Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informačních technologií

REST API pro podporu a integraci ITSM procesů

Diplomová práce

Autor: Bc. Lubomír Mrtvý
Studijní obor: Aplikovaná informatika

Vedoucí práce: Ing. Zuzana Němcová, Ph.D.

Hradec Králové srpen 2020

Prohlášení:
Prohlašuji, že jsem diplomovou práci vypracoval samostatně a uvedl jsem všechny použité prameny a literaturu.

V Hradci Králové dne 14. srpna 2020 Bc. Lubomír Mrtvý

Poděkování:

Děkuji vedoucí své diplomové práce Ing. Zuzaně Němcové, Ph.D. za vedení této práce.

Anotace

Diplomová práce je zaměřena na vývoj na platformě ServiceNow, návrh aplikací, práci s REST API a integraci ITSM procesů mezi instancemi. Čtenář je seznámen s platformou ServiceNow z pohledu základního uživatelského rozhraní, implementace ITSM procesů a základy vývoje na platformě, včetně základních specifik platformy. Ve spojitosti s tvorbou aplikací, je při návrhu aplikací, navrženo několik možných řešení, která jsou po zvážení prakticky využitelná, dle požadavků, které jsou na výsledné aplikace kladeny.

Výstupem diplomové práce je ucelený návrh architektury a použitelné prototypy aplikací pro řešení REST API a integrace ITSM záznamů, které je možné po úpravě konfigurací použít na jakékoliv instanci platformy ServiceNow.

Annotation

Title: REST API for support and integration of ITSM processes

The diploma thesis is focused on development on the ServiceNow platform, application design, work with REST API, and integration of ITSM processes between instances. The reader is acquainted with the ServiceNow platform from the point of view of the basic user interface, the implementation of ITSM processes, and the basics of development on the platform, including the basic specifics of the platform. Along with the creation of applications, when designing applications, several solutions are defined for suitable usability, as required by the final application.

The output of the diploma thesis is a proposal for architecture and usable application prototypes for REST API solutions and integration of ITSM records, which can be modified by the configuration, to be used on any instance of the ServiceNow platform.

Obsah

Úvod **1**

1 Cíle práce **2**

2 Platforma ServiceNow **3**
2.1 Představení platformy 3
2.2 Grafické uživatelské rozhraní 3
2.3 Seznamy 4
2.4 Formuláře 5

3 ITSM aplikace v ServiceNow **7**
3.1 ITSM v ServiceNow 7
3.2 Incident Management 7
3.2.1 Aktivace 8
3.2.2 Obsah 8
3.2.3 Životní cyklus Incidentu 9
3.3 Service Catalog 10
3.3.1 Catalog Item 10
3.4 Request Management 11
3.4.1 Obsah 11
3.5 Customer Service Management 13
3.5.1 Case management 13

4 Technologie **15**
4.1 Javascript 15
4.2 REST API 16
4.2.1 HTTP metody 16
4.2.2 JSON 17
4.3 JIRA 18
4.4 Verzování 18
4.4.1 Git 18
4.4.2 Update Set 19

5 Architektura platformy **20**

5.1 Javascript . . . 20
5.2 Databáze . . . 20
5.3 Aplikace . . . 21
5.3.1 Aplikační rozsah . . . 21
5.3.2 Globální rozsah . . . 21
5.4 Glide API . . . 21
5.4.1 GlideRecord . . . 21
5.4.2 GlideAggregate . . . 23
5.4.3 GlideSystem . . . 24
5.4.4 GlideDateTime . . . 26
5.5 Skriptování na straně serveru . . . 27
5.5.1 Script include . . . 27
5.5.2 Business Rule . . . 28
5.6 Skriptování na straně klienta . . . 29
5.6.1 Client Script . . . 29
5.6.2 UI Policy . . . 31
5.7 REST API . . . 31
5.7.1 Hlavičky . . . 31
5.7.2 Customizované dotazové parametry . . . 32
5.7.3 Dot-walking v REST API dotazech . . . 32
5.8 Skriptované REST API . . . 33
5.8.1 URL . . . 33
5.8.2 Zdroje . . . 34
5.9 Update Set . . . 34
5.9.1 Stavy . . . 34
5.9.2 Přenos mezi instancemi . . . 35

6 Analýza a návrh **36**

6.1 REST API . . . 36
6.1.1 Požadavky . . . 36
6.1.2 Možnosti řešení . . . 37
6.1.3 Návrh architektury . . . 41
6.1.4 Implementace . . . 47
6.2 Integrace . . . 61
6.2.1 Požadavky . . . 61
6.2.2 Možnosti řešení . . . 62
6.2.3 Návrh architektury . . . 65
6.2.4 Implementace . . . 67

7 Diskuze 72

7.1 Význam výsledků 72

7.2 Možnosti rozšíření 73

Závěr 74

Použité zkratky a pojmy 76

Seznam obrázků 77

Seznam tabulek 78

Seznam ukázek kódu 79

Bibliografie 80

Úvod

ServiceNow je nástroj pro účinné řízení a automatizaci firemních procesů a služeb. Jedná se o platformu americké společnosti ServiceNow, Inc., která je určená zejména k využití ve velkých zahraničních či nadnárodních společnostech. Primárním určením platformy je automatizace ITSM procesů, nicméně platforma samotná poskytuje mnohem více možností a je tedy využitelná v mnoha průmyslových odvětvích.

Struktura práce je rozdělena na dvě hlavní části. První částí je teoretické seznámení čtenáře s platformou, jejím prostředím a vývojem na platformě. Jsou zde popsány základní specifika platformy a specifické prvky a rozhraní, které jsou důležité pro implementaci aplikací na této platformě.

Druhou částí práce je praktická část, zaměřující se na návrh, možnosti řešení a implementaci aplikací pro práci s ITSM záznamy, zejména skrze REST API platformy. Jsou zde navržena možná řešení pro dosažení optimálních výsledků při tvorbě požadované funkcionality. Dále jsou jednotlivé návrhy posouzeny z hlediska optimálnosti řešení a jejich vlastností a nejvýhodnější řešení jsou dále blíže specifikována a posléze implementována.

Na závěr jsou posouzeny dosažené výsledky a deklarovány další možnosti rozšíření vytvořených řešení, vycházející nejen z návrhu aplikace, ale také z poznatků získaných při práci s vytvořenými prototypy.

Kapitola 1

Cíle práce

Diplomová práce si klade tři dílčí cíle. Prvním z cílů je seznámení čtenáře s platformou ServiceNow. S touto platformou se mnoho vývojářů v České republice příliš často nesetká. Cílem je seznámit čtenáře s prostředím platformy, prostředky pro zpracování ITSM požadavků, ale také se samotným vývojem na platformě a jejími specifiky.

Druhým z cílů je seznámení s nativním REST API platformy a možnostmi tvorby vlastních, skriptovaných API pro podporu ITSM procesů. Primárně jde o návrh a implementaci vlastního, jednoduššího řešení pro přístup k ITSM záznamům. Toto rozhraní je určeno k tvorbě aplikací pro správu ITSM požadavků, jako jsou např. monitorovací aplikace a případně také k integraci jak dalších instancí ServiceNow, tak i dalších aplikací třetích stran. Výsledné API musí nabízet jednoduchý a snadno rozšiřitelný přístup k záznamům a práci s nimi.

Posledním cílem je vytvoření návrhu a implementace integračního řešení, které využívá výše zmíněné API k synchronizaci ITSM záznamů mezi instancemi ServiceNow. Návrh integrace musí počítat s konfigurovatelností platformy a procesy, které jsou na synchronizované záznamy vázány.

Praktické cíle práce jsou zaměřeny na návrh a tvorbu použitelných prototypů REST API a integračního řešení. Výsledný návrh obou prototypů musí být jednoduše konfigurovatelný, rozšiřitelný a přizpůsobitelný proměnnému prostředí platformy ServiceNow.

Kapitola 2

Platforma ServiceNow

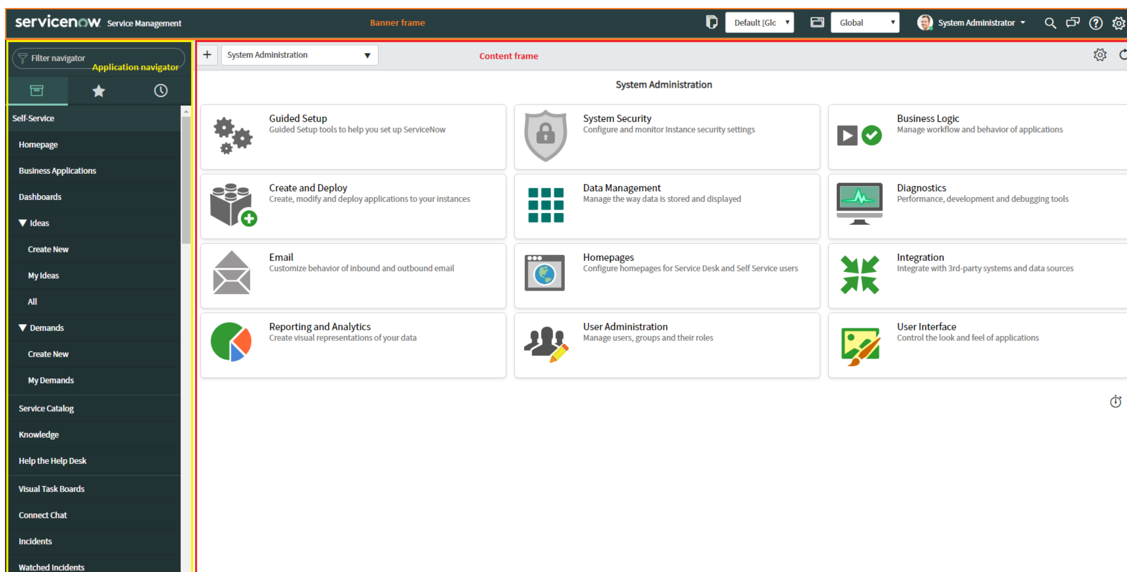
2.1 Představení platformy

ServiceNow je nástroj vyvíjený společností Service-Now a poskytovaný formou SaaS (*Software as a Service, volně přeloženo jako software jako služba*)[1]. Jde tedy o cloudovou platformu, která nevyžaduje na straně zákazníka nutnost budovat a udržovat datacentra ani kombinovat různá softwarová řešení pro management chodu firmy. ServiceNow poskytuje široké možnosti v oblastech ITSM, jako je například Incident management. Dále také nabízí velké množství vlastních řešení pro různá odvětví (např. HR) a také vysokou míru rozšiřitelnosti. Samozřejmostí je i automatizace procesů, díky které lze omezit nutnost zasahovat do procesu. Platforma také poskytuje možnost integrace s dalšími systémy pomocí webových služeb, REST API a zpracování emailových zpráv.

2.2 Grafické uživatelské rozhraní

Grafické rozhraní platformy ServiceNow je rozděleno do tří základních sekcí (viz. 2.1). **Banner frame** (Záhlaví) obsahuje základní ovládací prvky pro komunikaci, vyhledávání a konfiguraci rozhraní. **Application navigator** (Panel aplikací) umožňuje vyhledávat a procházet dostupné aplikační moduly, vytvářet záložky a trasovat historii navštívených položek. Tyto dvě sekce tvoří navigaci v rámci platformy. **Content frame** (Obsah) slouží k zobrazení seznamů a formulářů sloužících k práci s daty. Nejčastěji je používán v kombinaci s navigační částí rozhraní.

[1] dnes již, vzhledem ke komplexnosti, spíše PaaS (*Platform as a Service, volně přeloženo jako platforma jako služba*)



Obrázek 2.1: Grafické rozhraní platformy ServiceNow *(Vlastní zpracování)*

K databázovým datům je v rámci ServiceNow možno přistupovat unifikovaně, nezávisle na nízkoúrovňové implementaci uložení dat. Pro zobrazování a procházení většího množství záznamů nad jednotlivými tabulkami jsou v rámci platformy zvoleny seznamy. Pro práci s jednotlivými záznamy jsou pak implementovány formuláře.

2.3 Seznamy

Seznamy zobrazují sady záznamů z tabulky, počet záznamů na stránku je volitelný. V seznamech je možné vyhledávat, třídit, filtrovat a upravovat data. Dále mohou být hierarchické (obsahují podseznamy) a je možné je vložit do formulářů. (ServiceNow 2020l)

Rozhraní seznamu obsahuje záhlaví, filtry a drobečkovou navigaci (breadcrumbs), sloupce dat a zápatí. V záhlaví můžeme najít základní konfiguraci seznamu a s potřebným oprávněním i tlačítko pro tvorbu nového záznamu. Každý sloupec v seznamu odpovídá sloupci v tabulce a je možné pomocí něj záznamy třídit a seskupovat. (ServiceNow 2020l)

Filtrování na seznamu umožňuje vytvářet pokročilé po filtry pomocí řetězení jednotlivých podmínek, využívaní operací `AND` a `OR` a v případě potřeby je také možné využít v rámci podmínky jednoduchý JavaScript. Vytvořený filtr lze dále uložit, případně do schránky nakopírovat odkaz pro zobrazení seznamu s filtrem, ale také nakopírovat pouze zakódovanou podmínku, kterou lze využít v jiných částech systému.

Sloupce zobrazené v seznamu je možné globálně konfigurovat, určit, které sloupce jsou viditelné a v jakém pořadí. Každý uživatel může dále viditelnost a pořadí sloupců na seznamu přizpůsobovat svým potřebám. Toto přizpůsobení je však omezeno pouze na účet uživatele a je omezeno pouze na sloupce, které jsou součástí globální konfigurace.

2.4 Formuláře

Formuláře umožňují číst a zpracovávat jednotlivé záznamy z tabulky. Součástí formuláře mohou být nejen pole dané tabulky, ale také pole z referencovaných tabulek a žurnálů, jako například komunikace. Formulář je konfigurovatelný, je tedy možné měnit rozložení jednotlivých polí, rozdělovat pole do sekcí a záložek, případně pole skrýt. (ServiceNow 2020f)

Pole na formuláři lze skrývat dynamicky, případně je zobrazovat, měnit zda jsou povinná nebo zapisovatelná na základě předem definovaných podmínek. Dále je možné spuštění připravených skriptů při načtení formuláře nebo změně hodnoty pole.

Jak již bylo zmíněno, v rámci formuláře je možné využívat žurnálová pole pro komunikaci. Pro komunikaci je využíván *Activity log* (Seznam aktivit) na formuláři. Jednotlivé aktivity lze rozdělit do tří skupin (ServiceNow 2020f):

- **System activities** (Systémové aktivity, světle šedá barva) jsou záznamy o aktivitách a změnách dat. Tyto záznamy jsou tvořeny automaticky při sledovaných akcích a změnách.
- **Additional comments** (Další komentáře, tmavě šedá barva) jsou záznamy sloužící ke komunikaci se zadavatelem.
- **Work notes** (Pracovní poznámky, žlutá barva) slouží ke komunikaci mezi řešiteli, tyto záznamy nejsou viditelné pro koncového uživatele, zadavatele.

ITIL User Additional comments • 2018-11-14 05:49:39
SAP Sales app is not accessible

ITIL User Work notes • 2018-11-14 05:49:39
Researching this. I think there is an outage.

ITIL User Field changes • 2018-11-14 05:49:39

Assigned to	Beth Anglin
Configuration item	SAP Sales and Distribution
Impact	1 - High
Incident state	In Progress
Opened by	ITIL User
Priority	1 - Critical

Obrázek 2.2: Záznamy v Seznamu aktivit *(Vlastní zpracování)*

Na formuláři je také možné prohlížet a spravovat přílohy daného záznamu. Tyto soubory jsou uchovávané ve zvláštní struktuře, která následně obsahuje vazbu na daný záznam.

Dalšími prvky jsou také *Související odkazy* (Related links) a *Související seznamy* (Related lists). Odkazy v tomto případě mohou uživatele přesměrovávat na další užitečné záznamy nebo funkcionalitu, seznamy pak obsahují další záznamy, které souvisí s aktuálně zobrazeným záznamem. Tyto seznamy nejčastěji obsahují záznamy, které se na aktuální záznam odkazují, nicméně definice těchto seznamů jsou plně konfigurovatelné a mohou obsahovat záznamy, které splňují i jiné podmínky, než je přímá reference.

Kapitola 3

ITSM aplikace v ServiceNow

3.1 ITSM v ServiceNow

ITSM[1], volně přeloženo jako Správa IT služeb, se vztahuje na veškeré činnosti, které organizace provádí, aby navrhovaly, plánovaly, dodávaly, provozovaly a kontrolovaly IT služby nabízené zákazníkům.

Na rozdíl od technologicky orientovaných přístupů řízení IT se ITSM vyznačuje přijetím procesního přístupu se zaměřením na potřeby zákazníků a IT služby pro zákazníky.

Mezi hlavní oblasti ITSM procesů (důležitých pro naši aplikaci) patří:

- Incident Management
- Request Management
- Case Management

Výše zmíněné procesy jsou v rámci ServiceNow implementovány v rámci samostatných aplikací a rozšiřujících pluginů. Tato architektura umožňuje aktivovat pouze potřebnou funkcionalitu, případně získat rozšiřující funkce k základním procesům.

3.2 Incident Management

Cílem Incident managementu je „minimalizovat negativní dopad incidentů co nejrychlejším obnovením běžného provozu služby“(Axelos 2019). Jinak řečeno, obnovit normální provoz služeb v co nejkratším čase a zároveň minimalizovat dopad na obchodní operace, čímž se zajistí zachování co nejlepší úrovně kvality a dostupnosti služeb.

[1]IT Service Management

Incident samotný je definován jako „neplánované přerušení služby nebo snížení kvality služby“ (Axelos 2019).

3.2.1 Aktivace

Incident management je obsažen v aplikaci *Incident*, která je v rámci ServiceNow uložena pod ID *com.snc.incident*. Tato aplikace obsahuje pouze základní funkcionalitu Incident Managementu, je však rozšiřitelná pomocí dalších pluginů a plně upravitelná.

3.2.2 Obsah

ServiceNow poskytuje v rámci aplikace Incident účinnou implementaci Incident Managementu, poskytující efektivní řízení a správu zdrojů při zpracování různých typů incidentů. Základním stavebním prvkem této aplikace je Incident, tedy jeho záznam v databázi, obsahující důležité informace o incidentu, nabízející předdefinované volby pro klasifikaci. Dále také poskytuje propojení se znalostní databází poskytující přehled známých chyb a také dalšími ITSM implementacemi, jako například Problem a Change management.

Kategorizace incidentů slouží k určení oblasti, do kterého daný incident spadá a tím umožňuje lépe spravovat zdroje pro jeho řešení. Určení kategorie umožní přiřadit správnou skupinu řešitelů, případně přímo řešitele dané závady a tím zajistit efektivní řešení.

Prioritizace incidentu pomáhá určit dopad incidentu na fungování služeb a tím také určit prioritu, s jakou bude daný incident řešen. V rámci základní implementace je priorita určena pomocí předem definované tabulky, kde jsou vstupními parametry dopad závady a naléhavost řešení.

Stav incidentu umožňuje sledovat aktuální fázi řešení incidentu, mezi které patří jak samotné řešení tak i další stavy, jako například čekání na další informace, externího dodavatele nebo řešení v rámci jiné ITSM aplikace. Díky sledování stavu je možné efektivně sledovat aktuální množství řešených incidentů a zatížení jednotlivých řešitelů.

Propojení se znalostní databází umožňuje najít možné řešení pomocí již evidovaných řešení obdobných incidentů. Je tedy pomůckou jak pro řešitele ale také pro samotného zadavatele, který může nalézt řešení již v době zadávání nového incidentu.

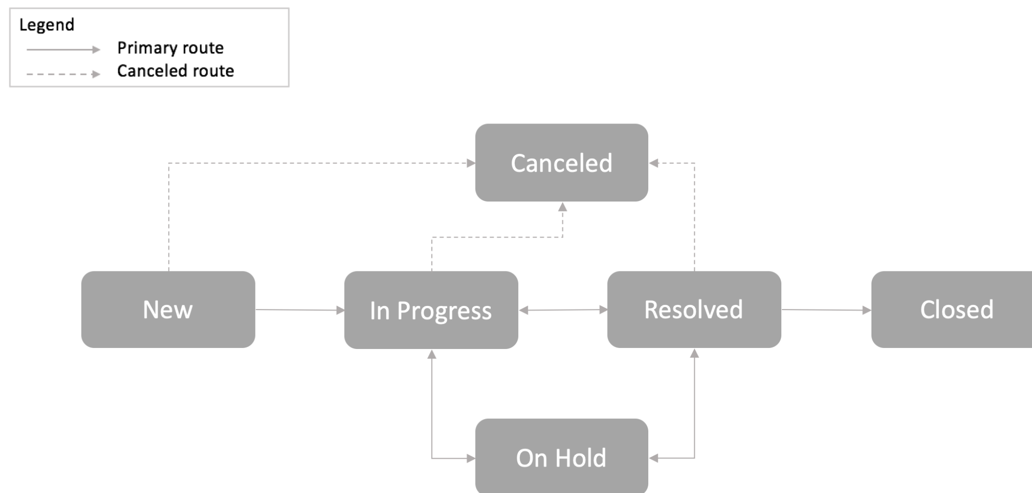
Obrázek 3.1: Formulář Incidentu v ServiceNow *(Vlastní zpracování)*

3.2.3 Životní cyklus Incidentu

Životní cyklus Incidentu je definován jednotlivými stavy a přechody mezi nimi. Základní konfigurace je zachycena na diagramu 3.2. Jak již bylo zmíněno, každý stav popisuje určitou fázi procesu zpracování Incidentu.

Stav **New** (Nový) odpovídá fázi založení Incidentu. Incident je zaevidován v systému, nicméně ještě nebyl přiřazen a není zpracováván. V této fázi je nejčastěji prováděna základní kategorizace a následné přiřazení na řešitele. Po přiřazení na řešitele přechází Incident do stavu **In Progress** (Probíhá), ve kterém je prováděna investigace a samotné řešení Incidentu. Z tohoto stavu je možné přejít do stavu **On Hold** (Odloženo), ve kterém je řešení přerušeno z důvodu dotazu na zadavatele, třetí stranu nebo návaznou operaci, případně do stavu **Resolved** (Vyřešeno), kde je řešení Incidentu navrhnuto zadavateli. Pokud je zadavatel s řešením spokojen, je Incident uzavřen, přechází tedy do stavu **Closed** (Uzavřeno).

Není-li zadavatel s řešením spokojen, může toto řešení odmítnout a Incident přechází zpět do stavu In Progress, případně On Hold. (ServiceNow 2020k)



Obrázek 3.2: Životní cyklus incidentu *(Převzato z ServiceNow (2020k))*

Incident přechází do stavu **Cancelled** (Zrušen) je-li klasifikován jako duplicitní, neodůvodněný nebo není incident jako takový. Do tohoto stavu je možné přejít z jakéhokoliv stavu, kromě stavu Closed.

3.3 Service Catalog

Service Catalog application (volně přeloženo jako aplikace Katalogu Služeb) je aplikace umožňující tvořit katalogy služeb poskytované zákazníkům k samoobslužnému řešení jejich požadavků. V rámci platformy je možné definovat více katalogů, např. podle poskytovaných služeb. Každý katalog je pak členě do kategorií v nichž jsou k dispozici jednotlivé Katalogové položky (angl. Catalog Items). (ServiceNow 2020r)

3.3.1 Catalog Item

Katalogová položka představuje zboží nebo službu, které je možné objednat skrze Katalog Služeb.Jednotlivé položky lze vystavovat v různých katalozích, případně omezit jejich viditelnost v rámci těchto katalogů pouze pro určité skupiny uživatelů. (ServiceNow 2020s)

Je-li nutné získat od uživatel nějaký vstup, je možné do položky přidat proměnné, které nesou tyto informace v rámci celého procesu zpracování. Proměnné je dále možné předvyplnit, validovat, případně skrývat, na základě předem definovaných podmínek.

Jednotlivé položky jsou zpracovávány podle předem definovaných postupů. Postup zpracování je řízen dvěma způsoby a to pomocí **Execution Plan** (volně přeloženo jako Exekuční plán) nebo **Workflow**. (ServiceNow 2020x)

Execution Plan je poměrně jednoduchý způsob řízení procesu. Tento způsob je využíván převážně pro jednodušší procesy, které nevyžadují velké dynamické zásahy do procesu. Jedná se primárně o určitou, jednoduchou posloupnost kroků, které jsou postupně vykonávány.

Workflow umožňuje vyšší možnosti dynamického řízení procesu. V rámci tohoto typu řízení je možné definovat složitější struktury pro rozhodování a dynamické generování úkolů, případně pro schvalování jednotlivých fází procesu. (ServiceNow 2020x)

3.4 Request Management

Aplikace Service Managementu využívá obecný proces Request Managementu (správy požadavků). Request Management umožňuje, aby jednotlivé katalogové položky mohly být objednávány a k nim vázaný proces splněn na základě definovaných postupů. (ServiceNow 2020m)

3.4.1 Obsah

Request Management aplikace využívá pro realizaci procesu strukturu založenou na třech základních tabulkách, tedy modelech dat:

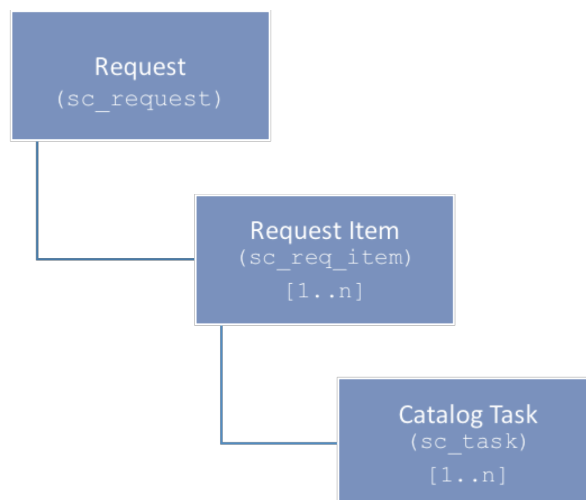
- Request (*sc_request*)
- Request Item (*sc_req_item*)
- Catalog Task (*sc_task*)

Proces je vázán na Katalog Služeb a je spuštěn pouze v případě objednání Katalogové položky. V okamžiku odeslání požadavku na objednání Katalogové položky jsou generovány záznamy podle hierarchie znázorněné na obrázku 3.3, nicméně tyto záznamy nejsou do databáze uloženy přesně v tomto pořadí.

Zjednodušeně je možné na proces vytváření záznamů nahlížet následovně:

1. Inicializace záznamu v tabulce Request (*sc_ request*), není uložen do databáze
2. Inicializace záznamu v tabulce Requested Item (*sc_ req_ item*), není uložen do databáze
3. Reference Request záznamu je uložena do záznamu Requested Item
4. K záznamu Requested Item jsou navázány proměnné z Katalogové položky, které jsou uloženy v pomocné tabulce
5. Záznam Requested Item tabulky je uložen do databáze
6. Záznam Request tabulky je uložen do databáze
7. V rámci procesu jsou vygenerovány Approvaly (Schvalovací úkoly, pokud jsou vyžadovány
8. Po schválení Approvalů (pokud byly vyžadovány) jsou vygenerovány a navázány Catalog Task záznamy

V rámci procesu je tedy práce primárně soustředěna do Catalog Task záznamů a jejich plnění. Do okamžiku vygenerování těchto záznamů je tedy proces z hlediska plnění služby soustředěn na schvalování požadavků a doplnění potřebných informací. Po vygenerování Catalog Task záznamů je možné tyto záznamy přiřadit jednotlivým skupinám či uživatelům k vykonání požadovaných úkolů.



Obrázek 3.3: Hierarchie Request Managementu *(Převzato z ServiceNow (2020m))*

3.5 Customer Service Management

Customer Service Management aplikace umožňuje poskytovat služby a podporu externím zákazníkům skrze komunikační kanály jako například web, email, chat telefon nebo sociální média. Aplikace tedy poskytuje nástroje pro řešení incidentů externích zákazníku, odpovědi na jejich otázky a celkově, pro poskytování servisních služeb. (ServiceNow 2020e)

3.5.1 Case management

Aplikace obsahuje velké množství prostředků pro poskytování služeb, řízení procesů a jejich optimalizaci. Mezi hlavní prvky patří **Case Management**. Case management umožňuje vytvářet Case záznamy (volně přeloženo jako Případ) a následně je řešit. Tyto záznamy mají vlastní typ, podle typu zákaznického požadavku. Dále je možné definovat prioritu pro řešení tohoto požadavku. K prioritě a typu požadavku jsou následně definovány tzv. **Service Level Agreements** (zkráceně SLA), které definují, v jakém časovém horizontu je nutné na tyto požadavky reagovat a jaký je limitní čas jejich řešení. Tyto časy jsou určeny na základě smlouvy se zákazníkem a je nutné se těchto limitů držet neboť jejich nedodržení může mít velký dopad na byznys zákazníka. (ServiceNow 2020c)

Case záznam se může nacházet v jednom z šesti stavů:

- New
- Open
- Awaiting Info
- Resolved
- Closed
- Cancelled

Case ve stavu **New** byl právě založen a zatím nebyl analyzován. V tomto stavu by neměl záznam zůstávat déle než je nutné. Mezní čas je určen SLA. Záznamy ve stavu **Open** jsou již analyzovány a pracuje se na nich, zákazník byl o tomto kroku již také informován. V případě nejasností, nutnosti doplnit informace nebo čekání na odpověď zákazníka, při které není možné dále pracovat je záznam přepnut do stavu **Awaiting Info**. Tento stav také pozastaví odpočet SLA. Je-li nalezeno řešení požadavku, je toto navrhnuto zákazníkovi a stav záznamu je přepnut do **Resolved**. V tomto stavu záznam zůstává, dokud není řešení odmítnuto nebo přijato. V případě přijetí řešení nebo uplynutí předem definované

doby, je stav změněn na **Closed** a řešení požadavku je úspěšně ukončeno. Je-li požadavek z nějakého důvodu nevalidní, např. duplicitní, je možné jej uzavřít přepnutím do stavu **Cancelled**. (ServiceNow 2020c)

Kapitola 4

Technologie

4.1 Javascript

JavaScript je objektově orientovaný skriptovací jazyk pro klientské a serverové aplikace. Původní jazyk měl být multiplatformním skriptovacím jazykem na straně klienta, který měl být vkládán přímo do HTML dokumentů. Jádro JavaScriptu se soustředí kolem skriptovacího jazyka, který byl standardizován asociací *ECMA*[1], přesněji standardem *ECMA-262*. (International 2020)

Přestože je Java součástí názvu jazyka JavaScript, není JavaScript od jazyka Java odvozený ani není jeho zjednodušenou verzí. Přestože oba jazyky sdílejí jistou syntaktickou podobnost, JavasScript je dynamicky typovaný. Na rozdíl od Javy tak může proměnná v JavaScriptu obsahovat hodnotu libovolného datového typu. Datový typ proměnné je tedy dán kontextem, ve kterém je právě používána. (T.Garfolo 2002)

Vykonávání kódu JavaScriptu a Javy se také liší. JavaScript není kompilovaný, ale interpretovaný jazyk. O překlad a spuštění kódu se tedy stará interpret jazyka.

JavaScript, jako takový, nedisponuje konceptem třída-instance, který je typický pro objektově orientované jazyky. Tento mechanismus je zde částečně nahrazen prototypováním. Tímto je dosaženo schopnosti simulovat mnoho principů a vlastností třídově založených funkcionalit. Mezi tyto prvky patří např. dědičnost, v případě jazyka javaScript tedy mluvíme o dědičnosti prototypové.

Dalším rozdílem oproti klasickým objektově orientovaným jazykům je také fakt, že JavaScript nerozlišuje definici funkce a metody. Funkce je možné volat jako metody, v tom případě je metoda svázána se svým objektem klíčovým slovem `this`.

[1] European Computer Manufacturers Association - Evropská asociace výrobců počítačů

4.2 REST API

REST[2] je softwarový architektonický styl, který definuje sadu omezení pro tvorbu webových služeb. Webové služby, které tomuto stylu odpovídají se nazývají RESTful webové služby a poskytují interoperabilitu mezi počítačovými systémy v rámci sítě internet. Tyto webové služby umožňují externím systémům přístup a manipulaci s textovou reprezentací zdrojů, tedy dat na straně serveru. Přístup a manipulace s daty je pomocí předdefinované sady bezstavových operací. (Fielding 2000)

Jedná se tedy architekturu rozhraní, navržená pro distribuované prostředí. REST jako takový je koncept návrhu distribuovaného systému, kdy jednotlivé části programu běží na samostatných strojích v rámci sítě, kterou využívají pro komunikaci. Programem v tomto případě může být například webová aplikace a za stroje tedy můžeme považovat např. webový prohlížeč a server.

Základními principy REST jsou:

- explicitní využívání HTTP metod
- bezstavovost
- zdroje (URI) mají adresářovou strukturu
- přenášená data jsou nejčastěji ve formátu XML, JSON, HTML

Data jsou na straně serveru reprezentována kolekcemi, klient tedy nepřistupuje přímo ke zdroji dat, ale pouze k jejich reprezentaci.

4.2.1 HTTP metody

REST využívá základní HTTP metody k provádění jednotlivých operací na straně serveru. Tyto metody je možné využít pro práci s celou kolekcí nebo, v případě zadání identifikátoru, k práci s jedním prvkem kolekce. Mezi nejpoužívanější metody pak patří:

- GET
- POST
- PUT
- DELETE

[2] Representational state transfer

Metoda **GET** je určena pro získání dat. Při volání bez identifikátoru se očekává vrácení seznamu prvků z kolekce, s identifikátorem je očekáván detail o prvku kolekce.

POST je určena pro uložení nových dat na server. Tato metoda využívána pro vytvoření nové kolekce nebo prvku kolekce.

Pro úpravu dat na serveru je určena funkce **PUT**. Takto je opět možné modifikovat celou kolekci nebo jen její prvek. Očekávaným využitím metody **DELETE** je odstranění prvku či celé kolekce.

Vzhledem k tomu, že REST jako takový je pouze styl, tedy koncept, pro návrh API, je i využití těchto metod v praxi často uvažováno pouze jako doporučení, což může vést k neočekávanému chování některých implementací.

4.2.2 JSON

Pro implementaci a práci s rozhraním pracujícím s daty je nutné tato data jednoznačně reprezentovat. V tomto ohledu se stal velmi výhodným dnes již široce rozšířený formát JSON. Výhodou tohoto formátu je snadná čitelnost.

Reprezentace dat v tomto formátu umožňuje přenos textu, číselných údajů, pravdivostních hodnot polí i objektů. Data jsou v tomto případě reprezentována formou textového řetězce s jasně danou strukturou `"klíč": "hodnota"`. (Crockford n.d.[a])

```
1  {"menu": {
2    "id": "file",
3    "value": "File",
4    "popup": {
5      "menuitem": [
6        {"value": "New", "onclick": "CreateNewDoc()"},
7        {"value": "Open", "onclick": "OpenDoc()"},
8        {"value": "Close", "onclick": "CloseDoc()"}
9        ]
10 }}}
```

Ukázka kódu 4.1: Ukázka formátu JSON *(Převzato z (Crockford n.d.[b]))*

4.3 JIRA

JIRA je softwarový nástroj vyvíjený společností Attlassian. Tento nástroj je určen k evidenci chyb a problémů při vývoji softwaru nebo řízení projektů. Podobně jako ServiceNow poskytuje JIRA možnost zadávání, předávání a sledování servisních záznamů, stejně tak jako poskytování zpětné vazby. (Atlassian 2020)

Software JIRA dále slouží k řízení procesů v rámci vývojových procesů SCRUM a Kanban.

4.4 Verzování

Verzování slouží k uchování historie všech změn provedených v konfiguraci aplikace. Ve většině případů se jedná o historii pouze ve zdrojových kódech a konfiguraci, verzovat však lze i data.

Ke sledování těchto změn slouží tzv. VCS[3], který obvykle eviduje kým a jak byl zdrojový kód změněn. Díky tomuto je možné zjistit přesný stav zdrojového kódu sledovaných záznamů a v případě potřeby vrátit stav zpět.

Verzovací systém umožňuje hlídat změny a pomáhá řešit případné konflikty v případě, že dochází ke změně ve zdrojových kódech z více stran současně.

4.4.1 Git

Git je distribuovaný systém správy verzí, vytvořený Linusem Torvaldsem, pro vývoj jádra Linuxu. Původně byl navržen jako nízkoúrovňový nástroj, postupně se však rozrostl do kompletního systému a je tedy použitelný bez dalších doplňkových nástrojů. (Organization n.d.)

Projekty v rámci nástroje Git jsou uchovávány v tzv. repozitářích. Pro vývoj a jednotlivé úpravy pak slouží větve, tzv. branches. Hlavní větví je *master*, z níž vycházejí hlavní vývojové větve. Tyto větve jsou následně po dokončení vývoje opět spojeny s hlavní větví, pomocí tzv. merge.

Platforma ServiceNow mimo svůj vlastní verzovací nástroj nabízí možnost využití systému Git prostřednictvím *Source control integration.* (ServiceNow 2020t).

[3]Version control system - systém správy verzí

4.4.2 Update Set

Update set (volně přeloženo jako Sada aktualizací) je skupina konfiguračních změn, kterou je možné přenášet mezi instancemi ServiceNow. Tato vlastnost umožňuje administrátorům seskupit provedené konfigurační změny v rámci pojmenované sady změn a tuto sad pak přesunout jako celek na jinou instanci pro testování nebo deployment do produkčního prostředí. Detailněji jsou Update sety popsány v sekci 5.9 Update Set. (ServiceNow 2020u)

Kapitola 5

Architektura platformy

5.1 Javascript

V rámci ServiceNow je aktuálně využíván JavaScript modul využívající standard ECMAScript5 a to od verze Helsinki. Pro podporu existujících skriptů a nových skriptů vyvinutých podle standardu ECMAScript5 má JavaScript modul dva režimy:

- Režim kompatibility pro skripty před verzí Helsinki a globální skripty
- Režim standardu ES5 pro skripty aplikací

JavaScript modul dynamicky určuje, který režim se použije.

5.2 Databáze

Data v ServiceNow jsou uchovávána v tabulkách, uživatel je však od databázové vrstvy oddělen pomocí Glide Server API. Koncový uživatel je tedy od databáze odstíněn a není nutná znalost jazyka SQL, ani typu databáze, ve které jsou data uložena. Uživatele tedy nemusí zajímat, zda jsou data uložená v databázi MySQL, PostgreSQL nebo Oracle, přístup k nim je pro něj vždy jednotný.

Pro všechny záznamy je v databázi určen jednotný jedinečný identifikátor `sys_id`. Tento sloupec je tedy součástí každé tabulky a obsahuje 32-znakové GUID[1]. Sys_id slouží jako primární klíč a je hojně využíváno k referencování záznamů mezi tabulkami.

[1] Globally Unique ID - Globálně unikátní identifikátor

5.3 Aplikace

Funkcionalita, kterou je možné získat z oficiálních zdrojů ServiceNow, případně je vytvářena v rámci platformy jinými vývojáři, je ve většině případů „zabalena“ formou funkčních balíčků, aplikací. Pro oddělení této funkcionality od zbytku platformy a také aplikací jako takových, je v rámci ServiceNow využíván tzv. Scope, tedy rozsah platnosti dané aplikace či její části. Tento rozsah je pak dvojí:

- Aplikační (Application scope)
- Globální (Global scope)

5.3.1 Aplikační rozsah

Aplikační rozsah umožňuje chránit aplikaci určením rozsahu a omezením přístupu k aplikačním souborům a datům. Díky této vlastnosti je možné, aby administrátor určil, které části aplikace jsou dostupné pouze v rámci aplikace a které jsou naopak přístupné i z jiných aplikací. Nastavení přístupu je pak možné jak pro aplikační záznamy, tak i pro každou tabulku dané aplikace. (ServiceNow 2020a)

5.3.2 Globální rozsah

Globální rozsah je specifickou formou aplikačního rozsahu. Aplikace vytvořené v tomto rozsahu jsou primárně vytvořeny tak, aby bylo možné k nim a jejich datům přistupovat ze všech aplikací v globálním rozsahu.

5.4 Glide API

ServiceNow poskytuje poměrně velké množství API pro interakci s platformou a daty. Níže popsaná API jsou pouze výběrem těch nejpoužívanějších. Stejně tak jsou zde popsány pouze základní metody každého rozhraní. (ServiceNow 2019)

5.4.1 GlideRecord

GlideRecord je speciální třída jazyka Java[2], kterou lze v JavaScriptu v ServiceNow použít přesně tak, jako by to byla nativní JavaScript třída. Objekt této třídy slouží k práci s daty v rámci ServiceNow a umožňuje provádět databázové operace. Jednotlivé operace jsou

[2] třída GlideRecord.java

definovány pomocí funkcí a není tedy nutné využívat klasické SQL dotazy. (ServiceNow 2020i)

Inicializace objektu je prováděna pomocí konstruktoru třídy `GlideRecord()`. Parametrem tohoto konstruktoru je systémové jméno tabulky. Výsledek dotazu je uložen v objektu této třídy jako seznam objektů, jejichž atributy jsou sloupce dané tabulky.

Zadávání dotazů je prováděno pomocí volání předdefinovaných funkcí, mezi nejpoužívanější patří:

- addQuery()
- addencodedQuery()
- query()
- get()

Funkci `addQuery()` je možné volat se 2 nebo 3 parametry. Při použití 3 parametrů předáváme trojici *atributu*, *operátor*, *hodnota*. Volání se 2 parametry nevyužívá explicitní operátor, dotaz se bude chovat jako by byl zadán operátor pro rovnost. Několikanásobné volání této funkce řetězí podmínky pomocí operace *AND*.

Funkce `addEncodedQuery()` přijímá pouze jeden parametr a to dotaz v zakódovaném tvaru. Výhodou je možnost zadání složitějších dotazů v rámci jednoho volání, nevýhodou pak může být menší přehlednost dotazu. Obecně se používání této funkce omezuje na nezbytné minimum.

Volání `query()` je nejčastěji prováděno bez parametru, k provedení dotazu specifikovanému pomocí výše uvedených funkcí. Druhou možností je volání se 2 parametry, dvojicí *atribut*, *hodnota*. Tato funkce nemá návratovou hodnotu.

Pomocí funkce `get()` můžeme získat záznam, příp. záznamy, které odpovídají zadaným parametrům *atribut*, *hodnota*. V případě volání s jedním parametrem je tento parametr předpokládán jako unikátní identifikátor (sys_id). Funkce **get()** má na rozdíl od výše zmíněné `query()` návratovou hodnotu a indikuje, zda byl odpovídající záznam nalezen či nikoliv.

Jak již bylo zmíněno, výsledek dotazu je uložen jako seznam objektů. Procházení tohoto seznamu je realizováno pomocí cyklického volání funkce `next()`, která iteruje přes všechny prvky výsledného seznamu.

Na výsledcích dotazu lze provádět další operace, nejčastěji se jedná o čtení a zápis hodnot. Vzhledem k objektové reprezentaci dat je možné využít přístup k atributům pomocí tečkové notace, bezpečnější a obecně doporučovaný přístup je pomocí vestavěných funkcí:

- getValue()
- setValue()

Použití `getValue()` vyžaduje zadání parametru a to jména atributu jehož hodnota má být navrácena. Obecně se doporučuje číst hodnoty pomocí této funkce, jelikož má pevně daný datový typ návratové hodnoty. Funkce tedy vrací textový řetězec, což u použití tečkové notace nemusí být vždy pravda a to hlavně v mezních případech.

Pro nastavení hodnoty je vhodné využít funkci `setValue()` volanou se 2 parametry, kterými jsou *atribut* a *hodnota*.

```
1 var incidentGR = new GlideRecord("incident");
2 incidentGR.addQuery("priority", 5);     //priority 5 - Planning
3 incidentGR.addQuery("state", 6);        //state Resolved
4 incidentGR.query();
5
6 while(incidentGR.next()){
7   incidentGR.setValue("state", 7);    //state Closed
8   incidentGR.update();
9 }
```

Ukázka kódu 5.1: Ukázka práce s GlideRecord API *(Vlastní zpracování)*

Speciálním rozšířením této třídy je pak třída *GlideRecordSecure*, která vynucuje užití ACL, tedy pracuje s vyšší mírou zabezpečení jak při čtení tak při zápisu.

5.4.2 GlideAggregate

GlideAggregate je rozšířením funkcionality třídy GlideRecord umožňující jednoduchou tvorbu agregačních dotazů. Pro běžné dotazy umožňuje využití využití agregačních funkcí: COUNT, SUM, MIN, MAX, AVG. (ServiceNow 2020g)

Pro zadání, agregaci a čtení hodnot jsou implementovány následující funkce:

- addAggregate()
- getAggregate()
- groupBy()

Pro zadání agregace je nejčastěji využita funkce `addAggregate()`. Prvním parametrem je název agregační funkce, tento parametr je povinný. Druhým, nepovinným, parametrem je jméno sloupce pro agregaci, není-li zadán je výchozí hodnota *NULL*. Při zadání sloupce pro agregaci je výsledek pro agregační funkci seskupen právě podle tohoto sloupce.

Pro práci získání výsledku námi zadané agregační funkce slouží metoda `getAggregate()`. Tato metoda má dva parametry, prvním je název agregační funkce, druhým je sloupec, pro který byla daná agregace definována.

Další používanou metodou je `groupBy()`, která umožňuje seskupit záznamy podle sloupce, jehož jméno je předáno jako parametr. Tuto metodu lze volat vícenásobně pro seskupení podle více než jednoho sloupce.

Díky těmto metodám je možné vytvářet i poměrně složité agregace. Tyto agregace jsou výhodné zejména pro vytváření reportů, sumarizaci dat a další analytické funkce.

```
1 var count = new GlideAggregate('incident');
2 count.addQuery('active','true');
3 count.addAggregate('COUNT','state');
4 count.query();
5 while(count.next()){
6   var state = count.state;
7   var stateCount = count.getAggregate('COUNT','state');
8   gs.log("Incidents in state "+ state +": "+ stateCount);
9 }
```

Ukázka kódu 5.2: Ukázka práce s GlideAggregate API *(Vlastní zpracování)*

5.4.3 GlideSystem

Rozhraní GlideSystem poskytuje řadu metod pro pohodlné získání a zpracování dat ze systému. API není nutné před použitím manuálně inicializovat, je ve všech částech serverového rozhraní dostupné v globální proměnné `gs`. Toto rozhraní poskytuje metody pro práci s datumy (např. aktuální datum, začátek/konec dne/týdne/měsíce/kvartálu, porovnávání datumů), logování, práci s aktuálním uživatelem, načtení zpráv a konfiguračních záznamů atd. (ServiceNow 2020j)

Nejčastěji se setkáváme s metodami:

- getUserID()
- hasRole()
- getMessage()
- getProperty()
- log()

Metoda `getUserID()` slouží k získání sys_id aktuálního uživatele. Tato metoda je často využívána pro indentifikaci uživatel v rámci skriptů a podmínek nebo pro zalogování interakcí pro auditování operací.

V rámci ověřování přístupů mimo klasické ACL definice je možné ověřit, zda má uživatel určitou roli přímo přes GlideSystem. K tomuto slouží metoda `hasRole()`, která je volána s jedním parametrem, tedy s názvem ověřované role. Metoda vrací hodnotu *true/false* podle toho, zda má aktuální uživatel tuto roli přiřazenou. Vyjímkou je pak uživatel s rolí *admin*, pro takového uživatele vrací metoda vždy hodnotu *true*.

Vzhledem k lokalizaci platformy je výhodné nemít veškeré texty uložené jako proměnné přímo v proměnných, které využíváme ve skriptech. Pokud jsou texty takto uložené, není možné tyto texty překládat podle aktuální lokalizace. Metoda `getMessage()` umožňuje načítat lokalizované texty ze systému, případně do takovýchto textů (jsou-li vhodně formátované) doplňovat dynamické proměnné. Metodu je možné volat se dvěma parametry, prvním je textový klíč, podle kterého je odpovídající text vyhledán. Druhým, nepovinným, parametrem je seznam dynamických proměnných, které mají být do vráceného textu vloženy. Výhodou tohoto volání je možnost jazykových mutací textů, neboť texty jsou uloženy v tabulce *sys_ui_message*, kde je pro daný klíč možné uložit text a definovat jazyk, v jakém je text napsán. Při volání metody je pak pro daný klíč vrácen text, který odpovídá jazyku přihlášeného uživatele.

Pro větší variabilitu a přehlednost systému je vhodné mít některá, často opakovaně používaná, data uložena na jednom místě, odkud je možné je načíst v jakékoli části systému. Ve většině případů mluvíme o velmi malých datech, nejčastěji konfiguračních, která jsou potřeba na různých místech systému a jejich změna musí být jednoduchá a nesmí narušit integritu systému. pro taková data slouží tabulky *sys_properties*. Zde je možné ukládat data odpovídající předkonfigurovaným datovým typům a následně je načíst v jakékoli serverové části systému voláním metody `getProperty()`, jejímž prvním parametrem je klíč takto

uložené hodnoty a nepovinným parametrem je alternativní hodnota, která bude použita, pokud pro daný klíč neexistuje záznam. Metoda takto maskuje složitější vyhledávání a je možné ji využít i v podmínkách a skriptech, čímž v mnoha případech docílíme zkrácení a zpřehlednění kódu.

```
1 //  Vypis jmena instance pomoci getProperty()
2 gs.log('Instance name: ' + gs.getProperty('instance_name'));
3
4 //  Vypis pomoci dohledani v databazi
5 var propertyGR = new GlideRecord('sys_properties');
6 if(propertyGR.get('name','instance_name')){
7     gs.log('Instance name: ' + propertyGR.getValue('value'));
8 }
```

Ukázka kódu 5.3: Využití gs.getProperty() *(Vlastní zpracování)*

Metoda `log()` umožňuje zápis textu do systémových logů. Metoda má 2 parametry, prvním je zpráva, které má být zapsána, druhým parametrem je pak název zdroje, ze kterého zprávu logujeme, pro větší přehlednost logů. Druhý parametr není povinný a často je ignorován.

5.4.4 GlideDateTime

Třída GlideDateTime poskytuje metody pro práci s objekty reprezentujícími datum a čas. Na rozdíl od metod API GlideSystem, které pracuje s datem ve formátu textového řetězce, GlideDateTime pracuje s objektovou reprezentací. Rozhraní umožňuje vytvářet objekty z textového zápisu, pokud splňuje základní formát data a času. Dále je možné pracovat s časovými zónami, měnit čas po milisekundách, sekundách, dnech, týdnech, měsících či rocích, provádět operace porovnávání a substrakce, případně extrahovat pouze datum či čas. (ServiceNow 2020h)

Obdobnou funkcionalitu pak nabízí i rozhraní GlideDate a GlideTime, která zde však nebudou dále popisována.

5.5 Skriptování na straně serveru

Serverové skripty běží na serveru nebo nad databází a mohou měnit vzhled nebo chování platformy, případně běží formou tzv. Business rules při přístupu nebo změně záznamů.

Funkcionalita je v rámci systému uzavřena v okamžitě volaných funkcích. Toto zajišťuje určitou míru separace a zaručuje, že skript nezasahuje do jiných částí systému jako například přepisem globálních proměnných. Dále je možné takovýmto funkcím v rámci systému předávat potřebné parametry a případně dohledat volání funkce v rámci ladění.

5.5.1 Script include

Script include (dále jen SI) je databázový záznam uchovávající skript, který je následně spouštěn na serveru. Tento záznam může svým obsahem reprezentovat funkci nebo třídu, tak jak ji známe z objektově orientovaného programování. (ServiceNow 2020p)

V případě definice třídy je důležité, aby název třídy odpovídal názvu SI. Pokud uživatel tvoří nový SI, systém v okamžiku zadání jména SI automaticky doplní šablonu těla nově tvořené třídy. V rámci systému lze pak tuto třídu vyvolat klasickou inicializací, jak ji známe z jazyka JavaScript, tedy pomocí konstruktoru ve formě názvu třídy. (ServiceNow 2020p)

Takto definované třídy lze pomocí funkcionality SI libovolně aktivovat a deaktivovat a tím dávají možnost aktivovat a deaktivovat části funkcionality systému.

Pro dodělení funkcionality je dále možné omezit viditelnost těchto tříd v rámci aplikace, případě udělat tyto skripty globální, tedy dostupné ve všech aplikacích.

```
1  var NewClass = Class.create();
2
3  NewClass.prototype = {
4      initialize: function () {
5      },
6
7      myFunc: function () {
8          //  telo funkce
9      },
10
11     type: 'NewClass'
12 };
```

Ukázka kódu 5.4: Třída deklarovaná v rámci SI *(Vlastní zpracování)*

5.5.2 Business Rule

Business rule (dále jen BR) je server-side skript, tzn. skript, který je vykonáván na straně serveru. Tento skript je spuštěn při splnění předem definovaných podmínek, tyto podmínky jsou podle typu BR ověřovány v okamžiku, kdy je záznam zobrazován, vkládán, upravován nebo mazán, případně v okamžiku, kdy je nad tabulkou prováděno dotazování. (ServiceNow 2020b)

BR jsou s výhodou využívány pro spouštění automatizovaných úloh. Kromě automatické změny dat v záznamu je možné také spouštět události, které jsou následně zpracovány, a také definovat vlastní skripty, které mohou vykonávat změny v záznamu samotném, případně pracovat se záznamy jiných tabulek.

Spuštění BR závisí na dvou základních kritériích, přesněji na **čase spuštění** a **databázové operaci**.

Čas spuštění udává v jakém okamžiku má být logika BR vykonána:

- Before
- After
- Async
- Display

Before a **After** BR jsou spouštěny *před*, případně *po* provedení databázové operace. Použitím těchto podmínek je tedy možné upravit data před provedením operace, tedy např. ověřit zda je vstup validní případně vstup naformátovat nebo naopak po provedení operace provést zápis do jiné tabulky.

Async BR jsou na rozdíl od předchozích spouštěny asynchronně. Takováto BR je tedy vykonána současně s databázovou operací. Asynchronní BR jsou často využívány pro provádění přidružených operací, které mohou být časově náročnější a zároveň na ně nejsou plně vázány operace, které BR inicializovaly.

Display BR jsou využívány pro úpravu dat, která jsou prezentována uživateli na formuláři. Pomocí tohoto typu BR je možné data získaná z databáze upravit před zobrazením a řídit tak tedy, co uživatel vidí.

Čas spuštění je nutné určit pro každou BR. Zároveň je možné vybrat jen jednu z výše uvedených voleb pro danou BR. Na rozdíl od časových podmínek je možné zvolit více než jednu databázovou operaci, při které má být BR spuštěna. Operace, pro které je možné tyto podmínky definovat jsou:

- Insert
- Update
- Query
- Delete

Insert, **Update** a **Delete** BR se využívají ke spouštění skriptů před, po nebo současně s databázovými operacemi pro práci s daty. Jmenovitě jde o operace vkládání, upravování a mazání dat.

Dále je zde dostupná volba operace **Query**. Podmínka definovaná v Query BR je vyhodnocována během dotazování do databáze. Nejčastěji se využívá s časovou podmínkou *Before*. Takto definovaná BR je často označovaná jako *Before query BR*, která je hojně používána pro úpravu dotazování. S její pomocí lze doplnit podmínky pro vyhledávání v databázi a je možné např. omezit uživateli přístup k neaktivním záznamům, případně dovolit procházet jen záznamy přiřazené uživateli, který volání do databáze inicializoval.

5.6 Skriptování na straně klienta

Skriptování na straně klienta je v rámci ServiceNow určeno primárně k manipulaci se zobrazovanými daty a validací vstupů před odesláním formuláře. Nejčastěji se lze setkat s *Klientskými skripty* (Client Script) a *UI Policy*. Vzhledem k povaze těchto skriptů je vhodné brát v potaz jejich funkcionalitu při práci s daty mimo formuláře, tedy v serverových skriptech.

5.6.1 Client Script

Client Script umožňuje systému spustit JavaScript kód v prohlížeči na straně klienta na základě klientem vyvolané akce. Tyto skripty jsou používány pro konfiguraci formulářů, jejich polí a hodnot v nich. (ServiceNow 2020d)

Pomocí Client Scriptu lze:

- zobrazit/skrýt pole
- umožnit zapisovatelnost/nezapisovatelnost pole
- určit, zda je pole povinné/nepovinné
- nastavit hodnotu pole na základě ostatních polí
- upravit volby v seznamech na základě uživatelské role
- zobrazovat zprávy na základě hodnot v poli

Pro definici Client Scriptu je důležité určit, jaká akce tento skript spustí, čímž je určen i typ Client Scriptu. Akce, které je možné sledovat jsou následující:

- onLoad()
- onSubmit()
- onChange()
- onCellEdit()

onLoad() skript je spuštěn v okamžiku načtení formuláře předtím, něž je uživateli umožněno přistoupit k datům. Ve většině případů je tento typ využíván k manipulaci s aktuálním formulářem nebo k nastavení výchozích hodnot.

Pro ověření dat v rámci odesílání formuláře je využíván `onSubmit()` skript, který provede validaci vstupů na formuláři v okamžiku odesílání formuláře a v případě nutnosti, tedy v případě nevalidních vstupů, lze toto odesílání zrušit.

V rámci Client Scriptu lze také definovat akce, které budou provedeny v okamžiku změny hodnoty některého z polí. Tyto akce jsou prováděny pomocí `onChange()` skriptu. Tento skript vyžaduje definovat, nad jakým polem mají být změny monitorovány. Při změně monitorovaného pole je poté provedena akce definovaná skriptem.

Zatímco první tři typy jsou spouštěny pouze na formulářích, `onCellEdit()` skript je definován pro spuštění při změně hodnoty pole na listu. tento skript je vhodné využít s kombinací s onChange() skriptem pro dané pole na formuláři k zaručení konzistence dat.

5.6.2 UI Policy

UI Policy slouží k dynamické změně chování informací na formuláři a řízení procesních flow. Tato funkcionalita umožňuje nastavit současně několik polí na formuláři jako nezapisovatelné, případně povinné nebo naopak části formuláře skrýt. Základní UI Policy se obejdou bez skriptování, nicméně, pro více pokročilé akce, je možné přidat skriptovanou logiku. Pro všechny zmíněné akce je možné využít i Client Script, pro rychlejší načítání je však vhodné využívat primárně UI Policy. (ServiceNow 2020v)

5.7 REST API

Platforma ServiceNow poskytuje velké množství REST API, která jsou ve výchozím stavu aktivní a je možné je hned používat. Mezi hlavní z těchto API patří *Table API*, které poskytuje rozhraní pro *CRUD* operace, tedy vytvářet, číst, měnit a mazat záznamy existujících tabulek. (ServiceNow 2020n)

Výchozí ServiceNow API dodržuje standard REST API protokolu a dále poskytuje také „customizované“ URI a dotazové parametry k zajištění zpětné kompatibility a poskytnout dodatečnou funkcionalitu, jako například stránkování v seznamech výsledků.

5.7.1 Hlavičky

ServiceNow REST API podporuje používání HTTP různých hlaviček, z nichž některé jsou povinné pro specifické HTTP metody nebo koncové body. Pro většinu ServiceNow REST API jsou to hlavičky *Accept* a *Content-Type*, které slouží ke správnému formátování dat, v tomto případě jsou podporovány:

- `Accept: application/json, application/xml`
- `Content-Type: application/json, application/xml`

Každé volání navíc může obsahovat hlavičku autentizace pro ověření uživatele.

5.7.2 Customizované dotazové parametry

Mnoho OOTB (Out-of-the-box) REST API v rámci ServiceNow využívá custom parametry, které umožňují stránkovat velké objemy dat, filtrovat výsledky nebo omezit počet vrácených záznamů v rámci jednoho dotazu:

- sysparm_query
- sysparm_limit
- sysparm_offset
- sysparm_fields
- sysparm_view

Pro filtrování výsledných dat je výhodné použití parametru **sysparm_query**, kterému je předán zakódovaný dotaz, jehož součástí mohou být jako filtrační podmínky, tak i operace pro řazení výsledků. Ve výchozím stavu instance vyhodnocuje celý dotaz a pokud narazí na nevalidní část, ignoruje ji. Toto chování je ovlivněno nastavením proměnné *glide.invalid_query.returns_no_rows*. Pokud je tato nastavena na hodnotu *true*, nejsou v případě nevalidního dotazu vrácena žádná data.

Pro výběr polí jednotlivých vrácených záznamů slouží parametry **sysparm_fields** a **sysparm_view**. První z těchto parametrů je předáván formou seznamu polí, oddělených čárkou. Pouze tato pole, pokud existují, jsou následně součástí výsledku dotazu. Součástí druhého parametru je název pohledu, který je použit pro výsledná data.

Parametr **sysparm_limit** určuje maximální počet záznamů, který bude navrácen po vyhodnocení dotazu. Bez použití tohoto parametru pro volání na velké kolekce, případně při zvolení příliš vysoké hodnoty parametru, mohou mít dotazy velký dopad na výkon systému.

Pro základní stránkování je určen parametr **sysparm_offset**. Tento parametr určuje počáteční index, od kterého se budou data vrácena. Volající zde může využít také referencí jako *first*, *next* a *last*. Ve výchozím stavu je parametr nastaven na 0, v případě, že byl nastaven uživatelem, měla by jeho hodnota být nezáporné číslo.

5.7.3 Dot-walking v REST API dotazech

Dot-walking je forma přístupu k referencovaným či jinak strukturovaným datům. Jedná se přístup pomocí tzv. tečkové notace, která umožňuje procházet data v různých úrovních dané struktury. jednotlivé úrovně struktury jsou odděleny právě tečkami. Pokud tedy např.

záznam uživatele obsahuje odkaz na oddělení, pod které uživatel patří, lze k jménu oddělení přistoupit tečkovou notací ve formě *uzivatel.oddeleni.jmeno*.

Tečkovou notaci lze v rámci ServiceNow REST API využít pro filtrační podmínky parametru sysparm_query a také pro parametr sysparm_fields. Díky této vlastnosti je možné data lépe filtrovat a vracet doplňující data z referencovaných tabulek v rámci jednoho dotazu. Lze tedy omezit množství dotazů a využití výpočetního výkonu klientské aplikace při zpracování dat.

5.8 Skriptované REST API

Skriptované REST API poskytuje vývojářům definovat vlastní webovou službu. Hlavním cílem tohoto řešení je možnost vytvářet vlastní koncové body s parametry a hlavičkami REST API a také implementovat skripty, obstarávající jednotlivé požadavky a odpovědi. Obecně tato funkcionalita dodržuje REST architekturu, nicméně je možné upravit ji pro využití jiných konvencí. (ServiceNow 2020q)

5.8.1 URL

URL skriptovaného REST API má následující tvar:

`https://<instance.service-now.com>/api/<namespace>/<ver>/<api_id>/<res>`

Některé části URL jsou povinné, další jsou volitelné:

- <**instance.service-now.com**>: URL ServiceNow instance ke které je pomocí REST API přistupováno
- <**namespace**>: Pro API s globálním rozsahem je jméno tohoto rozsahu nastaveno pomocí systémové property *glide.appcreator.company.code*. API s aplikačním rozsahem je využíváno systémové jméno tohoto rozsahu.
- <**ver**>: Jedná se o nepovinný parametr, určuje verzi API ke které je přistupováno. Pokud parametr není uveden, je dotaz přesměrován k výchozí verzi.
- <**api_id**>: Identifikátor REST API, ke kterému je přistupováno. Výchozí hodnota je odvozena od názvu API, je však možné nadefinovat vlastní.
- <**res**>: Relativní cesta ke zdroji REST API.

5.8.2 Zdroje

Zdroje pro skriptované REST API jsou ekvivalentní se zdroji klasického REST API. Vývojářům je zde umožněno definovat pro jaké HTTP metody jsou zdroje určeny a jak budou požadavky na ně zpracovány. Dále je samozřejmě možné definovat pro jedno API více těchto zdrojů.

Pro zdroje je dále možné definovat, jaké parametry je možné při volání předávat. Při tvorbě zdroje skriptovaného REST API lze specifikovat, jaké parametry jsou při volání k dispozici a také určit, zda jsou povinné nebo volitelné. Takto definované parametry jsou následně při zpracování požadavku dostupné v poli `queryParams` objektu `request`.

5.9 Update Set

V rámci Update setu jsou sledovány změny v aplikačních konfiguračních souborech a systémových vlastnostech. V rámci platformy je tímto umožněno vyvíjet novou funkcionalitu na sub-produkční instanci a do produkční instance přenášet až konečné změny. Ke každému Update setu jsou přiřazeny dílčí záznamy o aktualizacích. Tyto jsou následně přenášeny s celým Update setem. (ServiceNow 2020u)

5.9.1 Stavy

Tento verzovací systém dále umožňuje porovnávat jednotlivé verze konfiguračních souborů a tyto změny dále slučovat, případně vracet k požadované verzi.

Update set jako takový může být v jednom ze 3 stavů:

- In progress
- Complete
- Ignore

Pro vývoj v rámci ServiceNow je velmi podstatný stav **In progress**. v tomto stavu je Update set aktivní a je možné do něj zapisovat provedené změny. Každý rozpracovaný Update set by měl v tomto stavu setrvávat po celou dobu vývoje, dokud není připraven na přesun do další instance.

Jakmile je vývoj, příp. jeho část, dokončen, Update set je přepnut do stavu **Complete**. tento stav signalizuje, že jsou změny připravené k přesunu na jinou instanci. Update set v tomto stavu je dobré nikdy nepřepínat zpět do stavu *In progress*. V případě nutnosti opravy či úpravy funkcionality je doporučeno založit nový Update set a změny provést

v rámci něj. *Complete* Update set je dále možné exportovat do XML, což může být výhodné pro zálohování či přenos na nepropojenou instanci, případně celou implementovanou funkcionalitu „zahodit“ funkcí *Back out*.

Posledním stavem je stav **Ignore**. Update set ve stavu *Ignore* ve většině případů obsahuje nepotřebnou funkcionalitu, případně dokončený vývoj, který nechceme přenášet na další instance. Změny v rámci tohoto Update setu jsou stále provedeny na lokální instanci, do Update setu již nelze přidávat další, zároveň jej však není možné exportovat nebo přenést na jinou instanci.

5.9.2 Přenos mezi instancemi

Update set ve stavu *Complete* lze přenést na jinou instanci. Administrátor má na výběr 2 možnosti přenosu:

- vzdálený přenos ze zdrojové instance
- přenos pomocí XML souboru

Vzdálený přenos je aplikovatelný na instance, které jsou k tomuto nakonfigurovány. Zdrojová instance je zde nastavena jako jeden ze zdrojů aktualizací, nastavení dále vyžaduje přihlašovací údaje do této instance. Následně je možné všechny připravené Update Sety získat jedním kliknutím. (ServiceNow 2020w)

Alternativou ke vzdálenému přenosu je **přenos pomocí XML souboru**. Připravený Update set je na výchozí instanci exportován a následně je importován na cílové instanci.

Pro přenos implementovaných změn jsou pro nás důležité 2 moduly, jmenovitě *Local Update sets* (volně přeloženo jako Lokální sady aktualizací) a *Retrieved Update sets* (volně přeloženo jako Načtené sady aktualizací). *Local Update sets* administrátora zajímá primárně na zdrojové instanci. Update sety ve stavu *Complete* je možné přenést na jinou instanci. Přenesené Update sety jsou dostupné na cílové instanci v *Retrieved Update sets* modulu.

Update sety v modulu *Retrieved Update sets* jsou v prvotní fázi pouze načtené ze zdrojové instance. Dalším krokem je **Preview**, tedy jakýsi náhled, který otestuje validnost a aktuálnost změn a odhalí případné konflikty. Po vyřešení případných konfliktů následuje konečný **Commit**, tedy potvrzení. Tímto jsou změny z přeneseného Update setu aplikovány na cílové instanci a Update set je možné najít v modulu *Local Update sets* této instance. (ServiceNow 2020o)

Kapitola 6

Analýza a návrh

Součástí implementace práce je komplexní integrace řešení ITSM procesu a umožnění snadné tvorby dalších integračních aplikací, případně usnadnit práci s daty ITSM procesu v rámci aplikací třetích stran.

Implementovanou funkcionalitu lze rozdělit na dvě základní části:

- REST API
- Integrace

Vzhledem k předpokládanému využití implementovaného API v rámci integrace je nutné věnovat návrhu tohoto API zvýšenou pozornost. Primárním cílem je vytvořit API, které umožní rychlou a jednoduchou práci s požadovanými daty.

6.1 REST API

Vzhledem k nativnímu API pro práci s daty v rámci platformy by bylo možné využít pro integraci i toto řešení, tedy integraci pomocí *Table API*. *Table API* umožňuje přistupovat k datům ve všech tabulkách v rámci platformy, tyto data číst a upravovat. Pro jednoduché řešení přístupu však není příliš vhodné vzhledem k velkému rozsahu dostupných tabulek, tedy i příliš velké volnosti uživatele při přístupu k těmto datům. Dále je nutné znát i strukturu databáze a relací mezi definovanými tabulkami.

6.1.1 Požadavky

Hlavním požadavkem pro implementované REST API je jednoduchá práce s požadovanými daty. V rámci této implementace jde převážně o data uložená v tabulkách pro podporu ITSM procesů a data k nim navazující. Na rozdíl od *Table API* je tedy vyžadováno

sjednocení dat z více tabulek na jednom přístupovém bodu. Tento přístupový bod musí umožňovat přístup ke všem potřebným datům bez nutnosti znalosti struktury databáze a relací v této struktuře. Vyhledávání je v tomto případě značně zjednodušeno a převedeno na úroveň API, čímž je koncový uživatel odstíněn od nutnosti znát technické podrobnosti implementace a platformy samotné. Tímto je také snížen počet potřebných parametrů pro vyhledávání a zjednodušena práce s vytvářením REST API dotazů.

Využitím vlastního API lze předejít nevyžádaným dotazům na jiné tabulky a tím zamezit čtení a změně dat neoprávněnými stranami. Dále je také možné vytvořit restrikce pro dostupná data a tím omezit, s jakými daty je možné v rámci jedné tabulky pracovat.

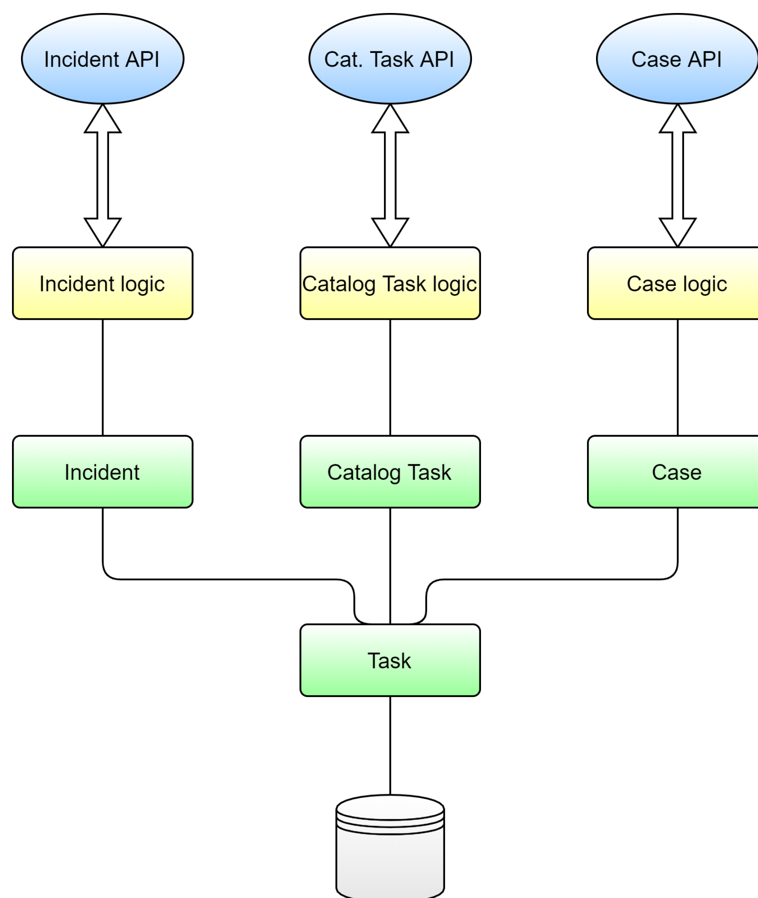
Pro snadnou tvorbu dalších, navazujících, aplikací je také vhodné vytvořit API pro přístup k referencovaným datům a reimplementovat restrikce, které jsou běžně dostupné pouze v rámci platformy. V takovém případě jde převážně vytvoření funkcionality umožňující získávat pro referenční pole správné vstupy z referencovaných tabulek a to převážně za použití filtrů, které jsou pro tato pole definovány, nicméně nejsou standardně přístupné v rámci nativního API.

6.1.2 Možnosti řešení

Pro implementaci je možné vybrat několik možných typů architektury API. V rámci analýzy je nutné určit nejvhodnější model, který bude splňovat definované požadavky, bude lehce udržovatelný, rozšiřitelný a konfigurovatelný.

Samostatné API

Architektura samostatných API pro každou z tabulek umožňuje implementovat metody pro práci s daty každé z těchto tabulek samostatně. Tato architektura poskytuje možnost specifické implementace API pro každou z tabulek, přesně dle jejích specifikací.



Obrázek 6.1: Struktura samostatných API *(Vlastní zpracování)*

Mezi **výhody** tohoto typu API je možné zařadit velkou míru upravitelnosti implementace a jednoduchosti řešení. Není zde nutné řešit univerzálnost řešení a je tedy mnohem jednodušší pro implementaci. Dále je možné při implementaci zohlednit všechna specifika dané tabulky.

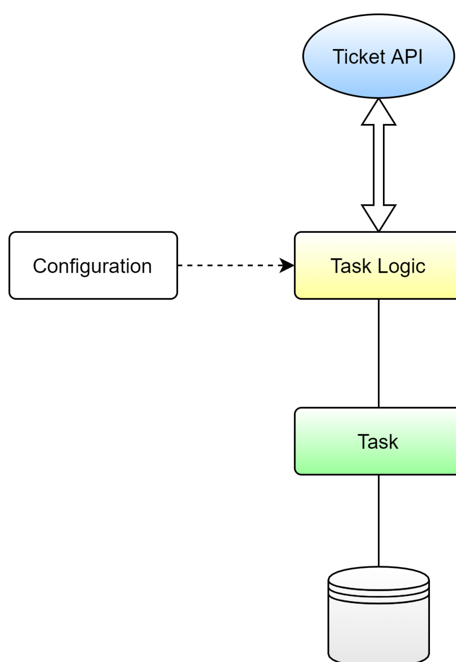
Implementace této architektury má i **nevýhody**. Mezi tyto patří například opakování kódu při práci s daty, která jsou v rámci struktury databáze pro tabulky společná. Takto opakované části kódu jsou pro implementaci a následné změny či opravy nevýhodné a mohou vést k nekonzistenci dat.

V případě potřeby API pro další z tabulek je nutné vytvořit samostatné API a celou řídící logiku. Tímto však narůstá množství kódu v rámci celé implementace a počet opakování společných částí kódu. S přibývajícími přístupovými body tak narůstá celková složitost celé struktury API a opravy či změny jsou stále obtížněji realizovatelné.

Architektura samostatných API tedy bude využitelná primárně pro tabulky, které spolu příliš nesouvisí, tedy například nedědí ze společné tabulky a je tedy nutné pro přístup k datům definovat individuální metody. Pro tabulky s jednotnou strukturou je vhodné využít jiný typ přístupu.

Jednotné API

Jednotné API zjednodušuje logickou strukturu a nahrazuje samostatná rozhraní jednotnou logikou, implementovanou primárně nad core tabulkou *Task*. K datům je přistupováno jednotně, specifika jednotlivých tabulek mohou být určena pomocí konfiguračních záznamů.



Obrázek 6.2: Struktura jednotného API *(Vlastní zpracování)*

Výhodou tohoto řešení je nulové opakování kódu pro logiku přístupu k datům. Jednotný přístup realizovaný nad tabulkou *Task* umožňuje jednoduchý přístup k hodnotám polí definovaných na core tabulce. Pole definovaná pouze pro tabulky dědící z *Task* tabulky však vyžadují individuální přístup, což může být **nevýhodou**, jelikož je nutná dodatečná konfigurace, která může být poměrně složitá.

Pro koncového uživatele může být použití takovéhoto API komplikované. Z hlediska přístupu se může zdát souhrnný přístupový bod výhodný, pro správné určení typu záznamu, ke kterému je přistupováno, je však nutné jako jeden z parametrů požadavku na takovýto bod odeslat i identifikátor. Tento identifikátor umožňuje určit typ požadovaného záznamu a pole, která je možné na takovýto požadavek vrátit.

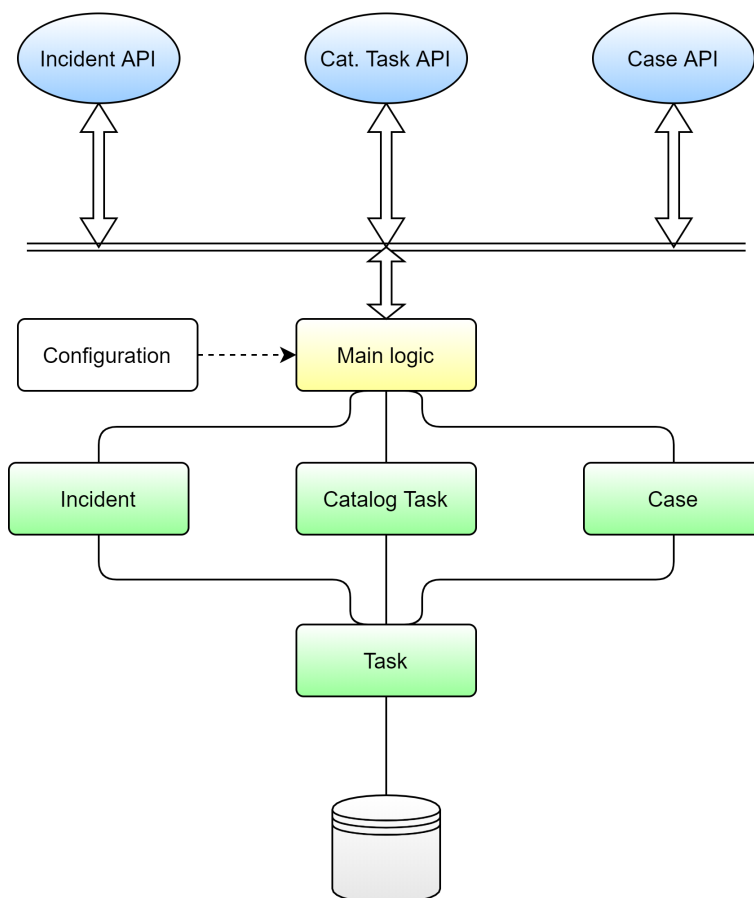
Je-li v rámci požadavku na toto API použit nesprávný identifikátor, může volání skončit chybou, případně dojít k nekonzistenci dat. Uživatel tedy musí být velmi obezřetný při sestavování parametrů požadavku.

Do procesu zde vstupuje i konfigurace, která určuje, jaká pole jsou přístupná v rámci API pro daného potomka *Task* tabulky. V této struktuře je konfigurace omezena pouze

na potomky *Task* tabulky, navíc je nutné přistupovat k polím, která nejsou na rodičovské tabulce pomocí speciální struktury. Pro pole *Incident* tabulky je to například použitím identifikátoru `ref_incident` před jménem pole v rámci tečkové notace.

Hybridní API

Architektura hybridního API kombinuje architekturu a procesy samostatných API a jednotného API. Pro jednotlivé tabulky jsou zde implementovány vlastní přístupové body, logika pro práci s daty je však společná. I zde vstupuje do procesu konfigurace, nicméně v tomto případě je jednodušší a přehlednější.



Obrázek 6.3: Struktura hybridního API *(Vlastní zpracování)*

Samotná implementace hybridního API může být složitější, nicméně pro koncového uživatele je tento přístup výhodnější. **Výhodou** tohoto řešení je jasné rozdělení přístupových bodů, odpadá tak nutnost v rámci požadavku používat identifikátor tabulky. **Nevýhodou** může být vyšší složitost implementačního řešení.

Jádrem tohoto řešení je hlavní, univerzální, logika pro přístup k datům. Přístup k jednotlivým tabulkám je zde řízen pomocí konfiguračních záznamů. Tyto záznamy jsou

unikátní pro každé z implementovaných API. Univerzálnost řešení umožňuje omezit opakování kódu pro různá API a řízení přístupu pouze pomocí konfigurace.

Pro implementaci API souvisejících s danou rodičovskou tabulkou je tedy nutné vytvořit pouze přístupový bod a konfigurační záznam. Není nutné zasahovat do kódu logiky pro přístup k datům.

Tato architektura je poměrně lehce rozšiřitelná a přes počáteční složitost implementace i jednoduše rozšiřitelná. Jedná se o výhodné řešení pro udržitelnou implementaci API pro přístup k potřebným datům. Zároveň se jedná o kompromisní řešení s ohledem na složitost implementace a objem implementace samotné.

6.1.3 Návrh architektury

Pro vytvoření implementace API, vhodné mimo jiné i k implementaci integrace ITSM procesů, byla zvolena kombinace **hybridního** a **samostatného** API. Hlavním důvodem je univerzálnost takového řešení a zároveň možnost implementace specifických funkcí pro některé z tabulek.

Obecný přístup

REST API budou implementovaná pro tabulky *Catalog Task* (sc_task), *Incident* (incident) a *Case* (sn_customerservice_case). Tyto tabulky dědí z tabulky *Task* (task), mají tedy základní pole společná (viz tabulka 6.1) a zároveň se jedná tabulky využitelné pro řízení ITSM procesů.

Pro společná pole je implementace hlavní logiky poměrně jednoduchá, k datům lze přistupovat homogenně a konfigurační záznam by v tomto případě mohl být společný pro všechny tabulky, jelikož se pole nacházejí ve všech výše zmíněných tabulkách. V rámci návrhu je však nutné zvážit i možnost zahrnutí těchto polí do samostatných konfigurací a to z důvodu využití těchto polí v jednotlivých tabulkách, případně k omezení přístupu k těmto polím pro jednotlivé tabulky.

Pro tabulku *Task* i další tabulky z ní dědící je definováno více polí, ty však nejsou pro vyvíjené API podstatná a nejsou tedy v tabulce 6.1 zahrnuta. Pro přístup k hodnotám těchto polí je, v případě potřeby, stále možné využívat generické *Table API*.

	Task	Incident	Cat. Task	Case
Active	✓	✓	✓	✓
Additional comments	✓	✓	✓	✓
Assigned to	✓	✓	✓	✓
Assignment group	✓	✓	✓	✓
Close notes	✓	✓	✓	✓
Closed	✓	✓	✓	✓
Closed by	✓	✓	✓	✓
Company	✓	✓	✓	✓
Contact type	✓	✓	✓	✓
Correlation display	✓	✓	✓	✓
Correlation ID	✓	✓	✓	✓
Created	✓	✓	✓	✓
Created By	✓	✓	✓	✓
Description	✓	✓	✓	✓
Impact	✓	✓	✓	✓
Number	✓	✓	✓	✓
Opened	✓	✓	✓	✓
Opened By	✓	✓	✓	✓
Parent	✓	✓	✓	✓
Priority	✓	✓	✓	✓
State	✓	✓	✓	✓
Sys ID	✓	✓	✓	✓
Short description	✓	✓	✓	✓
Task type	✓	✓	✓	✓
Updated	✓	✓	✓	✓
Updated by	✓	✓	✓	✓
Urgency	✓	✓	✓	✓
Work notes	✓	✓	✓	✓

Tabulka 6.1: Společná pole tabulek dědících z tabulky Task

Další pole jsou přítomna pouze na některých tabulkách (viz. tabulka 6.2), je tedy nutné je zahrnout do konfigurace jednotlivých API. I pro tato pole je posléze možné řešit v určitých případech poměrně homogenní přístup, nicméně je nutné brát v potaz, že tato pole jsou definována na každé z tabulek zvlášť a mohou tedy být rozdílného datového typu.

	Task	Incident	Cat. Task	Case
Account				✓
Caller		✓		
Catalog			✓	
Category		✓		✓
Contact				✓
On hold reason		✓		
Resolved		✓		✓
Resolved by		✓		✓
Resolution code		✓		✓
Request			✓	
Requested Item			✓	
Requested For			✓	
Subcategory		✓		✓

Tabulka 6.2: Pole definovaná na potomcích tabulky Task

Pro plný přístup ke všem datům ITSM záznamů je nutné implementovat také přístup k dalším tabulkám, obsahujícím přidružené záznamy a také ke konfiguraci tabulek samotných. Na rozdíl od *Table API* tedy není primárním cílem pouze poskytovat přístup k datům, ale poskytnout kompletní přístup k záznamům i se všemi vazbami a restrikcemi, které jsou běžně k dispozici v rámci webového rozhraní. Pro tuto funkcionalitu bude nutné přistupovat k definicím polí tabulky. Podle typu pole je možné určit, jakých hodnot může pole nabývat, což je zvláště důležité pro pole typu *seznam* nebo *reference*.

Pro návrh API je tedy nutné brát v potaz fakt, že nebude sloužit pouze k získávání a úpravě dat dané tabulky, ale i dat z referencovaných tabulek případně seznamů definovaných pro daná pole tabulky. Primárním účelem tohoto přístupu není změna dat v referencovaných tabulkách, ale využití těchto hodnot v pro přiřazení do polí záznamů. Cílem je tedy umožnit uživateli získat pouze relevantní data pro přiřazení do jednotlivých

polí a omezit tak chyby při ukládání dat, které mohou nastat z důvodů omezení, která mohou být na těchto polích definované.

Výše zmíněná pole, tj. pole typu *seznam* nebo *reference*, jsou běžně zdrojem chyb a konfliktů při ukládání dat, neboť většina implementací nepočítá s možností získání pouze relevantních hodnot pro ukládání. Velmi často jsou vráceny pouze hodnoty z referencovaných tabulek, bez patřičných restrikcí. Druhým případem jsou poté implementace, které definují stejnou logiku jako jsou omezení definovaná na polích tabulky, nicméně tato implementace je často řešena pouze kopií logiky, což v případě změny definice restrikce na daném poli může vést k nekonzistencím, pokud není tato změna následně provedena i v dané implementaci API. Pro přístup k těmto datům je tedy výhodné přistupovat k definicím těchto restrikcí přímo v definici tabulky a tyto restrikce následně použít při vyhledávání a následném vrácení požadovaných dat pro pole tabulky.

Rozdělení funkcionality

Celková struktura dat a logiky lze rozdělit do několika vrstev, pro tuto implementaci byla vybrána následující struktura:

- REST API (přístupové body)
- Přístupová a konfigurační logika
- Databázová logika

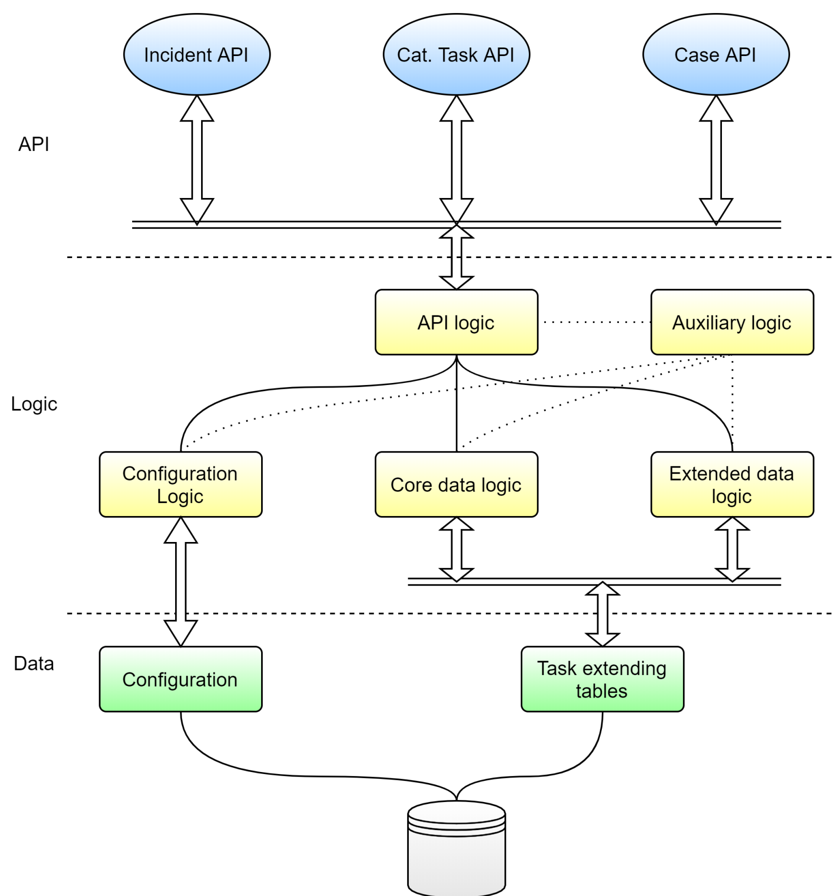
Databázová logika je jedinou z vrstev, kterou není nutné implementovat. Pro přístup k databázi je využito vestavěné třídy *GlideRecord*, případně *GlideRecordSecure*.

Přístupové body je nutné definovat pro každou z výše uvedených tabulek. Pro základní implementaci je v tomto případě nutné definovat tři přístupové body (jeden pro každou z ITSM tabulek) a jejich metody. Velká část těchto metod je společná pro všechna tři rozhraní, nicméně je také možné definovat specifické metody pro každý z přístupových bodů. Důvodem tohoto přístupu je zjednodušení přístupu ze strany uživatele a také možnost lehce rozdílných implementovaných metod.

Přístupová a konfigurační logika je rozdělena do několika samostatných Script Include záznamů, které reprezentují jednotlivé třídy implementované funkcionality. Součástí návrhu jsou třídy pro přístup k datům, přesněji pro přístup ke společným datům a doplňující, či rozšiřující třídy pro přístup ke specifickým datům. Tyto přístupy jsou zamýšleny jak pro čtení, tak i ukládání dat a také pro základní ošetření dat. Další částí

logiky jsou třídy pro práci s konfigurací, přesněji pro ukládání a načítání dat a další práci s konfiguračními záznamy.

Struktura implementace je naznačena na obrázku 6.4. Tento obrázek zachycuje návrh rozdělení logiky do vrstev a jednotlivých tříd. Každopádně, implementovaná logika se od tohoto schématu může drobně odchylovat, případně může obsahovat další třídy, které jsou implementovány jako pomocné a poskytují dodatečnou funkcionalitu.



Obrázek 6.4: Struktura implementace *(Vlastní zpracování)*

Jako první je z API vrstvy přistupováno k API logice. Tato část logiky je určena k poskytnutí funkcionality pro jednotlivé metody přístupových bodů. V případě potřeby je možné vytvořit hlavní třídu, která je následně použita jako prototyp pro rozšiřující třídy. Díky této struktuře je možné implementovat logiku se specifickými metodami pro každý z přístupových bodů a zároveň minimalizovat vytváření duplicitních částí kódu. Tato logika aktivně využívá dalších implementovaných tříd.

Pro práci s daty jsou navrženy dvě základní třídy a to tzv. *Core data logic*, tedy hlavní datová logika a *Extended data logic*, neboli rozšířená datová logika.

Hlavní datová logika je určena k poskytnutí přístupu ke společným polím tabulek, tedy primárně k polím *Task* tabulky, kterou ostatní tabulky rozšiřují. Tato logika je tedy jednotná pro všechny ITSM tabulky, pro které je aby vytvářeno. Společný přístup k datům je zde možné uplatnit vzhledem k jednotné definici polí a jejich datových typů.

Rozšířenou datovou logiku je zde možné uvažovat i společnou, případně využít obdobného modelu jako u API logiky, tedy vytvoření hlavního třídy a následné využití této třídy jako prototypu pro specifickou implementaci. Tento model může být nutné využít z důvodu možných rozdílů v definici polí tabulky, předně v rozdílu datových typů na polích rozšiřujících tabulek. Tyto rozdíly mohou být znatelné primárně v případě, že je nutné data nějakým způsobem upravovat před jejich uložením, případně jejich předáním na přístupový bod. Přestože databáze, tedy primárně třída *GlideRecord* vrací hodnoty polí jako textové řetězce, je možné počítat s následnou konverzí na jiný, adekvátní datový typ, který může v případě rozdílných definicí působit kolize.

Konfigurační logika je určena ke zpracování, tedy načítání a ukládání, konfiguračních dat. Tato data lze ukládat a zpracovávat různými způsoby s ohledem na jejich množství a složitost.

Vzhledem k poměrně malému objemu konfiguračních dat není vhodnou volbou uchovávat konfiguraci v samostatné tabulce. Takováto tabulka by v konečném důsledku vyžadovala větší množství konfiguračních záznamů než obsah tabulky samotný, navíc by každá dodatečná úprava konfigurace mohla vyžadovat další rozšiřování tabulky. Pro uchování konfigurace je tedy zvoleno ukládání do záznamů v tabulce `sys_properties`. Tyto záznamy umožňují uchovat konfiguraci ve vhodné datové struktuře, v tomto případě využitím vhodného JSON objektu.

Třída zpracovávající konfiguraci tedy musí implementovat i základní metody pro práci s JSON objekty, tedy jejich čtení, transformaci na instance objektů a také zpětnou transformaci objektu na JSON řetězec. Pro snadnou konfiguraci API lze uvažovat i o vytvoření formuláře pro zadávání konfiguračních dat, nicméně tato funkcionalita není primárním cílem implementace. Jelikož by konfigurace API neměla být měněna příliš často lze očekávat, že takovýto formulář sice zpříjemní práci s konfigurací, zároveň však není stěžejním prvkem implementace.

Součástí implementace je také pomocná logika. Tato logika je určena k poskytnutí podpůrných funkcí, které není vhodné umístit přímo do implementace jiných částí logiky, případně je implementovaná funkcionalita využívána napříč dalšími třídami logiky. Pomocná logika může být rozdělena do několika samostatných tříd, podle účelu jejího využití.

6.1.4 Implementace

Základním krokem implementace je vytvoření aplikace v rámci platformy ServiceNow. V tomto případě se jedná o aplikaci **ITSM API**. V rámci vytvořené aplikace je současně vytvořen i aplikační rozsah (scope) a to `x_273093_itsm_api`. Tento aplikační rozsah bude sloužit k implementaci většiny logiky aplikace.

Implementaci logiky API je nutné uchovat pro verzování a přenos mezi instancemi. K tomuto účelu byl založen Update Set *ITSM API Core*. Zde jsou uchovány konfigurační záznamy a změny hlavní logiky aplikace.

Konfigurační logika

Konfigurační logika je zastoupena třídou `ITSM_API_CoreConfiguration` uloženou ve stejnojmenném Script Include záznamu. Tato třída zpracovává konfigurační záznamy uložené v *sys_properties* tabulce, v záznamech určených speciálně pro konfiguraci přístupu k datům ITSM tabulek. Tyto záznamy je nutné definovat pro každé z implementovaných API, tedy pro každou z tabulek. Prefixem názvu záznamu je `x_273093_itsm_api.itsm_api.config`. Za tímto prefixem následuje název tabulky, ke které konfigurační záznam náleží.

```
1  {
2    "table": "task",
3    "fieldsRead": [
4      "sys_id",
5      "number",
6      ...
7      "state"
8    ],
9    "fieldsWrite": [
10     "number",
11     "active",
12     ...
13     "state"
14   ]
15 }
```

Ukázka kódu 6.1: Struktura konfiguračního záznamu *(Vlastní zpracování)*

Ukázka 6.1 zachycuje strukturu konfiguračního záznamu ve formátu JSON. Konfigurační záznam musí obsahovat název tabulky, pole systémových názvů polí pro čtení a pole názvů polí pro zápis. Tyto seznamy umožňují v rámci konfigurace definovat jistá omezení přístupu na úrovni API. Pro správnou funkci API je vhodné uvažovat v rámci konfigurace i omezení na úrovni databáze a případně omezená pole následně nezahrnovat do konfiguračního záznamu, čímž je možné předejít konfliktům při používání API.

Inicializace konfigurační třídy vyžaduje jako vstupní parametr název tabulky, nepovinným parametrem je pak seznam polí vyžadovaných uživatelem ke čtení. Během inicializace je do nově inicializovaného objektu uložen název tabulky, uživatelem vyžadovaný seznam polí a prefix konfiguračního záznamu. Dále je do objektu načtena konfigurace ve formě textového řetězce a následně také jako objektu. Z tohoto objektu jsou poté načteny seznamy polí pro zápis a pro čtení, přičemž seznam polí pro čtení je dán průnikem seznamu polí ke čtení načteným z konfigurace a uživatelem vyžadovaných polí. Pro relevantní výsledky při načítání záznamů jsou k polím ke čtení vždy přidána také pole `sys_id` a `number`, sloužící k identifikaci záznamu. Ověření vstupního parametru `tableName` a posloupnost volaných funkcí v rámci inicializace je zachyceno v ukázce 6.2.

```
1  initialize: function (tableName, userFields) {
2    if (tableName == null || tableName == '') {
3        throw new ReferenceError('Empty table name');
4      }
5      this.PROPERTY_PREFIX = 'x_273093_itsm_api.itsm_api.config.';
6      this._tableName = tableName;
7      this._userFields = userFields;
8      this._loadConfiguration();
9      this._prepareFieldsRead();
10     this._prepareFieldsWrite();
11 }
```

Ukázka kódu 6.2: Inicializační funkce konfigurační třídy *(Vlastní zpracování)*

Objekt konfigurační třídy je následně využíván v objektech tříd přístupové logiky, kde je nejčastěji využíváno přístupu k seznamu polí ke čtení a zápisu pro danou tabulku.

Přístupová logika

Pro přístupovou logiku je implementováno několik tříd. Hlavní třídou je přístupové logiky je třída `ITSM_API_CoreDataLogic`, definovaná ve stejnojmenném Script include záznamu. Tato třída slouží k přístupu k hlavní tabulce v ITSM struktuře, tedy k tabulce *Task*. Vzhledem k tomu, že jsou další ITSM tabulky „rozšířením“ *Task* tabulky je tato logika použitelná i pro tyto tabulky, nicméně nelze očekávat plnou funkcionalitu. Pro rozšiřující tabulky je dále implementována rozšířená logika, která z této hlavní logiky přímo vychází.

```
1 initialize: function (tableName, userFields) {
2   if (tableName == null || tableName === '') {
3       this._tableName = 'task';
4   } else {
5       this._tableName = tableName;
6   }
7   this._loadConfiguration(userFields);
8 }
```

Ukázka kódu 6.3: Inicializační funkce hlavní přístupové třídy *(Vlastní zpracování)*

Jak již bylo zmíněno, třída `ITSM_API_CoreDataLogic` je určena pro přístup k *Task* tabulce. Tato skutečnost je zachycena i v samotné inicializační funkci této třídy. Je-li objekt inicializován bez parametru `tableName`, je výchozí hodnotou název *Task* tabulky. Současně je při inicializaci objektu načtena i konfigurace pro vybranou tabulku. Při inicializaci objektu za použití obou parametrů je k inicializaci objektu konfigurace mimo jiné předán i seznam polí, která jsou uživatelem vyžadována ke čtení.

Pro práci se záznamy jsou definovány základní funkce pro čtení, vložení a úpravu záznamů a dále funkce pro přidávání tzv. *Work notes* a *Additional comments*, které slouží ke komunikaci v rámci záznamu a vyžadují mírně odlišný přístup než zbývající pole.

Pro čtení dat záznamu slouží funkce `getTicket()` zachycená v 6.4. Tato funkce vyžaduje identifikátor záznamu pro vyhledání záznamu. Funkce následně záznam vyhledá a do připraveného objektu načte hodnoty ze záznamu, dle polí, která jsou k dispozici ke čtení. Pro každé z polí jsou předány dvě hodnoty a to systémová a zobrazovací hodnota. Systémová hodnota je vhodná pro reprezentaci v systému, případně databázi. Zobrazovací hodnota je hodnota určená k zobrazení uživateli a je často přeložena do jazyka uživatele. Jedná se tedy o jakousi uživatelsky přívětivou a dobře čitelnou textovou reprezentaci uložené hodnoty.

```
getTicket: function (sys_id) {
  var ticket = {};
  var fields = this._configuration.getFieldsRead();
  if (gs.nil(sys_id) || gs.nil(fields)) {
      return ticket;
  }
  var taskGR = new GlideRecord(this._tableName);
  if (taskGR.get(sys_id)) {
      fields.forEach(function (field) {
      ticket[field] = {
        'value': taskGR.getValue(field),
        'display_value': taskGR.getDisplayValue(field)
      };
    });
  }
    return ticket;
}
```

Ukázka kódu 6.4: Funkce getTicket hlavní přístupové třídy *(Vlastní zpracování)*

Vytvoření nového záznamu je implementováno funkcí `insertTicket()`. Tato funkce je volána s jedním parametrem, jímž je objekt obsahující data k zápisu. Jednotlivé atributy tohoto objektu odpovídají polím nového záznamu. Před samotným vytvořením záznamu je ověřeno, zda je možné do všech polí, která byla v rámci objektu předána, možno zapisovat.

Je-li nalezeno pole, ke kterému není možné tento přístup povolit, je jeho hodnota ignorována. Po filtraci dat jsou následně údaje uloženy do nově inicializovaného záznamu nad danou tabulkou a po uložení všech dat je provedeno vložení do databáze.

Při přístupu k žurnálovým polím jako `work_notes` a `comments` je nutné upravit způsob přístupu k ukládání hodnoty pole. V tomto případě je nutné využit přímého přiřazení pro vložení hodnoty, jelikož využitím funkce `setValue()` nelze do těchto polí zapisovat. V rámci globálního rozsahu by bylo možné využít i funkce `setJournalEntry()`, ta však není dostupná v aplikačním rozsahu implementovaného řešení.

Obdobnou funkcionalitu poskytuje i funkce `updateTicket()`. Na rozdíl od předešlé funkce, vyžaduje dva parametry. Prvním parametrem je identifikátor záznamu, druhým jsou data k zápisu. Implementačně se logika liší především v získání objektu záznamu pro zápis. Vzhledem k tomu, že je funkce navržena pro úpravu záznamu, je nutné tento záznam nejprve najít v databázi a následně jeho data upravit. Samotná úprava dat je nutně podmíněna vyhledáním záznamu. Bez této podmínky by mohlo dojít k vytváření

duplicitních záznamů na danou tabulkou. Vyjímkou jsou opět žurnálová pole *Additional comments* a *Work notes*, která vyžadují při vkládání do záznamu individuální přístup.

```
insertTicket: function (ticketData) {
    var utils = new global.ArrayUtil();
    var ticket = {};
    var fields =
        utils.intersect(this._configuration.getFieldsWrite(),
        Object.keys(ticketData));

    if (!gs.nil(fields) && fields.length > 0) {
        var ticketGR = new GlideRecord(this._tableName);
        ticketGR.initialize();
        for (var i = 0; i < fields.length; i++) {
            var key = fields[i];
            if (this._journalFields.indexOf(key) === -1) {
                ticketGR.setValue(key, ticketData[key]);
            } else {
                ticketGR[key] = ticketData[key];
            }
        }
        if (ticketGR.insert()) {
            ticket = this.getTicket(ticketGR.getUniqueValue());
        }
    }
    return ticket;
}
```

Ukázka kódu 6.5: Funkce insertTicket hlavní přístupové třídy *(Vlastní zpracování)*

Výsledkem volání funkcí pro vložení a úpravu záznamu je, v případě úspěchu, nový záznam v databázi. Návratovou hodnotou je objekt záznamu. V případě neúspěchu při vkládání dat je vrácen prázdný objekt.

Pro přístup přímo k základním žurnálovým polím jsou implementovány samostatné funkce. K zápisu slouží funkce `addWorkNote()` a `addComment()`. Tyto funkce tedy slouží ke vkládání hodnot do těchto žurnálových polí a mohou být v rámci API využity pro přenos komunikace mezi instancemi nebo instancí a aplikací třetí strany právě pomocí komentářů nebo pracovních poznámek.

Ke čtení z těchto polí lze využít funkce `getWorkNotes()` a `getComments()`. Tyto slouží k čtení z žurnálových polí. Obě funkce přijímají dva parametry, první parametr je povinný,

jedná se o identifikátor záznamu. Druhý, nepovinný, parametr je index žurnálového záznamu, pro běžné použití jsou uváděny parametry -1 a 1. Hodnota indexu 1 je při volání funkce příznakem pro vrácení posledního hodnoty v žurnálovém poli, tedy nejnovějšího záznamu. Použitím parametru -1 dojde k vrácení všech hodnot žurnálového pole. Tyto hodnoty jsou vráceny formou textového řetězce. Hodnoty jsou seřazeny sestupně, podle data vložení a odděleny dvojitým odřádkováním, tedy znaky \n\n.

Pro jednodušší práci s poli, které mají předem definované hodnoty je implementována logika, která uživateli vrací data vhodná pro tato pole. Jedná se především o pole typu Reference a pro pole s definovaným seznamem hodnot. K tomuto účelu jsou definovány také funkce vracející seznamy těchto polí pro danou tabulku.

K získání všech polí s předdefinovaným seznamem hodnot pro tabulku je implementována funkce getChoiceFields(). Tato funkce přistupuje přímo k Dictionary záznamům tabulky, tedy přímo k definici jednotlivých polí. Pro každé z polí je ověřeno, zda jsou pro pole předdefinované hodnoty, případně by vzhledem k typu pole měly být definovány.

```
1  getChoiceFields: function () {
2    var choiceFields = {};
3    var dictionaryGR = new GlideRecord('sys_dictionary');
4    dictionaryGR.addQuery('name', this._tableName);
5    dictionaryGR.addQuery('choice', 'IN', '1,3');
6    dictionaryGR.addNotNullQuery('element');
7    dictionaryGR.addActiveQuery();
8    dictionaryGR.query();
9    while (dictionaryGR.next()) {
10     var element = dictionaryGR.getValue('element');
11     choiceFields[element] = {
12       name: element,
13       table: dictionaryGR.getValue('name'),
14       choiceNumber: dictionaryGR.getValue('choice'),
15       choiceName: dictionaryGR.getDisplayValue('choice'),
16       default_value: dictionaryGR.getDisplayValue('default_value')
17     };
18   }
19   return choiceFields;
20 }
```

Ukázka kódu 6.6: Funkce getChoiceFields hlavní přístupové třídy *(Vlastní zpracování)*

Návratovou hodnotou této funkce je objekt, jehož prvky jsou objekty jednotlivých polí. Objekty polí obsahují systémové jméno pole, systémové jméno tabulky, nad kterou je pole definováno, systémové a uživatelské označení typu pole a výchozí hodnota. Takto připravený objekt je možné předat uživateli, který následně může získat seznamy předdefinovaných hodnot pro jednotlivá pole.

```
1  _getChoicesForField: function (field) {
2    var choiceGR = new GlideRecord('sys_choice');
3    choiceGR.addQuery('name', field.table);
4    choiceGR.addQuery('element', field.name);
5    choiceGR.addQuery('inactive', false);
6    choiceGR.query();
7    var choices = {};
8    while (choiceGR.next()) {
9      var value = choiceGR.getValue('value');
10     var language = choiceGR.getValue('language');
11     if (!(value in choices)) {
12       choices[value] = {
13         value: value,
14         table: field.table,
15         label: {},
16         sequence: choiceGR.getValue('sequence'),
17       };
18     }
19     choices[value]['label'][language] = choiceGR.getValue('label');
20   }
21   return {
22     field: field.name,
23     choices: choices,
24     default_value: field.default_value
25   };
26 }
```

Ukázka kódu 6.7: Funkce _getChoicesForField hlavní přístupové třídy *(Vlastní zpracování)*

K získání seznamů hodnot pro pole je určena funkce `getChoicesForField()`. Tato funkce přijímá dva parametry. Prvním je jméno pole, pro které mají být hodnoty vyhledány, druhým indikátor, který umožňuje vynechat kontrolu, zda je pole v konfiguraci mezi poli pro zápis. Druhý parametr je v tomto případě nepovinný a ve výchozím stavu je kontrola na možnost zápisu prováděna. Dále je provedena kontrola, zda je vyžadované pole

mezi poli, pro které je seznam hodnot definován, tedy, zda je mezi poli vrácenými funkcí `getChoiceFields()`.

Po ověření pole je provedeno volání funkce `_getChoicesForField()`, která je zachycena v ukázce 6.7. Této funkci je nutné předat jeden parametr objekt definice pole ve formátu, který je vrácen funkcí `getChoiceFields()`. Funkce provede dotaz nad tabulkou `sys_choice` pro dané pole a tabulku. Vzhledem k povaze definice záznamů nad touto tabulkou, je ve většině případů pro jednu systémovou hodnotu a pole, nalezeno více záznamů, které se liší primárně hodnotou k zobrazení, která se liší v závislosti na jazykové mutaci. Toto je do jisté míry omezení, jelikož pro každou ze systémových hodnoty nemusí být hodnota k zobrazení definována ve všech jazycích.

Implementovaná funkcionalita počítá s jazykovou mutací záznamů při konstrukci objektu, který je navrácen. Návratovou hodnotou této funkce je objekt obsahující jméno pole, výchozí hodnotu a seznam objektů hodnot pro dané pole. Tyto objekty mají pevnou strukturu a obsahují systémovou hodnotu záznamu, tabulku, pro kterou je hodnota definována, pořadí záznamu pro zobrazení a seznam hodnot k zobrazení, rozdělený dle definovaných jazyků. Počet hodnot k zobrazení se může u jednotlivých hodnot lišit. Struktura objektu je zachycena v ukázce 6.8 ve formátu JSON.

```
1  {
2  "field": "priority",
3  "choices": {
4    "1": {
5      "value": "1",
6      "table": "task",
7      "label": {
8        "en": "1 - Critical",
9        "cs": "1 - kriticka"
10     },
11     "sequence": "1"
12   },
13   ...
14 },
15 "default_value": "4"
16 }
```

Ukázka kódu 6.8: Reprezentace hodnot pro pole nad tabulkou Task *(Vlastní zpracování)*

Využití těchto funkcí je výhodné při implementaci řešení pracující s poli tohoto typu. Vzhledem k chování platformy v případě vložení nepředdefinované hodnoty je využití těchto funkcí v případě implementace doporučeno. Díky hodnotám vráceným přímo z definice pole je možné pracovat s validními daty a zamezit inkonzistenci dat, která může nastat v případě, je-li do pole uložena hodnota, která není předdefinována. Díky tomuto řešení je možné pracovat s aktuálními hodnotami i po změně definice pole pro danou tabulku.

Obdobná funkcionalita je implementována i pro referenční pole, jejichž hodnoty se odkazují na záznamy v jiných tabulkách. Vzhledem k vazbě, která je uskutečněna pomocí identifikátoru `sys_id` v textové podobě, je nutné před uložením do záznamu tento identifikátor získat. Pro správné uložení záznamu je nutné, aby byl identifikátor validní, ale také aby náležel záznamu, který je uložen v tabulce, na kterou se pole odkazuje.

K získání všech referenčních polí tabulky je implementována funkce `getReferenceFields()`. Implementovaná logika vyhledá v tabulce `sys_dictionary` záznamy polí náležící příslušné tabulce a splňující podmínku pro typ pole `Reference`. Návratovou hodnotou funkce je objekt reprezentující seznam objektů referenčních polí. Každý z objektů reprezentující jedno referenční pole má pevnou strukturu. Objekt pole obsahuje systémové jméno pole a tabulky, pro kterou je pole definováno, jméno tabulky, ke které se reference váže, jméno pole z referencované tabulky, které je použito k získání hodnoty k zobrazení a dále pak typ a definice referenčních podmínek. Referenční podmínka, neboli v originále *Reference qualifier* může být trojího typu:

- Simple
- Dynamic
- Advanced

Vzhledem k relativně omezeným možnostem je pro další funkcionalitu implementována logika pouze pro referenční podmínky typu *Simple*. V objektu referenčního pole jsou i přes toto omezení vráceny definice podmínek pro všechny tři typy, pokud jsou definované. Objekt definice referenční pole je zachycen v ukázce 6.9.

Pro načtení hodnot z referencované tabulky je implementována funkce příjímající 3 parametry, `getReferenceValuesForField()`. Prvním, povinným parametrem, je jméno referenčního pole. Druhým, nepovinným parametrem, je vzor, tedy část hodnoty, pomocí které bude upřesněno vyhledávání a třetím, opět nepovinným parametrem, je příznak pro ověřování zapisovatelnosti daného pole. Jméno pole je ve výchozím stavu vyhledáno mezi poli, do kterých lze zapisovat. Je-li tato validace úspěšná, následuje ověření, že se skutečně

jedná o referenční pole. V případě, že se skutečně jedná o referenční pole, je dále definice tohoto pole společně se vzorem předána funkci `_getReferenceValuesForField()`.

```
{
"assigned_to": {
  "name": "assigned_to",
  "table": "task",
  "reference": "sys_user",
  "ref_qual_type": "simple",
  "ref_qual_cond": "roles=itil",
  "ref_qual": "roles=itil",
  "dyn_ref_qual": null,
  "display": "name"
  }
}
```

Ukázka kódu 6.9: Reprezentace objektu definice referenčního pole tabulky Task *(Vlastní zpracování)*

Funkce `_getReferenceValuesForField()` vyhledává záznamy v referencované tabulce podle definice referenčního pole. Vzhledem k omezením je referenční podmínka do vyhledávání přidána pouze v případě jednoduchého (*Simple*) typu podmínky. Dále je do podmínky přidán i vzor, je-li definován. Vzor může být použit dvěma způsoby, a to pro vyhledávání názvu, který vzorem začíná, případně názvu, který vzor obsahuje, pokud vzor začíná znakem `*`.

Návratovou hodnotou je objekt obsahující název pole a seznam objektů reprezentujících záznamy z referencované tabulky. Tyto objekty mají jednoduchou strukturu, obsahující identifikátor záznamu `sys_id` a zobrazovací hodnotu. Délka seznamu hodnot je omezena počtem, který je definován v záznamu `x_273093_itsm_api.core_data.reference_limit` uloženým v tabulce `sys_properties`.

Jak již bylo zmíněno, pro vyhledávání jsou využity pouze definice podmínek jednoduchého typu podmínky. Tento přístup byl zvolen primárně z důvodu proveditelnosti. Vzhledem k povaze podmínky jednoduchého typu je možné ji využít i bez přítomnosti specifického záznamu, jehož hodnoty jsou často využívány v pokročilejších podmínkách. Pokročilé podmínky dále ve velkém množství případů využívají skriptů pro vytvoření databázového dotazu, případně kombinaci skriptu a hodnot ze záznamu, což není v případě obecných dotazů možné využít.

Rozšířená přístupová logika

Rozšířená přístupová logika je implementována třídou `ITSM_API_ExtendedDataLogic`, uloženou ve stejnojmenném Script Include záznamu. Tato třída je rozšířením základní datové logiky, která je určená primárně k přístupu k *Task* tabulce. Třída rozšířené logiky je určena k využití jako předek dalších implementací, tedy bude rozšiřována dalšími třídami pro každou z ITSM tabulek.

S tímto přístupem se bohužel v rámci ServiceNow setkáváme spíše sporadicky. Každopádně je tato možnost platformou dobře podporována a lze ji nalézt i v některých implementacích, vytvořených právě společností ServiceNow. Výhodou tohoto řešení je jednoduchá reimplementace funkcí, které vyžadují úpravu, bez nutnosti implementace celého řešení od samého počátku. Tímto je možné dosáhnout velmi variabilní implementace, která však umožňuje zachovávat jednotnou logiku pro přístup k datům.

```
1  var ITSM_API_ExtendedDataLogic = Class.create();
2  ITSM_API_ExtendedDataLogic.prototype =
       Object.extendsObject(ITSM_API_CoreDataLogic, {
3    /**
4     * Init function for the object
5     * @param userFields
6     */
7    initialize: function (tableName, userFields) {
8      ITSM_API_CoreDataLogic.prototype.initialize.call(this,
           tableName, userFields);
9    },
10
11   /**
12    * Get table info (name, parent)
13    * @returns {{}}
14    * @private
15    */
16   _getTableInfo: function () {
17     return new ITSM_API_Utils().getTableInfo(this._tableName);
18   },
19   ...
20   type: 'ITSM_API_ExtendedDataLogic'
21 });
```

Ukázka kódu 6.10: Ukázka definice třídy rozšířené logiky ITSM_API_ExtendedDataLogic *(Vlastní zpracování)*

Základní definice třídy je zachycena v ukázce 6.10. Prototyp třídy je definován jako rozšíření původní třídy pomocí metody `extendObject` třídy `Object`. V ukázce je zachycena i reimplementace inicializační funkce, tedy volání inicializační funkce z nadřazené třídy.

Nově implementovanou funkcí je funkce `_getTableInfo()`, která je vytvořena pro získání základní informace o tabulce, přesněji názvu tabulky a rodičovské tabulky. Funkce vrací objekt, obsahující tyto informace, pomocí volání funkční logiky z pomocné třídy `ITSM_API_Utils`. Tento objekt je dále používán v nově reimplementovaných funkcích.

Třída rozšířené datové logiky reimplementuje některé funkce a to `getReferenceFields()`, `getChoiceFields()` a `_getChoicesForField()` a to z důvodu nutnosti vyhledávání hodnot nejen na aktuální tabulce, ale i na rodičovské tabulce. Pole zděděná z rodičovské tabulky jsou v rámci definice tabulky stále vázány na rodičovskou tabulku, je tedy nutné je vyhledat na obou tabulkách. Toto je využito i při hledání předdefinovaných hodnot polí, které, v případě že nejsou definované na dceřiné tabulce, je nutné vyhledat na tabulce rodičovské.

Tato úprava původní logiky je postačující k přístupu pro většinu ITSM tabulek, nicméně z důvodu další rozšiřitelnosti, je pro přístup ke každé z těchto tabulek vytvořena vlastní třída, rozšiřující právě třídu rozšířené přístupové logiky. Tyto třídy jsou následně využity v implementaci jednotlivých přístupových bodů.

Přístupové body

Pro přístup k ITSM záznamům jsou určeny následující přístupové body:

- Task API
- Incident API
- CTask API
- Case API

Task API je dostupné na adrese `/api/x_273093_itsm_api/task_api` a poskytuje přístup k datům uloženým v *Task* tabulce.

Incident API je dostupné na adrese `/api/x_273093_itsm_api/incident_api` je rozšířením *Task API*, přesněji, využívá rozšířené logiky pro přístup k datům a základní konfiguraci tabulky. Tento přístupový bod umožňuje přístup k datům *Incident* tabulky dle vlastní konfigurace.

CTask API, dostupné na adrese `/api/x_273093_itsm_api/ctask_api`, je opět rozšířením *Task API* a slouží k přístupu k záznamům tabulky *Catalog Task*.

Case API, dostupné na adrese `/api/x_273093_itsm_api/case_api`, opět rozšiřuje *Task API* a v tomto případě slouží k přístupu k záznamům tabulky *Case*.

Vzhledem ke společné základní logice jsou společné zdroje API dále popsány pouze pro *Task API*.

Ke čtení záznamů z tabulky je určen zdroj **GetTicket**. Tento zdroj je přístupný na kořenové adrese s adresním parametrem *keys* a dotazovým parametrem *fields*. V platformě je tato relativní adresa reprezentována jako `/{keys}`. Zdroj umožňuje pomocí metody `GET` přistoupit k záznamu, případě záznamům, definovaných v parametru *keys*, pomocí sys_id záznamů, jako identifikátoru pro vyhledávání. Dále je možné pomocí parametru *fields* omezit seznam získaných polí záznamu, nicméně vrácená pole jsou stále omezena konfigurací přístupové logiky. Logika tohoto zdroje je zachycena v ukázce 6.11.

```
1  (function process(request, response) {
2    var keys = request.pathParams.keys;
3    var fields = request.queryParams.fields;
4    var body = {
5      data: {}
6    };
7    if (!gs.nil(keys)) {
8      var utils = new ITSM_API_CoreDataLogic('task', fields);
9      var keysArr = keys.split(',');
10     keysArr.forEach(function(key) {
11       body.data[key] = utils.getTicket(key);
12     });
13   }
14   response.setBody(body);
15 })(request, response);
```

Ukázka kódu 6.11: Ukázka logiky zdroje GetTicket *(Vlastní zpracování)*

Pro vložení nového záznamu je určen zdroj **InsertTicket**. Tento je opět dostupný na kořenové adrese `/`, tentokrát bez adresního parametru a metodou k přístupu je metoda `POST`. Parametry pro uložení záznamu jsou předány v těle požadavku, přesněji v JSON objektu, který je v těle požadavku uložen. Objekt je následně předán funkci `insertTicket()` přístupové logiky a následně, v případě úspěšného uložení dat, je navrácen objekt nově uloženého záznamu ve formátu JSON.

K úpravám záznamů je vytvořen zdroj **UpdateTicket**. Zdroj je dostupný metodou `PUT`, opět na kořenové adrese, s adresním parametrem *key*. V systému je adresa reprezentována jako `/{key}`. Pro předání dat, která mají být nově uložena do záznamu, je opět využito těla požadavku, které obsahuje pole a jejich hodnoty ve formátu JSON.

```
{
  "impact":"2",
  "urgency":"1"
}
```

Ukázka kódu 6.12: Ukázka těla požadavku pro změnu záznamu *(Vlastní zpracování)*

Dále jsou implementovány zdroje **AddWorkNote** a **AddComment** pro přidávání komentářů a pracovních poznámek k jednotlivým záznamům ITSM tabulek. Systémové adresy těchto zdrojů jsou `/note/{key}`, respektive `/note/{key}`. Jak je tedy vidět, tyto zdroje vyžadují předání identifikátoru v adresním parametru *key*. Zpráva, určená k uložení do záznamu, je předána v těle `POST` požadavku, opět ve formátu JSON, kde zpráva samotná je předána jako hodnota proměnné `message`. Ukázka JSON řetězce pro přidání zprávy je zachycena v ukázce 6.13.

```
{
  "message":"New message to be added to ticket"
}
```

Ukázka kódu 6.13: Ukázka těla požadavku pro přidání zprávy *(Vlastní zpracování)*

Tento zdroj je určen pouze ke vkládání těchto zpráv, přesněji pro jednodušší vkládání než v případě využití zdroje *UpdateTicket* a to bez omezení konfigurační logikou, ověřující možnost zápisu do těchto polí.

Pro získání seznamu předdefinovaných hodnot vybraných polí je implementován zdroj **GetChoices**, který je interně reprezentován adresou `/choices`. Nepovinným dotazovým parametrem je zde parametr *fieldName*. Zdroj je dostupný metodou `GET`. Není-li v požadavku předán parametr *fieldName*, jsou vyhledána pole, pro která jsou předdefinovány hodnoty. Seznam těchto polí je následně vrácen jako výsledek požadavku ve formátu JSON. V případě, že je v parametru předán název validního pole, jsou vráceny všechny předdefinované hodnoty, které jsou k tomuto poli definovány pro aktuální tabulku, případně pro tabulku rodičovskou a to včetně hodnot k zobrazení.

Obdobná funkcionalita je nabízena zdrojem **getReferences**. Tento je v systému reprezentován relativní adresou `/references`. Mezi nepovinné parametry patří parametr *field-*

Name a nově také parametr *template*. Zdroj je opět dostupný metodou `GET`. V případě, že není předán parametr *fieldName*, je opět vrácen seznam polí, tentokrát polí typu *Reference* s jejich základní definicí. V případě předání parametru *fieldName*, za předpokladu, že je validní, je navrácen seznam hodnot odpovídajících definici pole. Počet takto vrácených hodnot je omezen hodnotou záznamu `x_273093_itsm_api.core_data.reference_limit`. Pokud je dále předán také parametr template, je v případě vyhledávání referencovaných hodnot využita také hodnota tohoto parametru pro specifičtější vyhledávání v referencované tabulce.

Zdroje **GetChoices** a **getReferences** jsou určeny primárně jako pomocné zdroje pro získávání doplňujících informací, zejména těch informací, které nejsou k dispozici na straně zdroje, který požadavky vysílá, ale stále tyto informace potřebuje pro zadávání validních požadavků, z důvodu konfigurace záznamů na straně ServiceNow instance.

6.2 Integrace

Integrace je výhodným řešením pro komunikaci, zadávání a monitoring ITSM požadavků mezi samostatnými instancemi ITSM nástrojů. V praxi je toto využíváno nejčastěji pro komunikaci mezi zákaznickými a „supportními“ instancemi platformy ServiceNow, případně s ITSM nástroji a aplikacemi třetích stran. Vzhledem k tomu, že se v naprosté většině případů jedná o systémy založené na základě tzv. tiketů, je výhodné udržovat veškerou komunikaci a stavy těchto záznamů konzistentní na obou stranách komunikačního kanálu. Díky tomuto je dosaženo konzistence v pracovním postupu a možnost vyhledávání řešení v již dříve vyřešených záznamech, neboť ty v sobě nesou veškeré potřebné informace.

6.2.1 Požadavky

Cílem implementace je zjednodušit předávání nových požadavků a monitoring těch stávajících mezi instancemi platformy ServiceNow, případně platformou a nástrojem třetí strany. Integrace pomocí REST API je výhodná především z důvodu jednoduché implementace pomocí dostupných nástrojů a také rychlost komunikace mezi koncovými body.

Výchozí tabulkou je, v případě základní integrace, tabulka *Incident* nebo tabulka *Catalog Task*. Cílem může být totožná tabulka na cílové straně, případně tabulka *Case*, používá-li protistrana modul *Customer Service* nebo odpovídající záznam v systému třetí strany.

Pro návrh integrace budeme dále uvažovat jako výchozí právě tabulky *Incident* a *Catalog Task*, cílovou tabulkou bude tabulka *Case*. Tento model je výhodný pro implementaci podpory instance, vzhledem k tomu, že zákazník může zadávat své incidenty, případně po-

žadavky, ve vlastní instanci beze změny. Naproti tomu je na „supportní“ straně pro tyto požadavky založen záznam v *Case* tabulce, kde je typ požadavku určen pomocí kategorie. Tímto je možné obsluhovat všechny zákaznické požadavky na jednom místě.

6.2.2 Možnosti řešení

Pro řešení integrace samotné existuje více řešení. Mezi základní patří:

- Emailová integrace
- REST API integrace

Nejjednodušší možností integrace je integrace pomocí emailové komunikace. Vzhledem k funkcionalitě platformy je možné na základě předem definovaných podmínek odesílat emailové notifikace na jakoukoli adresu, tedy i na adresu spravovanou jinou instancí. Pomocí skriptů definovaných v tabulce *Inbound Email Actions* je možné emaily zpracovávat, tedy na základě podmínek třídit, parsovat jejich obsah a dále ho zpracovávat do záznamů na jiných tabulkách.

Odesílání těchto notifikací je poměrně jednoduché, konfigurace notifikace je uložena v záznamu tabulky *Notifications*. Takto je možné nadefinovat, za jakých podmínek a komu je notifikace odeslána a jaký má být přesně její obsah. Do obsahu zprávy je možné jednoduchým způsobem ukládat hodnoty z polí záznamu tabulky, pro kterou je notifikace definována. Pro složitější operace je možné definovat skripty, které jsou ukládány do záznamů tabulky *Email Scripts* a je možné je volat přímo z definice těla notifikace. Tímto je možné dosáhnout vytvoření strukturovaného emailu, který je možné unifikovaně zpracovávat odpovídajícím způsobem.

Zpracování emailu je na instanci možné pomocí již výše zmíněných skriptů tabulky *Inbound Actions*. Toto může být podmíněno např. předmětem emailu, cílovou adresou případně odpovídající částí těla emailové zprávy. Pomocí těchto skriptů je možné vytvářet nové záznamy v tabulkách, případně, je-li přítomen určitý klíč k párování, upravovat již existující záznamy.

Proces této integrace je jednoduchý. Email je odeslán z instance na základě splněné podmínky, např. změny stavu záznamu nebo přidání komentáře. Tento email je adresován druhému koncovému bodu, tedy další instanci. Zde je email analyzován, zařazen a zpracován dle definovaných podmínek.

Tato metoda integrace může být jednoduše a rychle implementována. Na druhou stranu je možné do integrace jednoduše zasáhnout, známe-li základní nastavení, tedy koncový bod,

výchozí bod a strukturu emailu. Toto je možné primárně z podstaty emailu jako takového. V případě, že je email doručen do instance, je ověření uživatele provedeno pouze na základě emailové adresy. Kdokoli tedy může odeslat podvržený email a do instance odeslat např. nevalidní data.

Další nevýhodou je rychlost tohoto řešení. Odeslání i zpracování emailu je v platformě velmi rychlou a jednoduchou operací, nicméně samotná cesta emailu může dobu mezi úpravou na jedné instanci a odpovídající změnou na instanci druhé značně prodloužit. Dalším prvkem, který tento interval prodlužuje, je i perioda kontroly emailové schránky. Ta je většinou s instancí propojena pomocí protokolu **POP3** nebo **IMAP** a perioda kontroly se nejčastěji pohybuje okolo pěti minut. V případě integrace tedy může být okno pro provedení změny téměř nulové nebo naopak až nepřijatelně dlouhé. Doba přenosu dat tedy není konstantní a závisí na čase poslední kontroly schránky a čase přijetí emailu.

Oproti tomuto je REST API integrace poměrně rychlou a spolehlivou volbou. Koncový bod vyžaduje pro komunikaci ověření pomocí přihlašovacích údajů, dále je možné pro REST API využít ověření podle definovaných ACL pravidel. Je tedy mnohem složitější do této komunikace zasáhnout. Komunikace mezi servery je také velmi rychlá a změny provedené na jedné instanci jsou přeneseny na druhou instanci téměř okamžitě.

Implementace REST API integrace je oproti emailové integraci poměrně složitá, vyžaduje vytvoření *Business Rule* pravidel nad integrovanými tabulkami a odchozí REST zprávy, pro odeslání na druhou instanci. Pro integraci je také nutné mít k dispozici odpovídající REST API pro přístup k datům jednotlivých tabulek.

Pro REST API integraci je možné využít dvě metody přístupu k řešení a to:

- Pull integrace
- Push integrace

Integrace založená na **pull** principu pracuje s daty, která si logika vyžádá z druhého serveru na základě notifikace o změně dat. Pro tento princip je typická obousměrná komunikace mezi servery, kdy při změněn dat na výchozím serveru je odeslán REST API požadavek na cílový server, který na základě tohoto požadavku provede vlastní volání zpět na výchozí server.

Toto řešení je výhodné hlavně v případě, že výchozí server nemá možnost komplexnější možnosti tvorby vlastních odchozích REST API volání, tedy umožňuje například pouze základní komunikaci, jako např. odeslání identifikátoru na cílový server. Nutností je v tomto případě existence přístupového API na straně výchozího serveru.

Výhodou tohoto řešení je možnost jednoduché implementace na výchozím serveru, zde je požadováno pouze odeslání požadavku s informací, že byla provedena změna. Veškerá logika je tedy implementována na straně cílového serveru, nutná je pouze znalost přístupového API výchozího serveru.

Nevýhodou může být vetší celkový objem komunikace, který je nutný pro každou ze změn. Vzhledem k nepřítomnosti další logiky na cílové straně je také omezena jakákoli úprava dat před jejich přenosem, tedy není např. možné formátovat či jinak upravit hodnoty polí, která jsou součástí integrace.

Jak již bylo zmíněno, tento typ integrace je vhodný zejména pro integraci různorodých systémů, např. instance ServiceNow s instancí ITSM nástroje *JIRA*.

Na druhou stranu, integrace **push** principu nabízí komunikačně jednodušší řešení, vyžaduje však možnost implementovat logiku odchozích REST API požadavků na obou integrovaných stranách. V případě změny dat jsou pak data odeslána z výchozího serveru přímo na cílový.

Tento způsob integrace může být implementačně složitější, nicméně je komunikačně jednodušší, neboť je v případě změny dat odeslán vždy pouze jeden požadavek. Ve velkém množství případů se toho využívá pro integraci architektonicky velmi podobných či stejných systémů, např. dvou instancí ServiceNow. Implementace tohoto způsobu je také vhodná pro obousměrnou synchronizaci dat.

Toto řešení je výhodné zejména při použití např. pro obousměrnou integraci dvou ServiceNow instancí. V případě využití tohoto řešení je nutné implementovat pouze komunikaci jedním směrem. Vzhledem ke shodné architektuře je pak možné pro komunikaci opačným směrem využit tu samou implementaci, pouze s odlišnou konfigurací.

Ne vždy je však možné jednoduchou implementaci integrace uskutečnit. V těchto případech je výhodné využít pro tento typ integrace modelu klient-server, kde je možné využít server, tedy instanci, do které chceme integrovat a několik klientských instancí. V praxi je s tímto modelem možné integrovat několik „zákaznických“ instancí do jedné „supportní“ instance.

6.2.3 Návrh architektury

Vzhledem k tomu, že cílem je integrovat ServiceNow instance, je pro návrh implementace vybrán klient-server model *push* integrace, která umožní jednodušší použití a konfiguraci.

Serverová část je implementována na „supportní“ instanci. Zde jsou pro sledování požadavků využity záznamy v *Case* tabulce. Pro klientskou část, která bude umístěna na „zákaznické“ instanci, budou využívány záznamy tabulek *Incident* a *Catalog Task*. Obě strany využívají vlastní logiku pro rozhodování o podmínkách a „cíli“ integrace.

Klientská strana implementace vyžaduje podmínku pro spuštění integrace. Cíl této integrace je jasný, klientská strana se připojuje pouze na jednu serverovou stranu. Vzhledem k využití a podstatě integrovaných záznamů je výhodné využít jako podmínku skupinu, případně uživatele, kterému je záznam přiřazen. Díky této podmínce je možné na klientské instanci samostatně pracovat na požadavcích a incidentech interně, bez odesílání informací do jiné instance. K integraci záznamu na cílovou, „supportní“ instanci, dojde až ve chvíli přiřazení na externí řešitelskou skupinu, případně uživatele. Následně je na cílové instanci založen záznam odpovídajícího typu a externí řešitelé mohou začít pracovat.

Ke každé straně integrace je tedy nutné přistupovat individuálně, neboť každá ze stran vyžaduje lehce odlišný přístup ke komunikaci s protistranou.

Společná konfigurace

Vzhledem k odlišnostem stavů výchozí a cílové tabulky je dobré uvažovat také o konfiguraci základních přechodů mezi stavy, případně určit správné vzájemné mapování jednotlivých stavů integrovaných tabulek.

Společnou konfigurací obou stran je v případě mimo jiné integrace mapování hodnot pro pole *Priority* a v případě incidentů i pole *Impact* a *Urgency*. Nejdůležitějším polem je pole *Priority*, které určuje závažnost incidentu nebo naléhavost požadavku. Toto pole by mělo být nastavováno pouze ze strany klientské instance, není požadováno aby byla určována externími řešiteli. Pole *Impact* a *Urgency* jsou podstatná pouze pro incidenty a pro řešitele mají pouze informativní charakter. Pole tedy mohou být integrována, nicméně je vhodné, vzhledem jejich nepřítomnosti na formuláři záznamu na serverové straně, uložit jejich hodnoty odpovídajícím způsobem např. do komentářů záznamu, kde budou pro řešitele viditelné.

Klientská strana

Klientská strana integrace je soustředěna na integraci záznamů z tabulek *Incident* a *Catalog Task*. Pro správnou funkcionalitu je nutné zajistit komunikaci se serverovou stranou integrace a to za specifických podmínek.

Pro identifikaci záznamu, do kterého je výchozí záznam integrován lze výhodně využít polí *Correlation ID* a *Correlation Display*. K získání identifikátoru je však nejprve nutné tento záznam vytvořit na serverové straně integrace. Pro tento účel je vhodné vytvořit *Business Rule*, která v případě přiřazení záznamu na externí řešitelskou skupinu, provede volání na serverovou stranu za účelem vytvoření nového záznamu. V případě úspěšného uložení budou zpět vráceny potřebné hodnoty, tedy identifikátor `sys_id` záznamu a číslo záznamu, pro jednodušší komunikaci.

Pro přenášení následných změn je vhodné implementovat další *Bsiness Rule* záznam, který provede volání v případě, že se změnila některá ze sledovaných hodnot a zároveň je záznam stále přiřazen na externí řešitelskou skupinu. Podmínkou je zde také již existující identifikátor vzdáleného záznamu v poli *Correlation ID*, bez něj by nebylo možné záznam na serverové straně dohledat.

Speciálně pro záznamy nad *Catalog Task* tabulkou je také vhodné připravit logiku pro čtení a zpracování *variable* záznamů, tedy dat, které jsou k záznamu přiřazeny z objednané katalogové položky. Tyto záznamy jsou často stěžejní pro řešení požadavků, nicméně nejsou pevně definovány na záznamu tabulky *Catalog Task* a jsou závislé na katalogové položce, ke které záznam náleží. *Variable* záznamy jsou uloženy v samostatné tabulce a je tedy nutné je dohledávat zvlášť. Záleží však na konfiguraci instance samotné, ne vždy je nutné s těmito záznamy pracovat, např ve výchozí konfiguraci tyto záznamy nejsou na formuláři *Catalog Task* tabulky viditelné a není je tedy nutné integrovat.

Serverová strana

Serverová logika je soustředěna nad *Case* tabulkou. Zde jsou záznamy, dle kterých mohou řešitelé pracovat a vzájemně komunikovat, případně komunikovat se zadavatelem na zákaznické instanci. Podmínky pro integraci jsou zde poněkud složitější, opět je nutné mít k dispozici identifikátor záznamu na zákaznické instanci, nicméně pro určení této instance je nutná dodatečná konfigurace. Jedna serverová strana integrace může být napojena na několik klientských, k identifikaci správné instance tedy nestačí pouze vědět, že záznam má být integrován, ale je nutné také znát cíl integrace. Toto lze definovat např. v konfiguračním záznamu, který bude vytvořen po každého z klientů a např. na základě kontaktu,

který je přiřazen ke *Case* záznamu, pak lze rozhodnout, ke které klientské straně je nutné se připojit.

Jak je již z prvního popisu vidět, není vhodné využívat pro podmínku integrace přiřazení na řešitelskou skupinu. Toto je důležité zejména z toho důvodu, že na serverové, řešitelské, instanci může existovat několik řešitelských skupin, které si práci předávají dle kompetencí.

Pro integraci je opět možné využít *Business Rule*. V tomto případě je však nutné spustit integraci jen v případě, že dojde ke změně sledovaných dat. Vytváření záznamů na klientské straně není cílem, serverová strana by měla sloužit k řešení incidentů a požadavků z klientských instancí, ne k jejich vytváření.

Integrované záznamy na serverové straně tedy musí vždy obsahovat identifikátor záznamu v tabulce klientské strany, uložený v poli *Correlation ID*. Jen díky tomuto je možné odpovídající záznam vyhledat a přenést potřebná data.

6.2.4 Implementace

Implementace integrace je opět rozdělena na dvě hlavní části a to na *klientskou* a *serverovou* část. Toto rozdělení je nutné zejména z důvodu lehce odlišného způsobu přístupu k integraci, převážně díky rozdílnému počtu protistran ke komunikaci.

Pro implementaci integrace je vytvořen vlastní aplikační rozsah `x_273093_itsm_int` , určený nově vytvořenou aplikací **ITSM Integration**. Nová implementace logiky a provedené změny jsou ukládány do Update Set záznamu *ITSM Integration Core*.

Integrace využívá metody vytvořené v rámci výše zmíněné implementace rozhraní **ITSM API**.

Společná konfigurace

Pro konfiguraci jsou vytvořeny záznamy v tabulce `sys_properties`. Tyto záznamy jsou identifikovány klíčem se společnou strukturou, díky které je možné využívat pro jejich vyhledávání univerzální přístup. Konfigurace, v nich uložená, je uložena ve struktuře JSON řetězce. Záznamy využívají společný prefix `x_273093_itsm_int` a suffix je vytvořen na základě zdrojové a výchozí tabulky, viz 6.3.

Suffix	Zdroj	Cíl
integration_config.case_to_incident	Case	Incident
integration_config.case_to_sc_task	Case	Cat. Task
integration_config.incident_to_case	Incident	Case
integration_config.sc_task_to_case	Cat. Task	Case

Tabulka 6.3: Suffixy konfiguračních záznamů integrovaných tabulek

Základem konfiguračních záznamů jsou definice mapování hodnot vybraných polí a seznam sledovaných, integrovaných, polí. Základ struktury JSON reprezentace je zachycen v ukázce 6.14.

```
1  {
2      "mapping": {
3          "state": {
4              "-5": "18",
5              ...
6          }
7      },
8      "watched_fields": [
9          "description",
10         ...
11     ]
12 }
```

Ukázka kódu 6.14: Ukázka těla konfiguračního záznamu *(Vlastní zpracování)*

Díky mapování důležitých polí je možné pracovat s rozdílnými definicemi voleb jednotlivých polí a je tedy možné tato pole integrovat, tedy nastavit odpovídající hodnoty na základě těchto definic.

Dále je vytvořen uživatelský účet pro integraci samotnou a s jeho pomocí také základní autorizační profil k přístupu k REST API. Tento je v rámci prototypu integrační aplikace využíván pro obě strany integrace.

Společnou logikou pro integraci samotnou je třída `ITSM_Integration_DataLogic`, uložená ve stejnojmenném Script Include záznamu. Logika této třídy slouží k načtení integrační konfigurace a konstrukci těla požadavku pro odeslání na druhou instanci.

Klientská strana

Klientská strana integrace je je implementačně o něco jednodušší než strana serverová. Pro komunikaci se serverem je vytvořena REST Message **Integration server**. Záznam obsahuje cestu k cílovému API, v tomto případě ke **Case API** serverové strany. Dále je zde nastaven autorizační profil a definice HTTP hlaviček.

Pro jednodušší komunikaci se serverem jsou zde definovány také jednotlivé metody pro potřebné operace. Pro vytvoření nového záznamu na serverové straně je zde `POST` metoda **Create Case**, pro úpravu dat cílového záznamu je definována `PUSH` metoda **Update Case** a pro komunikaci `POST` metoda **Add comment**.

Jednotlivé metody obsahují cestu k cílové metodě API a v případě potřeby také definice substitucí proměnných v adrese API.

Pro vykonání potřebných operací jsou vytvořeny *Business Rule* záznamy nad tabulkami *Incident* a *Catalog Task*.

```
1  (function executeRule(current, previous /*null when async*/) {
2      var utils = new
           x_273093_itsm_int.ITSM_Integration_DataLogic('incident',
           'case');
3
4      var messageBody = utils.prepareRequestBodyInsert(current);
5      var request = new sn_ws.RESTMessageV2('Integration server',
           'Create Case');
6      request.setRequestBody(JSON.stringify(messageBody));
7      var response = request.execute();
8
9      if(response.getStatusCode() == 200){
10         var body = JSON.parse(response.getBody());
11         var data = body.result.data;
12         current.setValue('correlation_id', data.sys_id.value);
13         current.setValue('correlation_display', data.number.value);
14         current.update();
15     }
16 })(current, previous);
```

Ukázka kódu 6.15: Ukázka těla BR záznamu ITSM Integration initialize record

(Vlastní zpracování)

První potřebnou operací je vytvoření záznamu na serverové instanci. Pro tuto operaci je určen záznam **ITSM Integration initialize record**. Implementovaná funkcionalita se postará o načtení konfigurace pro zadanou kombinaci zdrojové a cílové tabulky, vytvoření těla požadavku, inicializaci a konfiguraci požadavku **Create Case** a jeho provedení. V případě úspěchu jsou ze serverové strany integrace navráceny data nově vytvořeného požadavku a jeho identifikátor a číslo jsou uloženy do zdrojového záznamu, pro identifikaci cíle pro případné úpravy. Kód tohoto záznamu je zachycen v ukázce 6.15. Tato logika je spuštěna pouze v případě, že jsou splněny všechny podmínky pro integraci záznamu, tedy záznam je přiřazen na externí řešitelskou skupinu, *Correlation ID* je prázdné a záznam není aktuálně upravován integračním uživatelem, čímž je zamezeno vytváření nepotřebných záznamů.

Obdobnou logiku implementuje i záznam **ITSM Integration update record**, který slouží k úpravě dat na cílové instanci. Podmínkou pro spuštění této logiky je trvající přiřazení záznamu na externí řešitelskou skupinu a existence identifikátoru v poli *Correlation ID*, který je využit k identifikaci cílového záznamu na serverové straně integrace. Pro správnou funkcionalitu je také podmínkou, že záznam nesmí být aktuálně upraven integračním uživatelem. V opačném případě by při obousměrné integraci docházelo k rekurzivnímu volání těchto operací, vedoucí k opakovanému upravování již upravených hodnot.

Posledním důležitým záznamem je **ITSM Integration add comment**, sloužící ke komunikaci pomocí komentářů. Díky tomuto je možné i po přiřazení na externí řešitele monitorovat aktuální průběh řešení, případně komunikovat přímo s řešitelem, např. kvůli upřesnění specifikací požadavku nebo jejich změně. Vykonávaný kód je v tomto případě poněkud jednodušší, není nutné načítat do těla požadavku všechna pole, pouze poslední přidaný komentář. Oproti předchozí implementaci v záznamu pro úpravu dat, je zde přidána podmínka, která omezuje spuštění této logiky pouze na případy, kdy je provedena změna komentáře, tedy přesněji jeho přidání.

Serverová strana

Serverová strana integrace vyžaduje lehce složitější implementaci, nelze zde využít pouze jednu REST Message, jako v případě klientské strany integrace. Vzhledem k povaze této strany integrace by bylo nutné pro každého z klientů vytvořit vlastní REST Message záznam. Jednodušší volbou je vytvoření konfiguračního záznamu, nesoucí v sobě cestu k cílovým API jednotlivých klientů, v tomto případě je konfigurace uložena v `sys_property` se suffixem `endpoint_configuration`. Při odesílání dat je tak nutné v logice k tomu určené

provést také zjištění odpovídajícího cíle, dle pole *Account* na záznamu tabulky *Case*.

Pro úpravu dat a komunikaci je implementována obdobná logika, jako v případě klientské strany. Tato logika se liší v podmínkách spuštění, kdy je nutné pracovat s hodnotou pole *Account* a kategorií záznamu a také v kódu, kdy je nutné při definici požadavku určit správné cílové API a doplnit potřebná data přímo ve skriptu.

Vzhledem k tomu, že cílem integrace není vytvářet záznamy na klientské instanci, není zde nutná ani definice operace pro vytváření záznamu.

Kapitola 7

Diskuze

7.1 Význam výsledků

Implementace ITSM API umožňuje jednodušší tvorbu aplikací pro zpracovávání ITSM požadavků a jejich monitorování. Díky tomuto API je možné přistupovat k ITSM záznamům jednodušším způsobem a není nutné znát strukturu celé databáze. Data, která jsou pro tyto tabulky důležitá a nejsou uložena přímo v nich, je možné získat pomocí jednoho volání na toto API, namísto několika požadavků pro získání konfigurace, cíle dotazu a poté i samotných dat.

Tímto přístupem je možné optimalizovat počet dotazů na instanci, případně omezit nutnost jiných řešení, jako například kódování potřebné logiky přímo do těla vytvářené aplikace. Výhodou je také to, že v případě změny definice hodnot, některého z takto získávaných polí, se tato změna promítne do těchto výsledků okamžitě.

Díky tomuto API lze také jednoduše a rychle implementovat integrační řešení a to jak mezi několika instancemi ServiceNow, tak i mezi instancí a aplikacemi třetích stran, jako je např. JIRA.

Implementovaná integrace v aplikaci ITSM Integration slouží k jednoduché a rychlé synchronizaci ITSM záznamů mezi jednotlivými ServiceNow instancemi. K tomuto je výhodně využit klient-server model, čímž je možné dosáhnout integrace několika samostatných instancí do jedné cílové.

Tento model je možné využít pro integraci „zákaznických“ požadavků na „řešitelskou“ instanci, tedy je to vhodným řešením pro firmy poskytující podporu ITSM procesů svým zákazníkům využívajícím platformu ServiceNow. Vzhledem k poměrně jednoduché struktuře integrace a formě její konfigurace je toto řešení jednoduše rozšiřitelné a upravitelné dle konfigurace jednotlivých instancí.

7.2 Možnosti rozšíření

ITSM API je možné vhodně rozšířit i pro další tabulky vycházející z tabulky *Task*, čehož je, díky struktuře implementovaných tříd, možné dosáhnout implementací potřebných tříd pro tyto tabulky. Pro každou z nových tříd je pak nutné pouze vytvořit nový přístupový bod obsahující vyžadované metody.

Mimo implementaci nových zdrojů je také další možností vytvoření komplexnějších metod pro získávání hodnot referenčních polí. Aktuální implementace dokáže pracovat pouze s jednoduchými podmínkami. Vhodným rozšířením je možné dosáhnout vyhodnocení složitějších podmínek, které vyžadují například spuštění skriptu nebo hodnoty polí záznamu, které není možné v rámci aktuálně implementovaného řešení získat.

Pro vyšší využitelnost a implementaci složitějších aplikací je možné stávající API rozšířit také o metody pro získávání dodatečných dat, jako např. záznamů které se na požadovaný záznam odkazují nebo jsou v rámci tohoto záznamu odkazovány. Tato část logiky je již částečně implementována v metodách pro získávání dat pro referenční pole.

Vhodným rozšířením řešení integrace je přidání integrace příloh záznamů, díky čemuž by bylo možné přenášet doplňující data, jako např. obrázky, datové soubory či další dokumenty. Další prvky integrace jsou již závislé na jednotlivých instancích. Pro většinu instancí je třeba pouze upravit konfiguraci tak, aby odpovídala konfiguraci klientské instance, tedy např. upravit mapování stavů a priorit. Pro synchronizaci dalších dat je možné využít také pouze úpravy konfigurace. V případě, že je cílem do integrace zahrnout hodnoty z dalších polí, které na cílové instanci nejsou přítomny, je nutné vytvořit nové metody, které tyto hodnoty zpracují takovým způsobem, aby bylo možné je přenést a následně uložit na cílové instanci.

Rozšíření funkcionality integrace je tedy silně závislé na konfiguraci integrovaných instancí, jelikož ty se mohou velmi lišit. Mezi produkčními instancemi lze narazit jak na naprosto neupravené, tak i velmi silně upravované ITSM tabulky a není tedy dopředu možné určit, jak rozsáhlá musí být funkcionalita poskytovaná integrací. Prakticky je tedy možné říci, že pro prvotní nasazení je možné využít prototyp implementovaný v rámci této práce, nicméně pro každého zákazníka je vhodné připravit integraci „na míru“, dle jeho potřeb a specifikací instance.

Závěr

Pro diplomovou práci, zabývající se možnostmi tvorby REST API a na něj navazující integrace, byly stanoveny tři hlavní cíle:

- seznámení čtenáře s platformou ServiceNow a problematikou vývoje
- návrh struktury a řešení vhodného REST API
- návrh implementace integrace využívající připravené API

Prvním z cílů je seznámení čtenáře s platformou ServiceNow, která není v České republice příliš známá. Tuto platformu využívají především velké zahraniční firmy a většina vývojářů se s ní běžně nesetkává. Platforma je tedy popsána jak z hlediska základního uživatelského rozhraní, tak i principu fungování a jejích možností zejména v oblasti ITSM.

Pro seznámení s vývojem na platformě jsou popsány základní technologie, se kterými je možné se na platformě setkat a také specifika, která jsou pro vývoj důležitá. Jedná se primárně o seznámení s rozsahy platností aplikací, reprezentaci tříd logiky, využívání *Business Rule* záznamů, rozdíly mezi vývojem na straně klienta a serveru a využitím základních rozhraní, která je nutné znát pro práci s daty.

Oba praktické cíle jsou rozděleny do čtyř navazujících částí, definujících požadavky na řešení, možnosti, návrh a samotnou implementaci.

V případě druhého z cílů, tedy návrhu a řešení REST API, byly jasně definovány požadavky na toto API, které jsou následně využity pro definici různých možností řešení. Každá z možností řešení je popsána a čtenář je seznámen s jejím možným řešení, výhodami i nevýhodami. Na základě definovaných požadavků bylo vybráno optimální řešení, pro které je dále zpracován návrh architektury a následně také samotná implementace.

Výsledný prototyp, tedy aplikace ITSM API, je použitelný pro základní operace se záznamy na ITSM tabulkách *Incident*, *Catalog task*, *Case* a *Task* a je dále využit pro implementaci prototypu integračního řešení.

Třetí z cílů je zaměřen na řešení integrace s využitím aplikace ITSM API. Pro integraci samotnou byly stanoveny cíle, kterých bylo nutné dosáhnout a možnosti jejího řešení. Z

těchto možností byl vybrán optimální přístup, pro který je dále popsán návrh architektury. Výsledkem implementace je prototyp aplikace ITSM Integration, která je, díky upravitelné konfiguraci, použitelná pro integraci několika ServiceNow instancí v modelu klient-server.

Výsledky diplomové práce lze označit za velmi přínosné. Kromě přínosu v podobě seznámení čtenáře s platformou, základními principy vývoje na ní a teoretickým návrhem řešení API a integrace, je přínosem také výsledek praktické části práce, tedy prototypy navrhovaných aplikací. Stanovené cíle tedy lze považovat za splněné a to nejen z pohledu teorie a návrhu aplikací, který může být použit jako předloha pro tvorbu obdobných řešení, ale také s ohledem na jednoduchost a rozšiřitelnost vytvořených řešení.

Použité zkratky a pojmy

ACL Access control list
GUID Globally Unique ID
ITSM Information Technology Service Management
JSON JavaScript Object Notation
OOTB Out Of The Box
PaaS Platform as a Service
PDI Personal Developer Instance
REST Representational state transfer
SaaS Software as a Service
SLA Service Level Agreements
VCS Version control system

Seznam obrázků

2.1 Grafické rozhraní platformy ServiceNow *(Vlastní zpracování)* 4

2.2 Záznamy v Seznamu aktivit *(Vlastní zpracování)* 6

3.1 Formulář Incidentu v ServiceNow *(Vlastní zpracování)* 9

3.2 Životní cyklus incidentu *(Převzato z ServiceNow (2020k))* 10

3.3 Hierarchie Request Managementu *(Převzato z ServiceNow (2020m))* 12

6.1 Struktura samostatných API *(Vlastní zpracování)* 38

6.2 Struktura jednotného API *(Vlastní zpracování)* 39

6.3 Struktura hybridního API *(Vlastní zpracování)* 40

6.4 Struktura implementace *(Vlastní zpracování)* 45

Seznam tabulek

6.1 Společná pole tabulek dědících z tabulky Task 42
6.2 Pole definovaná na potomcích tabulky Task 43
6.3 Suffixy konfiguračních záznamů integrovaných tabulek 68

Seznam ukázek kódu

4.1 Ukázka formátu JSON *(Převzato z (Crockford n.d.[b]))* 17
5.1 Ukázka práce s GlideRecord API *(Vlastní zpracování)* 23
5.2 Ukázka práce s GlideAggregate API *(Vlastní zpracování)* 24
5.3 Využití gs.getProperty() *(Vlastní zpracování)* 26
5.4 Třída deklarovaná v rámci SI *(Vlastní zpracování)* 27
6.1 Struktura konfiguračního záznamu *(Vlastní zpracování)* 47
6.2 Inicializační funkce konfigurační třídy *(Vlastní zpracování)* 48
6.3 Inicializační funkce hlavní přístupové třídy *(Vlastní zpracování)* 49
6.4 Funkce getTicket hlavní přístupové třídy *(Vlastní zpracování)* 50
6.5 Funkce insertTicket hlavní přístupové třídy *(Vlastní zpracování)* 51
6.6 Funkce getChoiceFields hlavní přístupové třídy *(Vlastní zpracování)* 52
6.7 Funkce _getChoicesForField hlavní přístupové třídy *(Vlastní zpracování)* . . 53
6.8 Reprezentace hodnot pro pole nad tabulkou Task *(Vlastní zpracování)* . . . 54
6.9 Reprezentace objektu definice referenčního pole tabulky Task *(Vlastní zpracování)* 56
6.10 Ukázka definice třídy rozšířené logiky ITSM_API_ExtendedDataLogic *(Vlastní zpracování)* 57
6.11 Ukázka logiky zdroje GetTicket *(Vlastní zpracování)* 59
6.12 Ukázka těla požadavku pro změnu záznamu *(Vlastní zpracování)* 60
6.13 Ukázka těla požadavku pro přidání zprávy *(Vlastní zpracování)* 60
6.14 Ukázka těla konfiguračního záznamu *(Vlastní zpracování)* 68
6.15 Ukázka těla BR záznamu ITSM Integration initialize record *(Vlastní zpracování)* 69

Bibliografie

Atlassian (2020). *Software Development and Collaboration Tools*. URL: https://www.atlassian.com/software/jira. [cit. 11.05.2020].

Axelos (2019). *ITIL® Foundation, ITIL 4 edition*. London: Stationery Office. ISBN: 9780113316076.

Crockford, Douglas (n.d.[a]). *JSON*. URL: http://www.json.org/. [cit. 17.05.2020].

— (n.d.[b]). *JSON example*. URL: http://www.json.org/example.html. [cit. 17.05.2020].

Fielding, Roy Thomas (2000). *Architectural Styles and the Design of Network-based Software Architectures. Chapter 5: Representational State Transfer (REST)*. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. [cit. 29.04.2020].

International, ECMA (2020). *Standard ECMA-262*. URL: https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf. [cit. 29.04.2020].

Organization, Linux Kernel (n.d.). *Git User Manual*. URL: https://mirrors.edge.kernel.org/pub/software/scm/git/docs/user-manual.html. [cit. 29.04.2020].

ServiceNow (2019). *Glide Server APIs*. URL: https://docs.servicenow.com/bundle/orlando-application-development/page/script/glide-server-apis/topic/p_GlideServerAPIs.html. [cit. 28.04.2020].

— (2020a). *Application scope*. URL: https://docs.servicenow.com/bundle/orlando-application-development/page/build/applications/concept/c_ApplicationScope.html. [cit. 17.05.2020].

— (2020b). *Business rules*. URL: https://docs.servicenow.com/bundle/orlando-application-development/page/script/business-rules/concept/c_BusinessRules.html. [cit. 05.06.2020].

— (2020c). *Case Management*. URL: https://docs.servicenow.com/bundle/orlando-customer-service-management/page/product/customer-service-management/concept/configure-csm-case-management.html. [cit. 10.06.2020].

ServiceNow (2020d). *Client scripts*. URL: https://docs.servicenow.com/bundle/orlando-application-development/page/script/client-scripts/concept/client-scripts.html. [cit. 07.06.2020].

— (2020e). *Customer Service Management*. URL: https://docs.servicenow.com/bundle/orlando-customer-service-management/page/product/customer-service-management/concept/c_CustomerServiceManagement.html. [cit. 16.06.2020].

— (2020f). *Forms*. URL: https://docs.servicenow.com/bundle/orlando-platform-user-interface/page/use/using-forms/concept/c_UsingForms.html. [cit. 10.04.2020].

— (2020g). *GlideAggregate - Global*. URL: https://docs.servicenow.com/bundle/orlando-application-development/page/app-store/dev_portal/API_reference/GlideAggregate/concept/c_GlideAggregateAPI.html. [cit. 28.04.2020].

— (2020h). *GlideDateTime*. URL: https://docs.servicenow.com/bundle/geneva-servicenow-platform/page/script/server_api/concept/c_GlideDateTime.html. [cit. 12.06.2020].

— (2020i). *GlideRecord*. URL: https://developer.servicenow.com/dev.do#!/reference/api/madrid/server/no-namespace/c_GlideRecordScopedAPI. [cit. 10.06.2020].

— (2020j). *GlideSystem - Global*. URL: https://docs.servicenow.com/bundle/orlando-application-development/page/app-store/dev_portal/API_reference/GlideSystem/concept/c_GlideSystemAPI.html. [cit. 28.04.2020].

— (2020k). *Incident Management*. URL: https://docs.servicenow.com/bundle/orlando-it-service-management/page/product/incident-management/concept/c_IncidentManagement.html. [cit. 08.03.2020].

— (2020l). *Product documentation*. URL: https://docs.servicenow.com/bundle/orlando-platform-user-interface/page/use/using-lists/concept/c_UseLists.html. [cit. 14.04.2020].

— (2020m). *Request Management architecture*. URL: https://docs.servicenow.com/bundle/orlando-it-service-management/page/product/planning-and-policy/concept/request-management-architecture.html. [cit. 14.06.2020].

— (2020n). *REST API*. URL: https://docs.servicenow.com/bundle/orlando-application-development/page/integrate/inbound-rest/concept/c_RESTAPI.html. [cit. 30.04.2020].

ServiceNow (2020o). *Retrieve an update set.* URL: https://docs.servicenow.com/bundle/orlando-application-development/page/build/system-update-sets/task/t_RetrieveAnUpdateSet.html. [cit. 30.04.2020].

— (2020p). *Script Includes.* URL: https://docs.servicenow.com/bundle/orlando-application-development/page/script/server-scripting/concept/c_ScriptIncludes.html. [cit. 07.03.2020].

— (2020q). *Scripted REST APIs.* URL: https://docs.servicenow.com/bundle/orlando-application-development/page/integrate/custom-web-services/concept/c_CustomWebServices.html. [cit. 17.05.2020].

— (2020r). *Service Catalog.* URL: https://docs.servicenow.com/bundle/orlando-it-service-management/page/product/service-catalog-management/concept/c_ServiceCatalogManagement.html. [cit. 14.06.2020].

— (2020s). *Service catalog items.* URL: https://docs.servicenow.com/bundle/orlando-it-service-management/page/product/service-catalog-management/concept/c_IntroductionToCatalogItems.html. [cit. 14.06.2020].

— (2020t). *Source control integration.* URL: https://docs.servicenow.com/bundle/orlando-application-development/page/build/applications/concept/c_SourceControlIntegration.html. [cit. 29.04.2020].

— (2020u). *System update sets.* URL: https://docs.servicenow.com/bundle/orlando-application-development/page/build/system-update-sets/concept/system-update-sets.html. [cit. 28.04.2020].

— (2020v). *UI policies.* URL: https://docs.servicenow.com/bundle/orlando-platform-administration/page/administer/form-administration/task/t_CreateAUIPolicy.html. [cit. 12.06.2020].

— (2020w). *Update set transfers.* URL: https://docs.servicenow.com/bundle/orlando-application-development/page/build/system-update-sets/reference/update-set-transfers.html. [cit. 29.04.2020].

— (2020x). *Workflow.* URL: https://docs.servicenow.com/bundle/orlando-servicenow-platform/page/administer/workflow/concept/c_WorkflowOverview.html. [cit. 14.06.2020].

T.Garfolo, Blaine (2002). *Encyclopedia of Information Systems.* Academic Press. ISBN: 9780122272400.

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor: **Bc. Lubomír Mrtvý**

Studium: I1800329

Studijní program: N1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

Název diplomové práce: **REST API pro podporu a integraci ITSM procesů**
Název diplomové práce AJ: REST API for support and integration of ITSM processes

Cíl, metody, literatura, předpoklady:

Cíl práce:

Návrh REST API pro podporu a integraci ITSM v rámci platformy ServiceNow. Navrhované REST API bude sloužit jako vstupní bod pro integraci s dalšími instancemi ServiceNow, případně s aplikacemi třetích stran. Součástí práce je také návrh integrace využívající toto rozhraní.

Osnova:

1. Úvod
2. Teoretická část
    1. ServiceNow
        1. Představení
        2. GUI
    2. ITSM aplikace v ServiceNow
        1. Incident Management
        2. Service Catalog
        3. Request Management
        4. Customer Service Management
    3. Technologie
        1. Javascript
        2. REST API
        3. Verzování
            1. Update sets
            2. Git
    4. Architektura platformy
        1. Javascript
        2. Databáze
        3. Aplikace
        4. API pro práci s daty
        5. Skriptování na straně serveru
        6. Skriptování na straně klienta
        7. REST API
        8. Update sets
3. Praktická část
    1. Analýza a návrh
        1. Požadavky
        2. Možnosti řešení
        3. Návrh architektury
        4. Implementace
    2. Dokumentace

ITIL Foundation, ITIL 4 edition. TSO (The Stationery Office). ISBN 978-0113316076.

Product Documentation | ServiceNow. Product Documentation | ServiceNow [online]. Copyright {\copyright [cit. 23.07.2020]. Dostupné z: https://docs.servicenow.com

Garantující pracoviště: Katedra informačních technologií, Fakulta informatiky a managementu

Vedoucí práce: Ing. Zuzana Němcová, Ph.D.

Oponent: Mgr. Daniela Ponce, Ph.D.

Datum zadání závěrečné práce: 21.10.2014