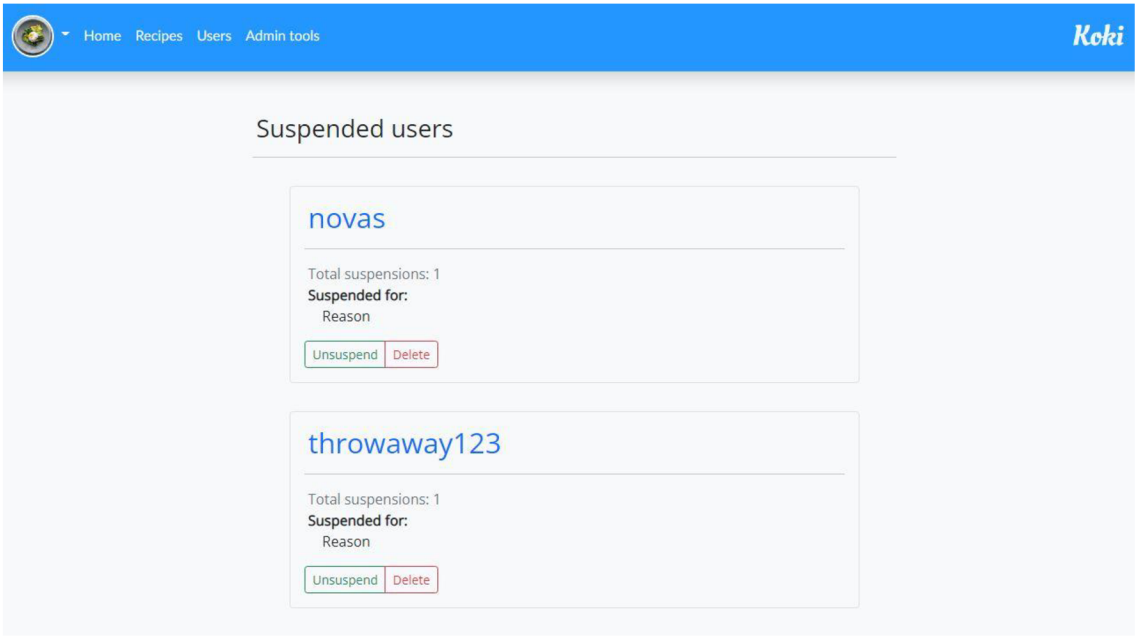
Obrázek 11: Detail uživatele



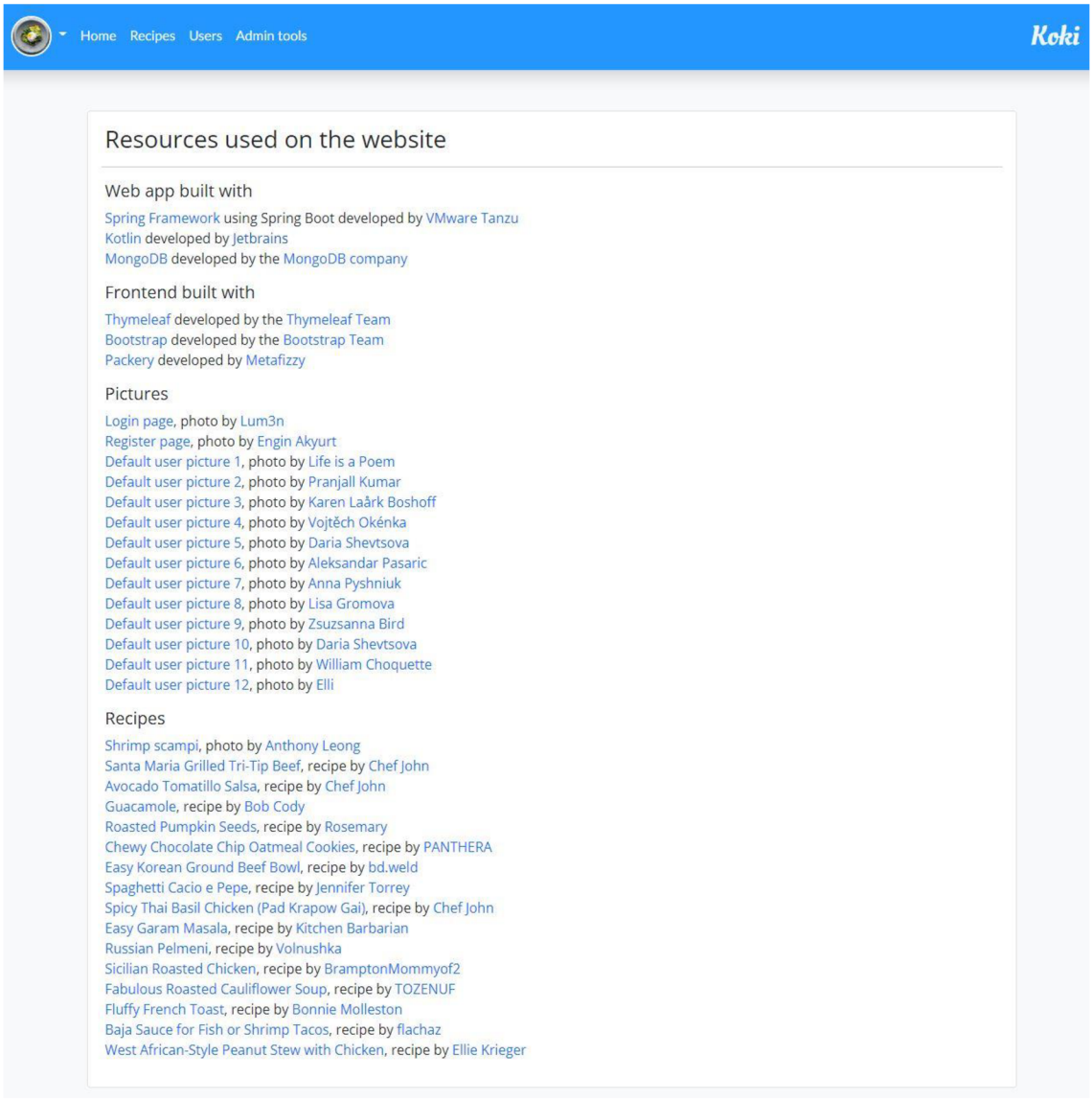
Obrázek 12: Přidání receptu



Obrázek 13: Úprava profilu



Obrázek 14: Pozastavené účty



Obrázek 15: Stránka zdrojů

9.2 Obrázky použité v aplikaci

1. *Pizza With Tomatoes on Black Surface*. Free Stock Photo [online]. In: *Pexels.* [online]. [cit. 2021-10-26]. Dostupné z: https://www.pexels.com/photo/pizza-with-tomatoes-on-black-surface-604969.
2. AKYURT, E. *Closeup Photo of Slice of Orange*. Free Stock Photo [online]. In: *Pexels.* [online]. [cit. 2021-10-26]. Dostupné z: https://www.pexels.com/photo/closeup-photo-of-slice-of-orange-1435735.
3. *Red Cherry Fruit in Close Up Photography*. Free Stock Photo [online]. In: *Pexels.* [online]. [cit. 2021-10-26]. Dostupné z: https://www.pexels.com/photo/food-field-summer-garden-5772261.
4. KUMAR, P. *Brown Garlic on Brown Bowel*. Free Stock Photo [online]. In: *Pexels.* [online]. [cit. 2021-10-26]. Dostupné z: https://www.pexels.com/photo/food-wood-fall-health-8705561.
5. BOSHOFF, K.L. *Black Berries in White Ceramic Bowl*. Free Stock Photo [online]. In: *Pexels.* [online]. [cit. 2021-10-26]. Dostupné z: https://www.pexels.com/photo/food-healthy-dawn-spoon-8281714.
6. OKÉNKA, V. *Photograph of Chocolate Cupcake With Red Strawberry Toppings*. Free Stock Photo [online]. In: *Pexels.* [online]. [cit. 2021-10-26]. Dostupné z: https://www.pexels.com/photo/photograph-of-chocolate-cupcake-with-red-strawberry-toppings-1055272.
7. SHEVTSOVA, D. *Carbonara in Grey Bowl*. Free Stock Photo [online]. In: *Pexels.* [online]. [cit. 2021-10-26]. Dostupné z: https://www.pexels.com/photo/carbonara-in-gray-bowl-1030947.
8. PASARIC, A. *Sliced Dragon Fruit on Pink Plate*. Free Stock Photo [online]. In: *Pexels.* [online]. [cit. 2021-10-26]. Dostupné z: https://www.pexels.com/photo/sliced-dragon-fruit-on-pink-plate-2907428.
9. PYSHNIUK, A. *Close-Up Shot of Leafy Vegetables on a Black Plate*. Free Stock Photo [online]. In: *Pexels.* [online]. [cit. 2021-10-26]. Dostupné z:

https://www.pexels.com/photo/close-up-shot-of-leafy-vegetables-on-a-black-plate-6083893.

10. GROMOVA, L. *Free stock photo of basket, berry, color.* [online]. In: *Pexels.* [online]. [cit. 2021-10-26]. Dostupné z: https://www.pexels.com/photo/food-pattern-texture-easter-7284760.
11. BIRD, Z. *Yellow and Green Pumpkins on Ground*. Free Stock Photo [online]. In: *Pexels.* [online]. [cit. 2021-10-26]. Dostupné z: https://www.pexels.com/photo/yellow-and-green-pumpkins-on-ground-5781644.
12. SHEVTSOVA, D. *Cooked Food*. Free Stock Photo [online]. In: *Pexels.* [online]. [Cit. 2021-10-26]. Dostupné z: https://www.pexels.com/photo/cooked-food-704569.
13. CHOQUETTE, W. *Cooked Food in Stainless Steel Plate*. Free Stock Photo [online]. In: *Pexels.* [online]. [cit. 2021-10-26]. Dostupné z: https://www.pexels.com/photo/cooked-food-in-stainless-steel-plate-2641886.
14. *Milk Chocolates*. Free Stock Photo [online]. In: *Pexels.* [online]. [cit. 2021-10-26]. Dostupné z: https://www.pexels.com/photo/milk-chocolates-1854664.
15. LEONG, A. *Shrimp Pasta Served on Grey Plate*. Free Stock Photo [online]. In: *Pexels.* [online]. [cit. 2021-10-26]. Dostupné z: https://www.pexels.com/photo/shrimp-pasta-served-on-gray-plate-2092906.
16. TENTIS, D. *Cooked Meat With Vegetables*. Free Stock Photo. [online]. In: *Pexels*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.pexels.com/photo/cooked-meat-with-vegetables-725991.
17. AUTOR. Jerk Chicken. 2021 [cit. 2021-10-26].
18. AUTOR. Whole Grain Bread. 2021 [cit. 2021-10-26].
19. AUTOR. Duck Breast with Rice Noodles. 2021 [cit. 2021-10-26].

9.3 Internetové recepty s obrázky použité v aplikaci

1. Santa Maria Grilled Tri-Tip Beef. *Allrecipes*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.allrecipes.com/recipe/236992/santa-maria-grilled-tri-tip-beef.
2. Avocado Tomatillo Salsa. *Allrecipes*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.allrecipes.com/recipe/233271/avocado-tomatillo-salsa.
3. CODY, B. Guacamole. *Allrecipes*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.allrecipes.com/recipe/14231/guacamole.
4. Roasted Pumpkin Seeds. *Allrecipes*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.allrecipes.com/recipe/13768/roasted-pumpkin-seeds.
5. Chewy Chocolate Chip Oatmeal Cookies. *Allrecipes*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.allrecipes.com/recipe/24445/chewy-chocolate-chip-oatmeal-cookies.
6. Easy Korean Ground Beef Bowl. *Allrecipes*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.allrecipes.com/recipe/268091/easy-korean-ground-beef-bowl.
7. TORREY, J. Spaghetti Cacio e Pepe. *Allrecipes*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.allrecipes.com/recipe/246628/spaghetti-cacio-e-pepe.
8. Spicy Thai Basil Chicken (Pad Krapow Gai). *Allrecipes*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.allrecipes.com/recipe/257938/spicy-thai-basil-chicken-pad-krapow-gai.
9. Easy Garam Masala. *Allrecipes*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.allrecipes.com/recipe/142967/easy-garam-masala.
10. Russian Pelmeni. *Allrecipes*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.allrecipes.com/recipe/259977/russian-pelmeni.
11. Sicilian Roasted Chicken. *Allrecipes*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.allrecipes.com/recipe/255263/sicilian-roasted-chicken.
12. Fabulous Roasted Cauliflower. *Allrecipes*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.allrecipes.com/recipe/151046/fabulous-roasted-cauliflower-soup.

13. MOLLESTON, B. Fluffy French Toast. *Allrecipes*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.allrecipes.com/recipe/16895/fluffy-french-toast.
14. Baja Sauce for Fis hor Shrimp Tacos. *Allrecipes*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.allrecipes.com/recipe/238599/baja-sauce-for-fish-or-shrimp-tacos.
15. KRIEGER, E. West African-Style Peanut Stew with Chicken. *Allrecipes*. [online]. [cit. 2021-10-26]. Dostupné z: https://www.allrecipes.com/recipe/276720/west-african-style-peanut-stew-with-chicken.

9.4 Zadání práce

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor: **Marek Bielik**

Studium: I1800156

Studijní program: B1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

Název bakalářské práce: **Vývoj aplikací v jazyku Kotlin a Spring frameworku**

Název bakalářské práce AJ: Application development in Kotlin language and Spring framework

Cíl, metody, literatura, předpoklady:

Cílem práce je představit jazyk Kotlin zejména ve zběžném srovnání s jazykem Java, dále představení Spring frameworku a následné využití těchto technologií pro tvorbu ukázkové webové aplikace. Vlastní aplikace bude mít podobu internetové kuchařky ve formě sociální sítě. V závěru práce bude zhodnocena kombinace Kotlinu a Spring frameworku pro tvorbu webových aplikací.

Osnova:

- Úvod
- Kotlin
- Spring
- Návrh a tvorba ukázkové aplikace
- Výsledná aplikace
- Shrnutí a hodnocení
- Závěr

Garantující pracoviště: Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Datum zadání závěrečné práce: 7.8.2020