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

Diplomová práce

Automatizace procesu sběru a analýzy chybových hlášení

Plzeň 2015 Bc. Tomáš Rojík

Prohlášení

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

V Plzni dne 11. května 2015

Bc. Tomáš Rojík

Abstract

The aim of this thesis is to replace the obsolete system for processing of Kerio products errors. The motivation is to facilitate analysis and aggregation by automation of error reports processing. Other requirements for functionality are integration with internal systems, and complete the missing functionality. Analysis of the previous solution shows its insufficiencies. Proposed solution provides speedup of data retrieval from a database by changing of the database structure. From the user point of view, the processing of error reports has been simplified. This includes a reduction of a number of events that must be performed in the processing of error reports.

Cílem diplomové práce je nahrazení nevyhovující aplikace na zpracování chybových hlášení z produktů společnosti Kerio. Motivací je usnadnění analýzy a agregace formou automatizace části procesu zpracování chybových hlášení. Mezi další požadavky na funkčnost patří integrace s interními systémy a doplnění chybějící funkčnosti. Při analýze stávajícího řešení byly stanoveny jeho nedostatky. V rámci práce se podařilo řádově urychlit načítání dat z databáze, a to změnou struktury databáze. Dále se podařilo zjednodušit proces zpracování z uživatelského hlediska, což zahrnuje i redukci počtu akcí, které je nutné při zpracování nahlášených chyb provést.

Obsah

1 Úvod

Kerio Technologies se zabývá, mimo jiné, vývojem webových aplikací pro usnadnění komunikace a spolupráce ve společnostech. Webové aplikace jsou jako takové zdrojem většího množství chyb než klasické desktopové aplikace. Důvodem většího množství chyb je, že fungují na větším množství platforem, v předem neurčeném množství prohlížečů a různé webové prohlížeče používají různé JavaScriptové enginy. Navíc i samotné webové prohlížeče obsahují chyby, což je nejzřejměji vidět na jejich častých aktualizacích.

Důsledkem výše popsaného je, že i jednoduchá chyba v aplikaci může být koncovými uživateli nahlášena v mnoha různých variacích. Toto mnohonásobné hlášení stejné chyby způsobuje poměrně značné problémy, jelikož je nutné každé hlášení analyzovat a správně identifikovat jeho příčinu, se kterou se bude vývojové oddělení následně zabývat.

Pro tuto činnost existuje v Keriu od roku 2013 aplikace, která umožňuje sběr chybových hlášení a jejich třídění kvalifikovanou osobou. Aplikace je v současné době zastaralá, protože například neumožňuje rychlé a intuitivní analyzování chybových hlášení nebo automatické vytváření e-mailové komunikace. Z tohoto důvodu bylo rozhodnuto o vývoji zcela nové aplikace samozřejmě s využitím všech zkušeností z aplikace původní.

Dopředu tedy byla poměrně jasná analýza domény a také existovaly poměrně přesné představy o funkčnosti budoucí aplikace. Základní požadavky na její funkčnost byly stanoveny takto:

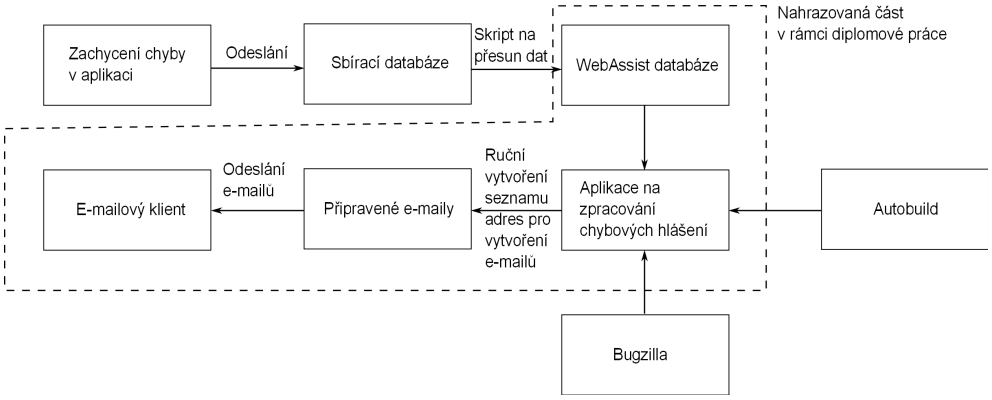
- automatizace části procesu,
- integrace s interními systémy,
- doplnění chybějící funkčnosti.

Cílem diplomové práce tedy bylo připravit webovou aplikaci, která bude maximálním způsobem usnadňovat analýzu a agregaci chybových hlášení poskytovaných uživateli, včetně zpětné komunikace s těmito uživateli. Výstupem aplikace pro vývojové týmy budou již jednoznačná neduplikovaná chybová hlášení.

2 Popis existující aplikace

V rámci této diplomové práce bude nahrazována aplikace na zpracování chybových hlášení produktů společnosti Kerio. Jedná se o komplexní systém, který pracuje se dvěma databázemi a systémy Autobuild a Bugzilla, jejichž napojení na původní aplikaci je podrobněji popsáno v kapitole 2.1.

Propojení celého systému existující aplikace na zpracování chybových hlášení je zobrazeno na obrázku 2.1, kde je také zvýrazněna část systému, která bude v rámci diplomové práce nahrazována. Podrobnější popis obrázku 2.1 a jeho jednotlivých částí je uveden v kapitolách 2.2 až 2.6.



Obrázek 2.1: Workflow původního řešení

2.1 Napojení na externí systémy

V této kapitole bude popsáno napojení externích systémů na původní aplikaci. Externími systémy, se kterými původní řešení aplikace na zpracování chybových hlášení pracuje, jsou Autobuild a Bugzilla.

2.1.1 Systém Autobuild

Ze systému Autobuild se získávají zdrojové soubory webové aplikace, které jsou komprimované ve formátu ZIP. Po stažení zdrojových kódů webové aplikace je potřeba komprimovaná data dekomprimovat. Dekomprimované zdro-

jové kódy se používají pro bližší analýzu reportované chyby (zobrazení call-stacku).

Zdrojové soubory produktů jsou dostupné přes restové[1] API systému Autobuild. Zdrojové soubory jsou získány pomocí http requestu na URL adresu, který může vypadat následovně:

`http://webbuild.kerio.local/webbuild/buildInfo.php?product=KMS&version=8&buildnum=2453&package=kerio-connect-unscrabmled-web`,

kde jednotlivé položky mají následující význam:

- `product` – produkt, jehož zdrojové soubory chceme získat,
- `version` – verze produktu,
- `buildnum` – pořadové číslo sestavení verze produktu,
- `package` – typ balíku souborů se zdrojovými kódy.

2.1.2 Systém Bugzilla

Původní aplikace na zpracování chybových hlášení je napojena na systém Bugzilla. Bugzilla je systém na sledování chyb při vývoji softwaru.

K chybovému hlášení může být přiřazeno unikátní číslo „změnového požadavku“[2] ze systému Bugzilla. V případě přiřazení je při prohlížení hlášení možné zobrazit v jakém stavu je zpracování chyb. Tato integrace pomáhá při zpracování nově zařazovaných chyb. Pokud se jedná o stejnou verzi produktu, je možné přiřadit číslo „změnového požadavku“. Pokud se chyba objevila v novější verzi než je oprava, je nutné ověřit, zda se nevyskytla v důsledku chybné opravy.

Hlavním důvodem zjišťování stavu „změnového požadavku“ je možnost kontroly, zda je již zavřený. Pokud byl „změnový požadavek“ již vyřešen

[1] REST (Representational State Transfer) – je styl klient-server architektury založený na bezestavosti a schopnosti použít cache (blíže viz [10]).

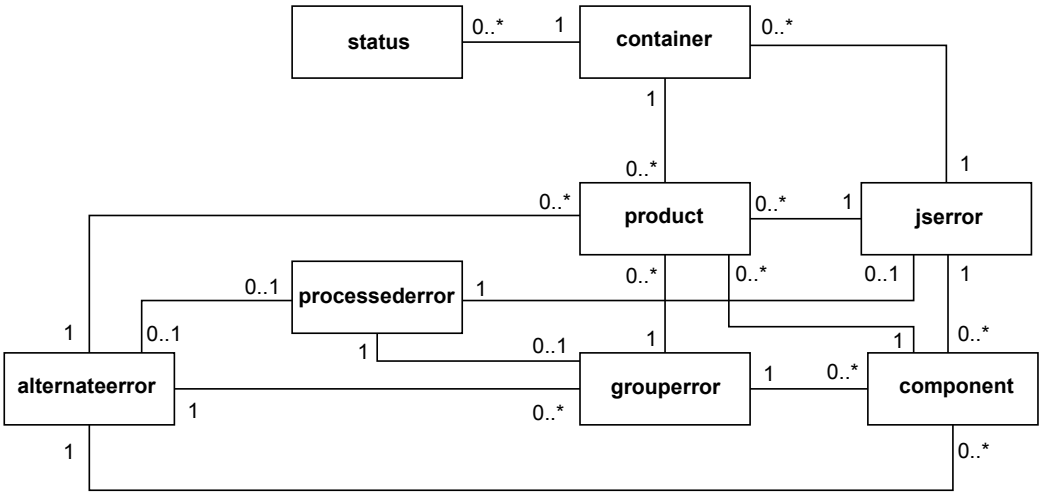
[2] Změnový požadavek (issue nebo bug) – záznam, pod kterým se eviduje požadavek na změnu či chyba v produktu.

(má stav zavřený), ale problém v aplikaci se stejně vyskytne (přijde chybové hlášení), tak je nutná důkladná analýza. Pokud se jedná o shodnou chybu je nutné tuto chybu znovu otevřít a vyřešit.

Pro usnadnění procesu znovuotevření „změnového požadavku“ se v aplikaci nachází hypertextový odkaz, který otevře webovou stránku požadovaného „změnového požadavku“ v systému na spravování vývoje softwaru. V současné době se již systém Bugzilla ve společnosti Kerio nepoužívá, byl nahrazen modernější alternativou – systémem Jira. Se systémem Jira však nebyla dosud provedena plná integrace a tak není možné automaticky zjišťovat stav chyb.

2.2 Struktura databáze

Původní databáze (analýza viz kapitola 2.7.1) se skládá z devíti tabulek, jejichž ER model je zobrazen na obrázku 2.2 a jejich struktura je popsána níže.



Obrázek 2.2: ER model původní databáze

Tabulka „jserror“

Tabulka „jserror“ obsahuje všechna chybová hlášení poskytnutá uživateli. Každé hlášení obsahuje informace o chybě a o aplikaci v době vzniku chyby,

jež slouží k odhalení příčiny vzniku. Tabulka „ jserror“ obsahuje následující sloupce:

- **pkey_jserror** – identifikátor položky tabulky (primární klíč),
- **fkey_product** – cizí klíč tabulky „product“,
- **fkey_component** – cizí klíč tabulky „component“,
- **fkey_container** – cizí klíč tabulky „container“,
- **userAgent** – text, který identifikuje internetový prohlížeč, poskytuje údaje o prohlížeči a operačním systému (jméno a verze prohlížeče, identifikátor operačního systému a jeho verze), například:
  `Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.2; Trident/4.0; .NET CLR 1.1.4322; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)`,
- **platform** – platforma, na které běžel server aplikace, například:
  `Win32`, `Mac OS X (10.9.4), x86_64`,
- **serverOs** – operační systém, na kterém běžel server, například:
  `Windows Server 2003`,
  `CentOS release 6.5 (Final) x86_64`,
- **language** – jazyk nastavený ve sledované aplikaci, například `en, ru, cs`,
- **resolution** – rozlišení obrazovky, například `1366x768`,
- **zoom** – hodnota přiblížení obrazovky, například `0, 1, 100`,
- **detectedBrowser** – detekovaný webový prohlížeč a jeho verze, například `Chrome 19`,
- **errorMessage** – chybová zpráva prohlížeče, například:
  `TypeError: head is undefined`,
- **url** – url skriptu, ve kterém nastala reportovaná chyba, například:
  `/webmail/webmail2.js`,
- **line** – číslo řádky skriptu, na které nastala reportovaná chyba,

- **callStack** – callstack je informace obsažená ve výjimce[3] obsahující sekvenci volaných funkcí až k řádce, která výjimku způsobila:

  ```
  kerio.dom.destroyNode@app.js:296
  .destroy@app.js:109588
  .destroy@extjs-4.1.3/ext.js:1910
  .destroy@extjs-4.1.3/ext.js:1909
  .destroy@app.js:109652
  .destroy@extjs-4.1.3/ext.js:1910
  ._clearFeed/<@app.js:109864
  ._clearFeed@app.js:109865
  ._changeFeed@app.js:109852
  .fire@extjs-4.1.3/ext.js:2465
  .continueFireEvent@app.js:492
  .fireEvent@app.js:473
  ._onClick@app.js:56452
  anonymous@app.js:46869
  Ext.EventManager.createListenerWrap/wrap@app.js:46825,
  ```

- **activeDialogId** – ID dialogu, který byl aktivní v době chyby, například:
  `_k_messageBox`, `waitingDialog`, `view [new window]`,

- **navigationItemId** – aktivní navigační prvek, například:
  `mail`, `calendar`,

- **clientDnsName** – doménové jméno klienta (detailní popis DNS viz [14]),

- **userComment** – komentář uživatele, který odesílal hlášení,

- **userEmail** – e-mailová adresa uživatele,

- **creationTime** – čas zápisu chyby do databáze,

- **descrambledMessage** – chybová zpráva prohlížeče, například:
  `Uncaught TypeError: Cannot read property 'dom' of null`,

- **buildNumber** – pořadové číslo sestavení produktu, například `2224`, `2225`,

- **windowSize** – velikost okna prohlížeče, například `1920x989`,

[3] Chyba aplikace vygeneruje callstack ve všech současných nejvíce používaných webových prohlížečích: Chrome, IE, Firefox, Opera

- **runTime** – doba od načtení webové aplikace do webového prohlížeče do vzniku chyby v sekundách,
- **developerToolActive** – informace o tom, zda byly aktivní nástroje vývojáře v prohlížeči (0 – nástroje vývojáře nebyly aktivní, nebo 1 – nástroje vývojáře byly aktivní),
- **browserSpeed** – detekovaná rychlost prohlížeče,
- **previousWidgetId** – identifikátor předchozího ovládacího prvku, například `mail`, `calendar`.

Tabulka „grouperror“

Tabulka „grouperror“ obsahuje jedinečné informace o chybových hlášeních. Na základě nového chybového hlášení se vytvoří skupina, do které se poté přiřazují všechna následující hlášení, která jsou shodná s vytvořenou skupinou. Tabulka „grouperror“ má následující strukturu:

- **pkey_group** – identifikátor položky tabulky (primární klíč),
- **fkey_product** – cizí klíč tabulky „product“,
- **fkey_component** – cizí klíč tabulky „component“,
- **detectedBrowser** – detekovaný webový prohlížeč,
- **url** – url skriptu, ve kterém nastala reportovaná chyba,
- **line** – číslo řádky skriptu, na které nastala reportovaná chyba,
- **descrambledMessage** – chybová hláška prohlížeče,
- **buildNumber** – pořadové číslo sestavení produktu,
- **bugList** – čísla chyb, která jsou přiřazena k dané skupině,
- **comment** – komentář skupiny,
- **callStack** – callstack je informace obsažená ve výjimce, obsahující sekvenci volaných funkcí až k řádce, která výjimku způsobila,
- **isRegex** – ohsahuje hodnotu 0, pokud je položka „descrambledMessage“ regulární výraz, jinak má hodnotu 1.

Tabulka „alternateerror“

Tabulka „alternateerror“ obsahuje alternativní skupiny chybových hlášení k některé hlavní skupině (položka tabulky „grouperror“) a je tvořena následujícími sloupci:

- **pkey_alternate** – identifikátor položky tabulky (primární klíč),
- **fkey_group** – cizí klíč tabulky „grouperror“,
- **fkey_product** – cizí klíč tabulky „product“,
- **fkey_component** – cizí klíč tabulky „component“,
- **detectedBrowser** – detekovaný webový prohlížeč,
- **url** – url skriptu, ve kterém nastala reportovaná chyba,
- **line** – číslo řádky skriptu, na které nastala reportovaná chyba,
- **descrambledMessage** – chybová hláška prohlížeče,
- **buildNumber** – pořadové číslo sestavení produktu,
- **callStack** – callstack je informace obsažená ve výjimce, obsahující sekvenci volaných funkcí až k řádce, která výjimku způsobila,
- **isRegex** – ohsahuje hodnotu 0, pokud je položka „descrambledMessage“ regulární výraz, jinak má hodnotu 1.

Tabulka „processederror“

Tabulka „processederror“ slouží jako spojovací tabulka mezi tabulkami „ jserror“ a „grouperror“. Položky tabulky „processederror“ definují přiřazení příchozího chybového hlášení k existující skupině, která definuje chybu. Tabulka „processederror“ obsahuje následující sloupce:

- **fkey_jserror** – cizí klíč tabulky „ jserror“,
- **fkey_group** – cizí klíč tabulky „grouperror“,
- **fkey_alternate** – cizí klíč tabulky „alternateerror“.

Tabulka „component“

Tabulka „component“ obsahuje moduly produktů ve spojení s jejich verzemi a má následující sloupce:

- **pkey_component** – identifikátor položky tabulky (primární klíč),
- **fkey_product** – cizí klíč tabulky „product“,
- **version** – verze produktu,
- **module** – název modulu produktu.

Tabulka „product“

Tabulka „product“ je pouhým seznamem produktů s následujícími sloupci:

- **pkey_product** – identifikátor položky tabulky (primární klíč),
- **name** – jméno produktu.

Tabulka „container“

Tabulka „container“ obsahuje doplňující informace k produktu, a to komentáře k chybovým hlášením, které zadal vývojář při analýze chyby a číslo chyby (získané zadáním chyby do trackovacího softwaru, například Jira, Bugzilla). Tato tabulka se již nepoužívá pro zpracování chybových hlášení. Tabulka „container“ obsahuje následující sloupce:

- **pkey_container** – identifikátor položky tabulky (primární klíč),
- **fkey_status** – cizí klíč tabulky „status“,
- **fkey_product** – cizí klíč tabulky „product“,
- **developerComment** – komentář k chybovým hlášením,
- **bug** – číslo chyby.

Tabulka „jserroroldproducts“

Tabulka „jserroroldproducts“ obsahuje chybová hlášení starých verzí produktů. Tato chybová hlášení se již nepoužívají pro stávající analýzu a rozpoznávání chyb. Struktura tabulky „jserroroldproducts“ je shodná se strukturou tabulky „jserror“.

Tabulka „status“

Tabulka „status“ je pouhým seznamem možných stavů, ve kterých se může chyba nacházet. Tato tabulka již není v analýze užívána.

- **pkey_status** – identifikátor položky tabulky (primární klíč),
- **name** – název stavu.

2.3 Odeslání chybových hlášení

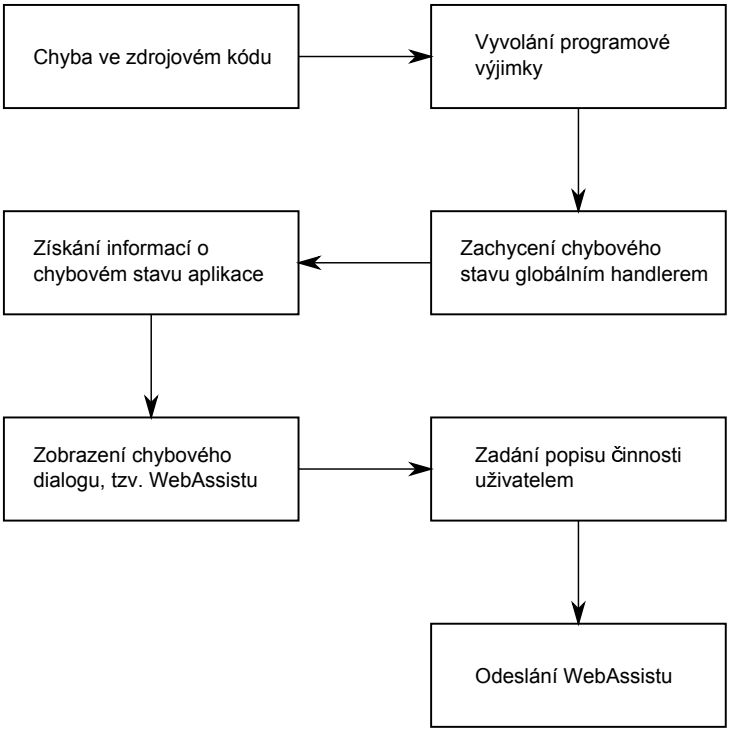
Popis zachycení chyby ve sledované aplikaci je popsán níže a zobrazen na obrázku 2.3.

Vše začíná chybou ve zdrojovém kódu aplikace. Při vzniku chyby v aplikaci je vyvolána výjimka, kterou lze zachytit globálním handlerem[4].

V globálním chybovém handleru jsou získána dostupná data o chybě a stavu aplikace, a je zobrazen chybový dialog, tzv. WebAssist (viz obrázek 2.4).

WebAssist kromě informací o chybě umožňuje uživateli zadat komentář, v rámci kterého uživatel může popsat co v aplikaci dělal a případně i další informace, které vedly k chybě.

[4]Globální handler zachytává pouze neošetřené výjimky. Globální handler je nutné definovat a v rámci něho je možné chybu zpracovat, například upozornit uživatele či odeslat chybové hlášení.



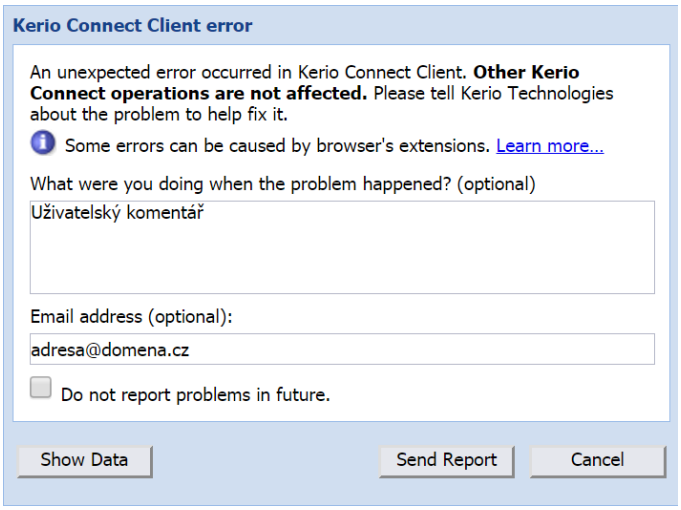
Obrázek 2.3: Popis zachycení WebAssistu ve sledované aplikaci

2.4 Automatické předzpracování chybových hlášení

Popis předzpracování chybových hlášení je popsán níže a zobrazen na obrázku 2.5.

Po odeslání chybového hlášení, se data hlášení uloží do tzv. sbírací databáze. Do sbírací databáze se ukládají všechna hlášení poslaná od uživatelů, všech produktů společnosti Kerio. Sbírací databáze neslouží pro zpracování chybových hlášení. Pro zpracování chybových hlášení slouží tzv. WebAssist databáze (viz obrázek 2.1).

Každý den je po půlnoci spuštěn automatický skript, který provede zkopírování reportovaných dat ze sbírací databáze do WebAssist databáze, se kterou pracuje původní aplikace na zpracování chybových hlášení.



Obrázek 2.4: Chybový dialog – WebAssist

2.5 Postup zpracování chybových hlášení

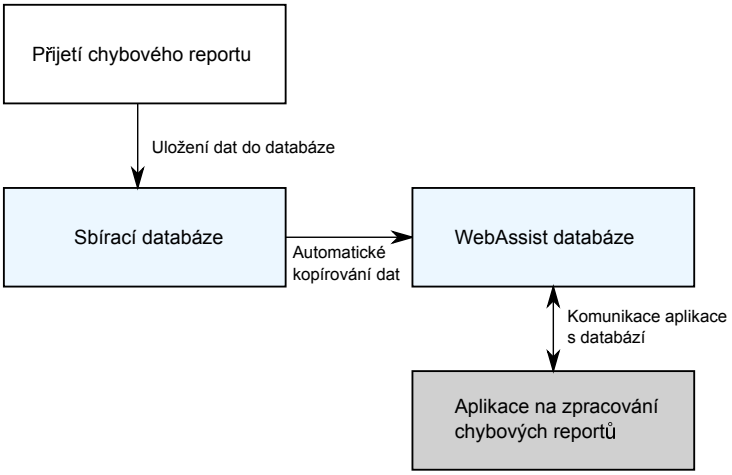
V této kapitole bude popsán postup zpracování chybových hlášení původního řešení včetně všech jeho předností a nedostatků. Celý proces zpracování je znázorněn na obrázku 2.6.

2.5.1 Přiřazení chybového hlášení do skupiny na základě shody

Při zpracování nových hlášení je třeba nejprve zjistit, zda se nejedná o další výskyt stejné chyby. Nejjednodušším způsobem je hledání **absolutní shody** s již existující skupinou chybových hlášení. Absolutní shodou se míní shoda dat hlášení v následujících parametrech: „product“, „modul“, „descrambledMessage“, „buildNumber“, „detectedBrowser“, „line“, „url“ a „callstack“.

Pokud není nalezena absolutní shoda, použije se shoda, kde data hlášení mohou obsahovat **rozdílnou verzi webového prohlížeče**. Zde se většinou jedná o stejnou chybu různých verzí stejného prohlížeče.

Následující stupeň je shoda, kde hlášení mohou obsahovat **jiný parametr „descrambledMessage“**. Důvodem tohoto typu shody jsou jazykové varianty chybových hlášek vygenerovaných webovými prohlížeči.



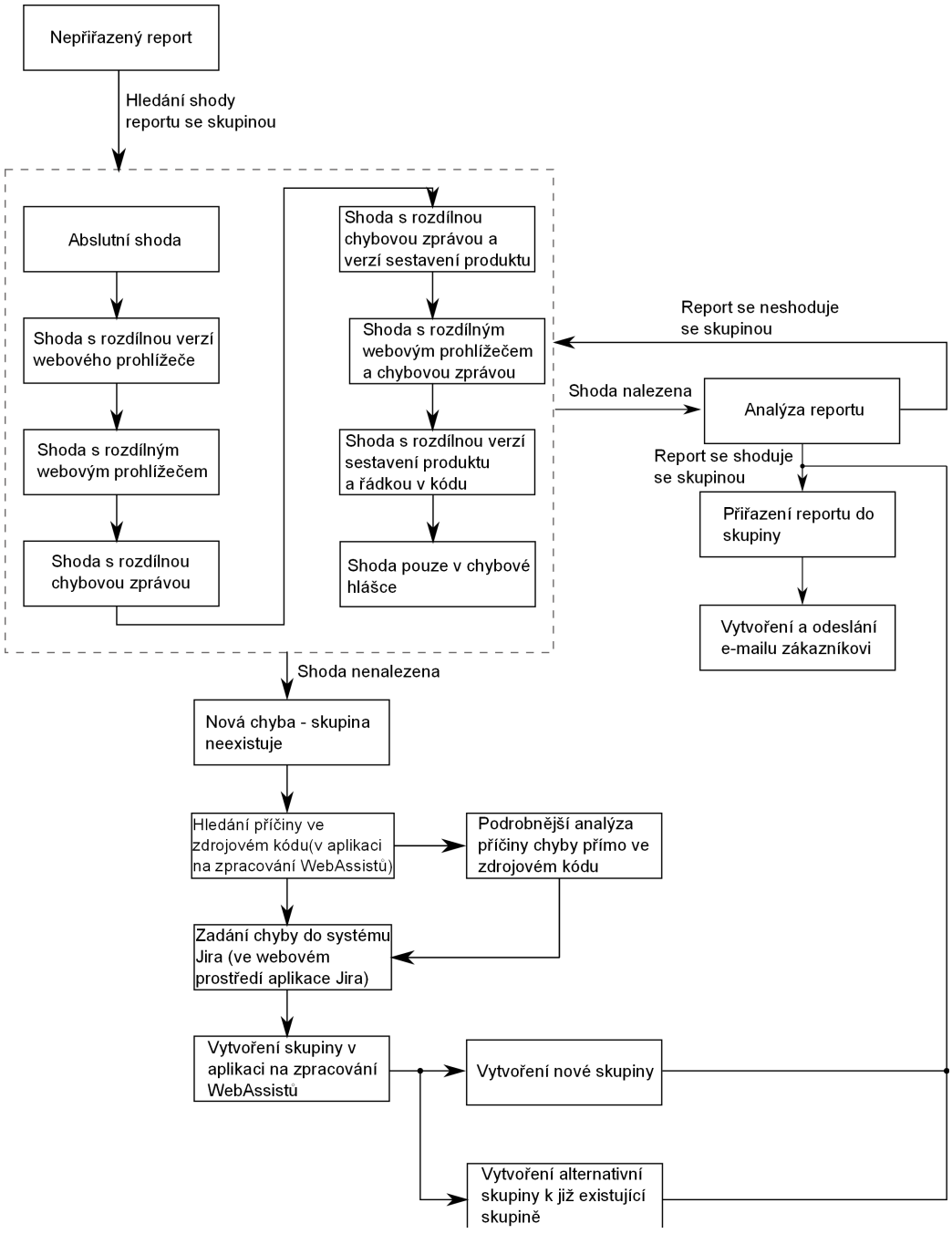
Obrázek 2.5: Předzpracování chybových hlášení

Dalším aplikovaným typem shody je shoda, kde data hlášení mohou obsahovat **rozdílnou hodnotu v parametru „detectedBrowser“**. Zde se podobně jako v předchozím případě většinou jedná pouze o stejnou chybu z různých verzí stejného prohlížeče.

Poté se aplikuje shoda, kde může být **rozdílná chybová zpráva webového prohlížeče a verze sestavení produktu** (tedy sloupce „descrambledMessage“ a „buildNumber“). Důvodem tohoto typu shody je, že v některé verzi produktu nemusela chyba nastat ve spojitosti s nastavenou lokalitou v produktu (například pro deset různých jazyků chyba nastala a ve dvou jazycích nenastala). V další verzi se však tato chyba ve spojitosti s daným jazykem objeví. Důvodů, proč chyba v některém jazyce nenastala, může být celá řada. Mezi nejjednodušší důvody patří například málo uživatelů, kteří danou jazykovou variantu produktu používají. Chyba může být také závislá na některé z jazykových variant produktu.

Následně aplikovanou shodou je shoda, kde může být **rozdílný webový prohlížeč a chybová zpráva** (tedy parametry „detectedBrowser“ a „descrambledMessage“). Rozdílné typy, verze a jazykové varianty webových prohlížečů mohou obsahovat různé chybové zprávy, které webový prohlížeč při chybě vygeneruje.

V další aplikované shodě se liší **verze sestavení produktu a řádka** ve zdrojovém kódu, kde chyba nastala (tedy rozdílné budou parametry „buildNumber“ a „line“). Tato možnost nastává tehdy, jedná-li se o stejnou chybu z různých verzí produktu. Čísla řádek, na nichž k chybě došlo, se mohou



Obrázek 2.6: Postup zpracování chybových hlášení

z důvodu oprav zdrojového kódu v jednotlivých verzích lišit.

Poslední aplikovaná a nejvíce benevolentní je **shoda pouze na základě stejné chybové zprávy** (tedy shoda v parametru „descrambledMessage“). Na základě tohoto typu shody již není snadné chybová hlášení do skupiny přiřadit. Tento typ shody spíše slouží pro možné objevení spojitosti mezi chybovými hlášeními, které spolu zdánlivě nijak nesouvisí.

Při nalezení absolutní shody se jedná o naprosto stejnou chybu, kterou není potřeba blíže analyzovat před jejím přiřazením do skupiny. Chybová hlášení s absolutní shodou je možné automaticky přiřadit všechny najednou, a to z důvodu, že již není potřeba další analýza. V ostatních typech shod je nutné provést analýzu zdrojových kódů (blíže viz kapitola 2.5.2).

Příklady všech různých typů shod chybových hlášení se skupinami jsou uvedeny v tabulce 2.1, přičemž pro každou shodu jsou uvedena data skupiny a data hlášení, která vyhovují dané shodě. Dále jsou v tabulce označena červenou barvou data, která se na základě dané shody mohou lišit. Pro lepší přehlednost nebyla v tabulce uvedena data callstacků, byla pouze uvedena nutnost jejich shody.

Typ shody	Chybová hláška	Číslo sestavení	Detekovaný prohlížeč	Řádka	URL	Callstack
Absolutní shoda	Error: Unexpected Content-Type: 200	2224	Safari 7	7150	/webmail/webmail2.js	...
	Error: Unexpected Content-Type: 200	2224	Safari 7	7150	/webmail/webmail2.js	...
Rozdílná verze detekovaného	RangeError: Maximum call stack size exceeded	3100	Chrome **37**	396	/webmail/lib/ext4/ext-all.js	...
prohlížeče	RangeError: Maximum call stack size exceeded	3100	Chrome **36**	396	/webmail/lib/ext4/ext-all.js	...
Rozdílná chybová hláška	Invalid procedure call or argument	2550	MSIE 11	5	/webmail/index2.js	...
	Argumento o llamada a procedimiento no válidos	2550	MSIE 11	5	/webmail/index2.js	...
Rozdílný detekovaný prohlížeč	TypeError: 'undefined' is not an object (evaluating 'c.isNode')	2224	Chrome 35	15094	/webmail/lib/ext4/ext-all.js	...
	TypeError: 'undefined' is not an object (evaluating 'c.isNode')	2224	Safari 7	15094	/webmail/lib/ext4/ext-all.js	...
Rozdílná chybová hláška a číslo	Invalid procedure call or argument	2550	MSIE 11	5	/webmail/index2.js	...
sestavení	Argumento o llamada a procedimiento no válidos	2985	MSIE 11	5	/webmail/index2.js	...
Rozdílný detekovaný prohlížeč	Invalid procedure call or argument	2550	MSIE 11	5	/webmail/index2.js	...
a chybová zpráva	Argumento o llamada a procedimiento no válidos	2550	MSIE 8	5	/webmail/index2.js	...
Rozdílné číslo sestavení a řádka	TypeError: this.win.query(...)[0] is undefined	2683	Firefox 30	53684	/webmail/webmail2.js	...
	TypeError: this.win.query(...)[0] is undefined	2817	Firefox 30	53678	/webmail/webmail2.js	...
Shodná chybová hláška	TypeError: input.charAt is not a function	2683	Firefox 31	13861	/webmail/webmail2.js	...
	TypeError: input.charAt is not a function	2843	Firefox 16	13854	/webmail/webmail2.js	...

Tabulka 2.1: Příklady shod chybových hlášení se skupinami

2.5.2 Analýza chybových hlášení před přiřazením do skupiny

Analýza chybových hlášení je manuální práce, která není nijak automatizovaná a musí se vykonávat ručně.

Analýza chybových hlášení se sestává z následujících kroků:

1. ověření, zda se skutečně jedná o chybu způsobenou produktem[5],
2. procházení zdrojového kódu v posloupnosti volání dle callstacku až k místu výskytu chyby,
   (a) většinou postačuje v každém bodě volání analyzovat jeho okolí – to umožňuje aplikace na zpracování hlášení, která si požadované zdrojové soubory získá ze systému Autobuild,
   (b) jinak je zapotřebí analyzovat body volání přímo ve zdrojovém kódu,
3. hledání kroků k reprodukci pro zjištění příčiny chyby a případně vytvoření „změnového požadavku“ v systému Jira.

2.5.3 Vytvoření nové skupiny chybových hlášení

Chybu, pro kterou nebyla nalezena žádná shoda (viz kapitola 2.5.1), je nutné nejprve analyzovat. Při analýze se hledá příčina chyby rozborem zdrojového kódu dle callstacku, který je součástí chybového hlášení. Následně je snaha chybu zreprodukovat.

Pokud se při analýze chyby nalezne příčina, je nejprve zadán „změnový požadavek“ do systému Bugzilla, který obsahuje informace zjištěné při analýze (kroky k reprodukci, poznámky ke zdrojovému kódu). Po založení „změnového požadavku“ je vytvořena nová skupina chybových hlášení. Při vytváření skupiny se do ní vloží také číslo souvisejícího bugu ze systému Bugzilla a slovní popis chyby (to pomáhá při zpracování dalších chybových hlášení).

[5]Současné webové prohlížeče jsou rozšiřitelné o tzv. pluginy, která mohou vložit kód do prostředí webové aplikace. Ve vloženém kódu může nastat chyba, která je pak odchycena globálním chybovým handlerem. Z toho důvodu se prochází callstack chyby, zda se skutečně jedná o chybu našeho produktu.

2.5.4 Vytvoření a odeslání e-mailu zákazníkovi

Zákazníci, kteří reportovali chybu a uvedli svůj e-mail, jsou informováni o tom, zda je již chyba opravena, popřípadě existuje-li způsob řešení chyby (například vypnutí pluginu webového prohlížeče). V případě, že nebylo možné chybu reprodukovat ani odhalit její příčinu, je zákazník kontaktován s žádostí o další informace.

V rámci zpracování chybových hlášení, tedy přiřazení chyb do skupin, jsou vytvářeny e-maily, které slouží jako počáteční krok pro e-mailovou komunikaci. Celý postup vytváření e-mailů je zobrazen na obrázku 2.7.

V době přípravy e-mailu pro zákazníka se nejprve zjišťuje, zda již nebyl zákazník ohledně stejné chyby v nedávné době (1-2 měsíce) kontaktován. Zjištění, zda zákazníkovi byl již e-mail odeslán, je zdlouhavá činnost, která vyžaduje prohledání odeslané pošty vývojáře, který e-maily vytváří.

Přednostně jsou kontaktováni zákazníci, kteří:

- často reportují chyby,
- uvedou komentář, který pomáhá při reprodukci,
- se angažují a poskytují další upřesňující informace, například kroky k reprodukci, podmínky vzniku chyby nebo uživatelská data, která chybu reprodukují.

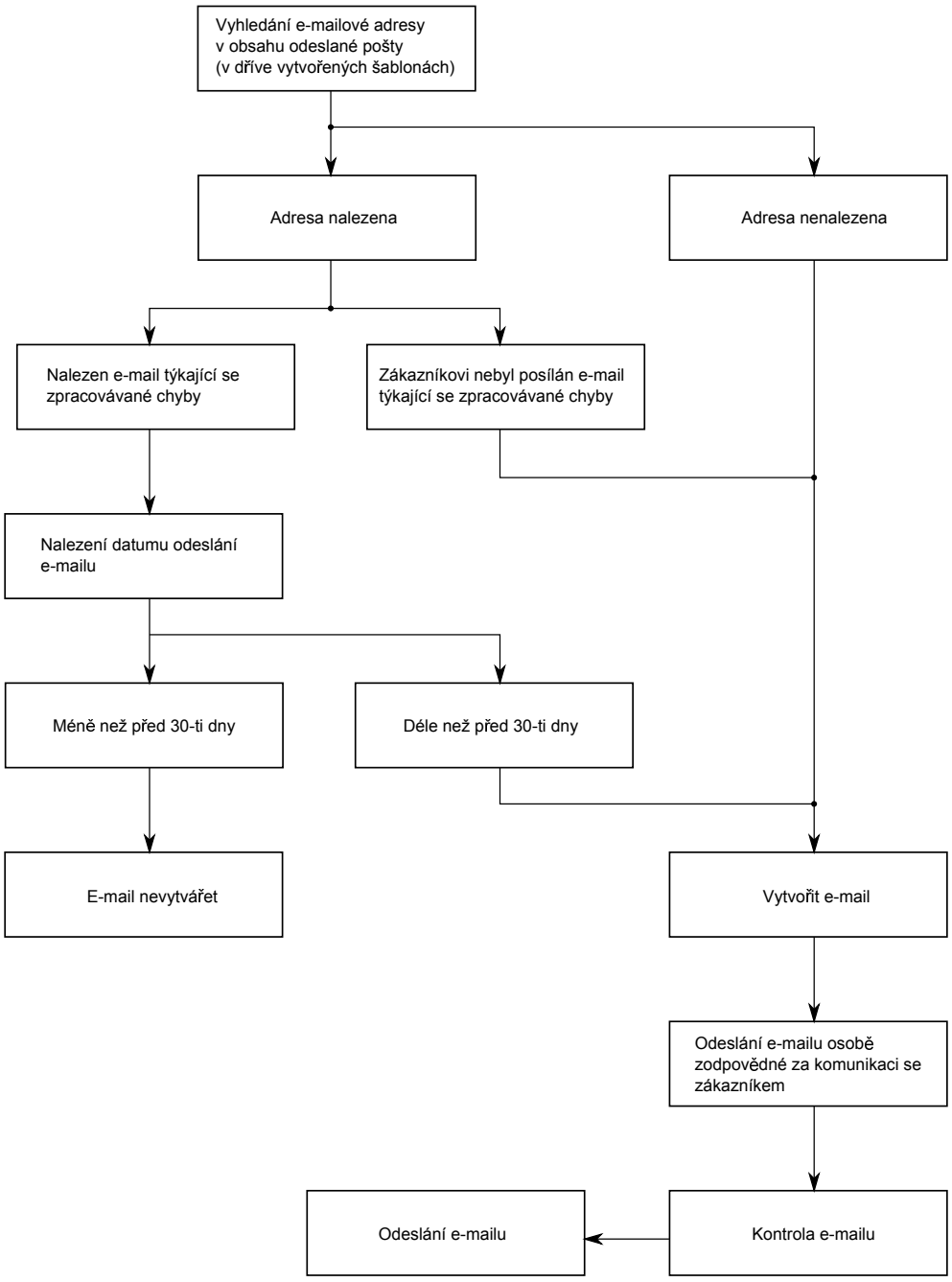
Po zpracování hlášení jsou připravené e-maily předány osobě zodpovědné za komunikaci se zákazníky, která zajistí jejich finální odevzdání.

2.6 Popis GUI aplikace

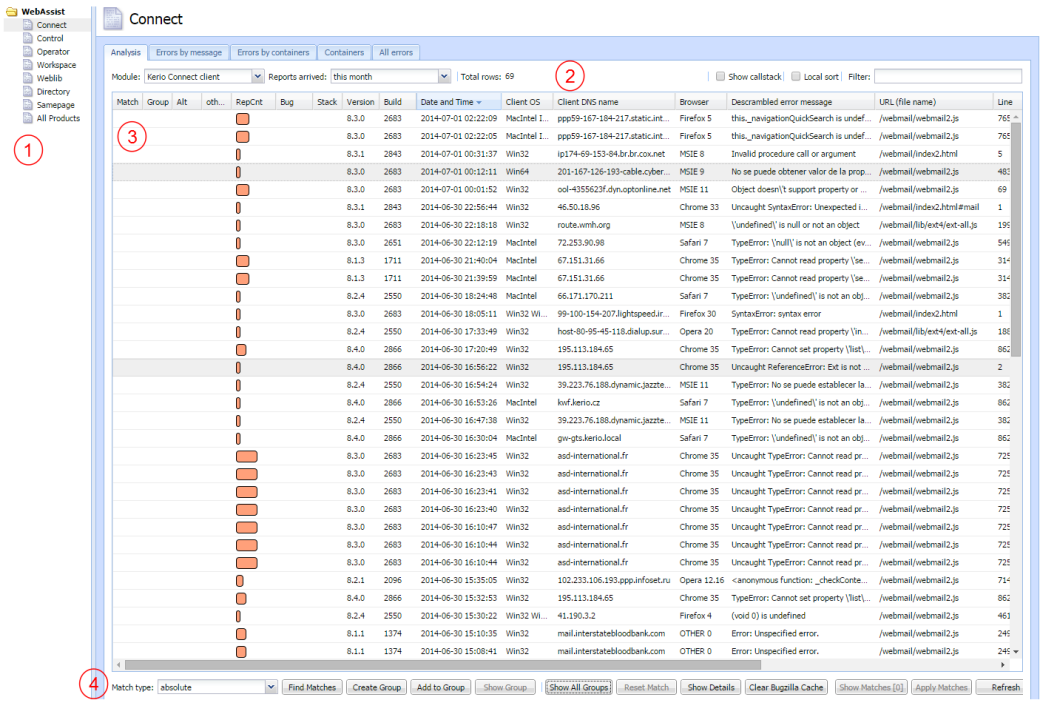
Na obrázku 2.8 je zobrazena hlavní stránka původní aplikace. V levé části lze zvolit produkt (označeno číslem 1), jehož chybová hlášení chceme analyzovat.

V horní části aplikace (označeno číslem 2) lze nalézt:

- **Module** – výběr části produktu (Kerio Connect Client, Administration,...),
- **Reports arrived** – období, ve kterém byla chybová hlášení doručena,



Obrázek 2.7: Workflow vytvoření e-mailu pro zákazníka



Obrázek 2.8: Hlavní stránka původní aplikace

- **Total rows** – celkový počet přijatých dosud nepřiřazených chybových hlášení za zvolené období,
- **Show callstack** – zobrazení dialogu s částmi zdrojových kódů call-stacku,
- **Local sort** – řazení pouze chybových hlášení na aktuální stránce, popřípadě řazení všech nalezených hlášení,
- **Filter** – filtr nad daty v tabulce chybových hlášení (zobrazí pouze data, která vyhovují filtru)

Po zvolení časového intervalu a modulu se načtou z WebAssist databáze chybová hlášení, která splňují vybraná kritéria. Tedy například na obrázku jsou zobrazena chybová hlášení produktu Kerio Connect, které spadají do modulu Kerio Connect Client a byly reportovány minulý měsíc. Chybová hlášení se zobrazí v části obrázku označené číslem 3.

Ve spodní části aplikace je možné řídit proces zpracování (označeno číslem 4) následujícími možnostmi:

- **Match type** – volba typu shody (viz kapitola 2.5.1),
- **Find Matches** – nalezení chybových hlášení splňujících typ shody s některou existující skupinou,
- **Create Group** – vytvoření nové skupiny,
- **Add to Group** – přidání chybových hlášení do skupiny,
- **Show Group** – zobrazení skupiny k danému chybového hlášení,
- **Show all Groups** – zobrazení všech existujících skupin,
- **Reset Match** – vrátí zobrazení do výchozího stavu,
- **Show Details** – zobrazí detaily chybového hlášení,
- **Clear Bugzilla Cache** – vymaže mezipaměť systému Bugzilla (při dalším načtení chybového hlášení se zjistí aktuální data v systému Bugzilla),
- **Show Matches** – v hlavní tabulce zobrazí pouze chybová hlášení, která splňují pravidla pro vybranou shodu,
- **Apply Matches** – přiřadí všechny nalezené absolutní shody chybových hlášení se skupinami do příslušných skupin,
- **Refresh** – znovu načte chybová hlášení podle vybraných kritérií (produkt, modul, časový interval).

2.6.1 Vytvoření skupiny

Při vytváření skupiny (viz obrázek 2.9) je možné přiřadit číslo bugu ze systému Bugzilla, který se k vytvářené skupině vztahuje. Dále je možné vložit chybovou hlášku, a to buď v textové formě nebo ve formě regulárního výrazu. Jako poslední lze zadat komentář vývojáře, který identifikuje či popisuje chybu.

Po vyplnění dat a potvrzení dialogu se vytvoří v databázi nová skupina, jejíž data budou reprezentovat chybu definovanou daty chybového hlášení, nad kterým bylo vytvoření skupiny iniciováno.

Obrázek 2.9: Dialog na vytvoření nové skupiny v původní aplikaci

2.6.2 Callstack prohlížeč

Obsahují-li data chybového hlášení callstack, tak je umožněno si prohlížet části zdrojového kódu pro jednotlivé řádky callstacku (viz obrázek 2.10). Pro každou řádku callstacku je zobrazeno sto řádků zdrojového kódu v okolí.

V levé části obrázku 2.10 je zobrazen chybový callstack, přičemž je každý řádek callstacku propojen se svým výřezem zdrojového kódu.

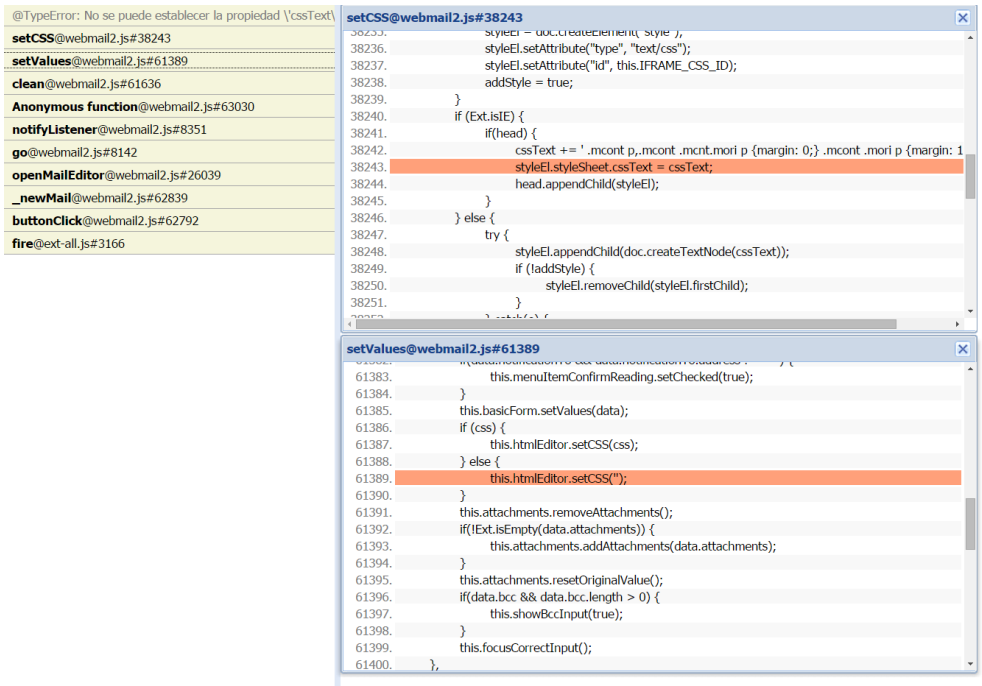
2.6.3 Přidání chybového hlášení do skupiny

Po nalezení shody chybového hlášení se skupinou je možné hlášení do skupiny přiřadit. Při přidání hlášení do skupiny je umožněno změnit číslo skupiny, které je předvyplněno.

Dále je umožněno změnit komentář skupiny, který byl vývojářem ke skupině přiřazen.

2.6.4 Zobrazení všech skupin

Dále je v původní aplikaci možné zobrazit všechny existující skupiny (viz obrázek 2.12) a informace o nich (včetně alternativních skupin k hlavním skupinám).

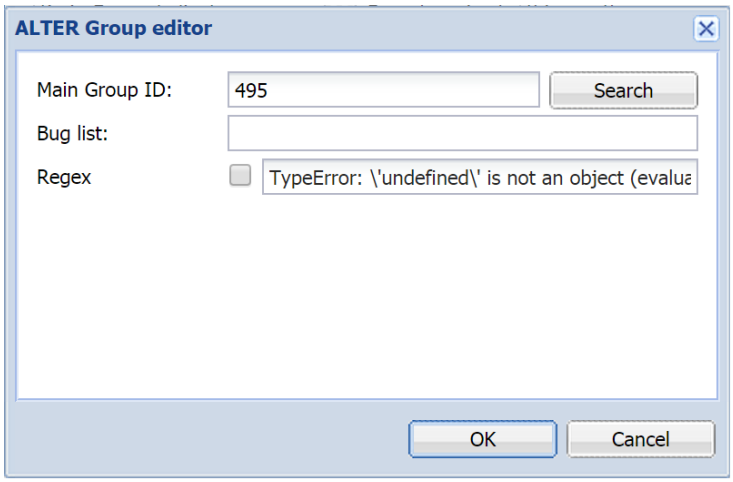


Obrázek 2.10: Prohlížeč callstacku v původní aplikaci

V levé části obrázku 2.12 jsou zobrazeny všechny skupiny. V pravé části jsou pak podrobnosti vybrané skupiny. V dolní části pravé strany je seznam s alternativními skupinami ke skupině vybrané vlevo. Dále je zde možné změnit informace o skupině, jako například chybovou hlášku, číslo bugu nebo komentář vývojáře. Pro uložení změn dat skupiny je v dialogu umístěno tlačítko „Save“.

2.7 Zjištěné nedostatky

V této kapitole budou popsány nedostatky původního řešení zjištěné v rámci analýzy. Hlavním cílem analýzy je zrychlení celého procesu zpracování chybových hlášení. To zahrnuje odhalování chyb a nedostatků od návrhu databáze až po zrychlení celého procesu zpracování.



Obrázek 2.11: Dialog na přiřazení chybového hlášení do skupiny v původní aplikaci

2.7.1 Databáze

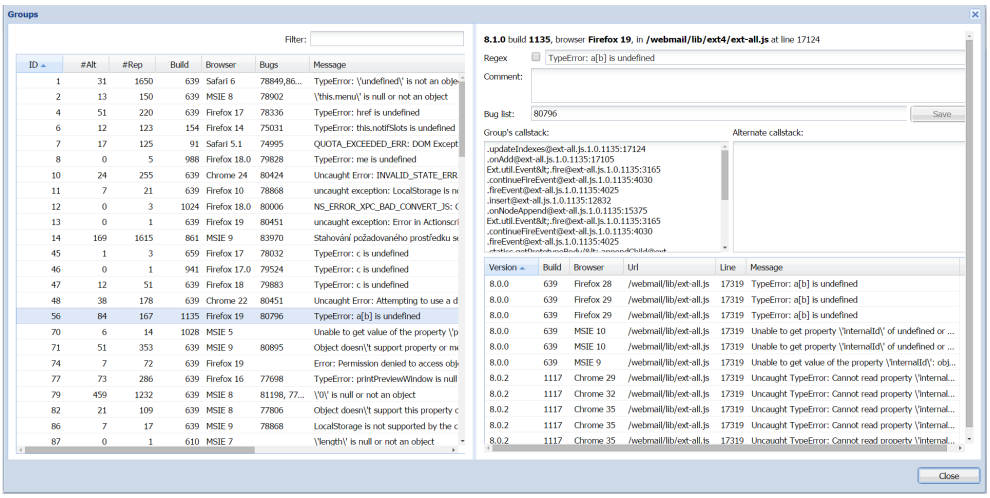
Struktura a jednotlivé tabulky databáze byly popsány v kapitole 2.2. Na základě seznámení se se strukturou databáze, byly odhaleny některé nedostatky.

První chybou návrhu jsou evidentně duplicitní data. Například tabulky „alternateerror“ a „grouperror“ obsahují stejné sloupce, které jsou i v tabulce „jserror“, přičemž nesou pouze drobnou přidanou hodnotu, a to identifikování skupiny chyb.

Pokud by se informace o skupině vložily do tabulky „jserror“, tak by bylo možné tabulky „alternateerror“ a „grouperror“ úplně odstranit. Po jejich odstranění by se stala nadbytečnou také tabulka „processederror“, která pouze propojuje záznam z tabulky „jserror“ se záznamem v tabulce „grouperror“, respektive v tabulce „alternateerror“.

Další nadbytečnou tabulkou je tabulka „container“. Tato tabulka obsahuje pouze komentář a číslo bugu zadané vývojářem, který prováděl zpracování. Obě tyto informace však také obsahuje tabulka „grouperror“.

Další nepotřebnou tabulkou je tabulka „status“, která obsahuje pouze seznam možných statusů, ve kterých se chyba může nacházet. Informace lze ovšem v případě potřeby získávat přímo z daného systému zpracování chyb



Obrázek 2.12: Dialog se zobrazením všech skupin v původní aplikaci

(v našem případě ze systému Jira). Pokud se tato informace udržuje v databázi, je ji stejně nutné získat ze systému Jira a databázi aktualizovat, pokud chceme stav udržet aktuální.

2.7.2 Vyhledávání shodných chybových hlášení

Postup nalezení shod v původní aplikaci je následující:

- získání všech nepřiřazených chybových hlášení z databáze,
- zjištění shody pro každé nalezené hlášení,
  - z načteného hlášení jsou získána data potřebná pro nalezení shody,
  - z dat je sestaven dotaz pro zjištění shody hlášení s některou skupinou, který je odeslán do databáze,
  - po získání výsledku je zkontrolováno, zda byla nalezena shoda,
  - pokud shoda nalezena nebyla, je sestaven další dotaz pro zjištění shody hlášení s alternativními skupinami, který je odeslán.

Nevýhodou tohoto postupu hledání shod je, že pro každé nepřiřazené chybové hlášení je nutné odeslat požadavek do databáze. Pokud navíc není žádná shoda nalezena, je nutné odeslat další požadavek. V nejhorším případě, je při hledání shod potřeba pro každé chybové hlášení odeslat dva požadavky do databáze.

2.7.3 Pracovní postup

V této části budou popsány nedostatky v pracovním postupu původní aplikace, mezi které patří především zdlouhavé vytváření e-mailů pro uživatele nebo nedostatky v zobrazování a přiřazování chybových hlášení do jednotlivých skupin.

Přiřazení chybových hlášení do skupin

Prvním nedostatkem původního pracovního postupu je přiřazování doručených chybových hlášení do skupin. Přiřazování chybových hlášení do skupin musí být provedeno po jednom hlášení. Pokud například přišla dvě chybová hlášení, která se shodují s některou existující skupinou na základně shody jiné než absolutní, je nutné tato hlášení do skupiny přiřadit po jednom, a to i když jsou navzájem naprosto shodná.

Nutnost přiřazovat chybová hlášení do skupin po jednom záznamu, vede ke zbytečné činnosti, která musí být vykonána při analýze hlášení, a to zejména jedná-li se o frekventovanou chybu.

Vytvoření e-mailu pro zákazníky

Před přiřazením chybového hlášení do skupiny se musí vytvořit e-mail, který slouží jako podklad pro komunikaci s koncovým zákazníkem, jenž chybu reportoval. Před vytvořením e-mailu (viz kapitola 2.5.4) je potřeba prohledat odeslanou poštu, zda již nebyl uživateli, který reportoval chybu, odeslán e-mail týkající se této chyby.

Pokud e-mail ještě odeslán nebyl nebo byl odeslán před delším časovým obdobím, e-mail se vytvoří. Všechny vytvořené e-maily se poté odešlou osobě zodpovědné za komunikaci se zákazníky, která e-maily zkontroluje a následně odešle koncovým zákazníkům.

Hlavní nevýhodou vytváření e-mailů je nutnost prohledávání dříve odeslané pošty a vyhodnocování, zda se e-mail bude odesílat. To je zdlouhavá činnost, která musí být provedena pro každé chybové hlášení, které je doručeno a je v něm uvedena e-mailová adresa.

Specifická hlášení prohlížečů

Jako další nedostatek mohou být považována rozdílná chybová hlášení (error message) napříč podporovanými webovými prohlížeči. Různé webové prohlížeče reagují na stejnou chybu vygenerováním rozdílné chybové hlášky, jak je zobrazeno v tabulce 2.2.

Detekovaný prohlížeč	**Chybová zpráva**
Chrome 35	TypeError: undefined is not a function
MSIE 8	Object doesn't support this property or method
Firefox 29	TypeError: input.charAt is not a function
Safari 6	TypeError: 'undefined' is not a function

Tabulka 2.2: Specifická hlášení webových prohlížečů vygenerovaná na základě stejné chyby

Rozdílnost chybových hlášek vede k vyšší pracnosti při zpracování chybových hlášení. To je způsobeno tím, že je nutné před přiřazením hlášení kontrolovat, zda se opravdu jedná o shodnou chybu, jelikož nedojde k absolutní shodě hlášení se skupinou (jestli už nebyla dříve vytvořena příslušná alternativní skupina pro danou variantu chybové hlášky).

Pomalé načítání chybových hlášení

Dalším nedostatkem je pomalé načítání chybových hlášení při volbě delšího časového období. Dotaz do databáze je totiž přes několik tabulek a nad velkým objemem dat. Nalezená chybová hlášení jsou zobrazena najednou až na konci zpracování dotazu.

Znovuotevření aplikace

Jako drobný nedostatek může být považována nutnost opakovat nastavení (produkt, modul, časový interval) po opětovném otevření webové aplikace na zpracování chybových hlášení. Není možné například vytvořit odkaz, který by zahrnoval konkrétní nastavení aplikace.

3 Analýza dostupných technologií

Dále zmiňované technologie jsou používány ve společnosti Kerio a byly stanoveny pro vývoj aplikace. Volba jiných technologií nebyla možná.

3.1 Node.js

Node.js je serverová platforma používající událostmi řízený neblokující model pro vstupně výstupní operace. Hlavní program v Node.js používá pouze jedno vlákno běžící v tzv. smyčce událostí[1] (event loop). V Node.js je smyčka událostí jazykový konstrukt, který je automaticky inicializován při spuštění platformy a veškerý aplikační kód běží v něm [13].

Za pomoci Node.js je možné obsloužit mnoho konkurenčních spojení, která se obsluhují na základě událostí, popřípadě „zpětných volání“ (tzv. callback). Uživatelé Node.js jsou odproštěni od řešení synchronizace vláken a nebezpečí deadlocku [13].

Při volání vstupně výstupní operace je vždy nutné předat metodu, která obdrží výsledek volání. Samotné volání tedy není blokující. O samotné vykonání těchto operací se stará nižší vrstva v jiném vlákně.

Jiným dnes častěji používaným řešením je tzv. konkurenční model, který je založen na vláknech operačního systému. Ovšem řešení vyžadující mnoho vstupně výstupních operací postavené na vláknech operačního systému je relativně neúčinné [6]. Použití vláken je považováno za obtížné a často vede k chybám [7].

3.1.1 MySql

Podporu MySql v prostředí Node.js zajišťuje modul nazvaný **node-mysql**. Pro použití ovladače **node-mysql** je nutné nejprve zapsat tzv. require. Poté je možné navázat spojení s mysql databází [19], například:

[1] Smyčka událostí (event loop) – je návrhový vzor pro vývoj tzv. reaktivních aplikací, které reagují na externí události (blíže viz [5])

```
var mysql = require('mysql');

var connection = mysql.createConnection({
    host: 'host',
    user: 'uzivatelske_jmeno',
    password: 'heslo',
    database: 'jmeno_databaze'
});
```

Po připojení k databázi je již možné definovat vlastní dotazy do databáze, například:

```
connection.query('SELECT * from jmeno_tabulky',
        function(err, rows, fields) {
    if (!err)
        console.log('Vysledek je: ', rows);
    else
        console.log('Chyba pri zpracovani.');
});
```

3.1.2 Nodemailer

Nodemailer je modul technologie Node.js pro posílání e-mailů používající protokol SMTP[2].

Mezi vlastnosti podporované modulem Nodemailer patří například [18]:

- podpora Unicode, tedy v mailu lze použít libovolné znaky,
- možnost vkládání HTML obsahu nebo prostého textu,
- možnost vkládání embedded obrázků do HTML těla e-mailu,
- SSL/STARTTLS pro bezpečné doručení e-mailu.

Příklad odeslání e-mailu modulem Nodemailer je uveden níže.

[2]Simple Mail Transfer Protocol (SMTP) – protokol určený pro přenos zpráv elektronické pošty. Protokol zajišťuje doručení pošty pomocí přímého spojení mezi odesílatelem a adresátem.

```
var nodemailer = require("nodemailer");

var smtpTransport = nodemailer.createTransport("SMTP",{
    service: "Gmail",
    auth: {
        user: "adresa_odesilatele",
        pass: "heslo"
    }
});

var mailOptions = {
    from: "adresa_odesilatele",
    to: "adresa_prijemce1, adresa_prijemce2", // seznam adres prijemcu
    subject: "predmet_e-mailu",
    text: "obsah_e-mailu",
    html: "html_obsah_e-mailu"
}

smtpTransport.sendMail(mailOptions, function(error, response){
    if(error){
        console.log('Chyba: ' + error);
    }
   else{
        console.log("Odeslana zprava: " + response.message);
    }
});
```

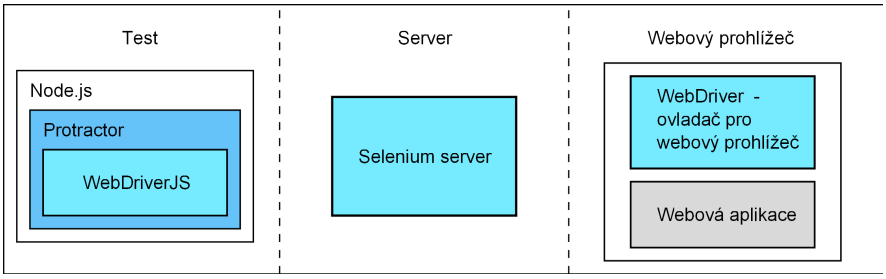
3.2 Protractor

Protractor je tzv. end-to-end testovací (viz kapitola 4.1.4) framework pro aplikace napsané v programovacím jazyce JavaScript. Protractor používá framework Selenium, což je framework pro testování webových aplikací [4].

Protractor pracuje ve spojení s frameworkem Selenium tak, aby poskytl automatické testovací prostředí, které simuluje interakci uživatele s aplikací spuštěnou ve webovém prohlížeči [4].

3.2.1 Meziprocesová komunikace

Testy používající Selenium WebDriver vyžadují tři procesy, a to testovací skript, server a prohlížeč (viz obrázek 3.1). Selenium server se stará o interpretaci příkazů testovacího skriptu a přeposílá je do jednoho nebo více prohlížečů. Komunikace mezi serverem a prohlížečem používá JSON WebDriver Wire Protocol (viz kapitola 3.2.3). Ve webovém prohlížeči je poté příkaz interpretován pomocí ovladače prohlížeče [20].



Obrázek 3.1: Architektura frameworku Protractor [20]

3.2.2 WebDriverJS

WebDriverJS API je založeno na tzv. promises[4], které jsou řízeny „control flow“ (řídící tok) a adaptovány pro testovací framework Jasmine [20].

WebDriverJS a tím i Protractor jsou zcela asynchronní. To znamená, že všechny funkce vrací promises. WebDriverJS udržuje frontu neukončených promises, která se nazývá „control flow“, aby bylo dodrženo pořadí vykonávání [20].

3.2.3 JSON WebDriver Wire Protocol

Všechny implementace WebDriverů, které komunikují s webovým prohlížečem nebo RemoteWebDriver serverem používají „wire protocol“. Wire proto-

[4]Promises – jsou objekty, které reprezentují hodnotu. Každá promise začíná v zahajovacím stavu a může být buď úspěšně vyřešena a vrátí hodnotu, nebo může být odmítnuta a vrací chybu.

col definuje RESTful[5] webové služby používající JSON přes HTTP. WebDriver Wire Protocol je implementován v modelu příkaz/odpověď (request/response) [20].

3.3 Ext Core

Ext Core je jádro knihovny ExtJS, které je dostupné pod MIT licencí. Ext Core funkce jsou navrženy tak, aby umožnily rychlý vývoj webových aplikací a současně podpořily dobrý návrh a škálovatelnost zdrojového kódu. Ext Core knihovna poskytuje AJAX (Asynchronous JavaScript and XML), události, animace, objektově orientované mechanismy a abstrakci pro procházení a manipulaci s DOM strukturou [1].

Ext Core má tyto přednosti:

- vysoký výkon,
- malou velikost knihovny,
- čistý, snadno udržitelný kód,
- intuitivní a snadno použitelné API,
- MIT open source licence.

3.4 MySQL

MySQL je multiplatformní databázový systém vytvořený švédskou firmou MySQL AB, která je v současnosti vlastněna společností Oracle Corporation [3].

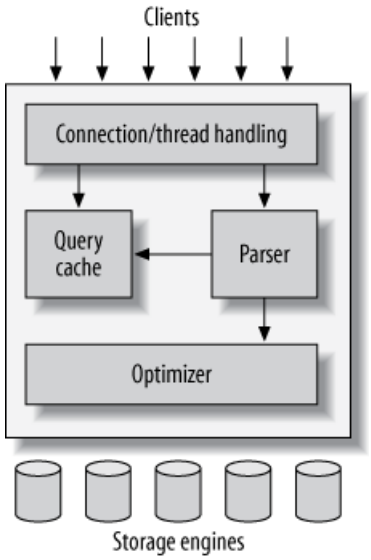
Komunikace s databází MySQL probíhá pomocí strukturovaného dotazovacího jazyka SQL (Structured Query Language). Pro svou snadnou implementovatelnost, výkon a především, že se jedná o volně šiřitelný software, má vysoký podíl v současnosti používaných databázích [3].

[5]REST (Representational State Transfer) – je styl klient-server architektury založený na bezestavosti a schopnosti použít cache (blíže viz [10]).

MySQL bylo již od počátku optimalizováno z hlediska výkonu, a to i za cenu, že ještě do nedávna nepodporovalo některé běžné techniky, jako například pohledy, triggery nebo uložené procedury. Tyto vlastnosti jsou doplňovány až v posledních letech, zejména z důvodu jejich zvýšené potřeby pro vývoj webových aplikací [3].

3.4.1 Architektura

Architektura MySQL databázového systému je znázorněna na obrázku 3.2. Nejvyšší vrstva obsahuje služby a nástroje, které jsou nejčastěji využívané v klient/server architekturách, jako například obsluha připojení, autentizace nebo bezpečnost [15].



Obrázek 3.2: Architektura MySQL databázového systému [15]

V druhé nejvyšší vrstvě se nachází samotná logika MySQL, která zahrnuje kód pro parsování dotazů, analýzu, optimalizaci, ukládání do mezipaměti (caching) a všechny vestavěné funkce (například datumy, časy nebo šifrování). Navíc tato vrstva poskytuje také funkce jako například uložené procedury, triggery nebo pohledy [15].

Poslední vrstva obsahuje tzv. storage engines. Ty jsou zodpovědné za ukládání a získávání všech dat uložených v MySQL databázi. Server s nimi komunikuje přes storage engine API, které odstiňuje odlišnosti mezi jednotlivými úložišti a tím je dělá transparentní z pohledu vyšších vrstev [15].

4 Možnosti testování softwaru

Softwarové testování je sada procesů zaměřených na zkoumání, hodnocení a zjišťování kvality počítačového softwaru. Testování softwaru ověřuje splnění technických, funkčních a uživatelských požadavků na softwarový produkt [2].

4.1 Typy testování

Testování lze rozdělit do šesti kategorií [12]:

- jednotkové testy,
- integrační testy,
- smoke testy,
- end-to-end testování,
- systémové testy,
- uživatelské akceptační testy.

Jednotlivé kategorie testů provádí různí lidé a systém je na každém stupni testován z vždy jiného úhlu pohledu, s jiným cílem, jiným způsobem a s jinými vyplývajícími důsledky.

4.1.1 Jednotkové testy

Jednotkové testy (nebo také unit testy) ověřují funkčnost a korektnost implementovaného systému. Pod pojem jednotkového testování se zahrnují nástroje, metodiky a činnosti, jejichž cílem je ověřování správné funkčnosti dílčích částí neboli jednotek (metody, třídy) zdrojového kódu. Zajímají nás především správné návratové hodnoty metod v závislosti na vstupních parametrech. Při jednotkovém testování bychom se měli snažit o maximální izolaci testované jednotky. Jednotkové testy jsou obvykle psány vývojářem, který daný programový kód psal [9].

4.1.2 Integrační testy

Úkolem integračních testů je ověřit, že jednotlivé části aplikace je možné sestavit do funkčního celku. Jde tedy o test integrace jednotlivých komponent. Obvykle jsou tyto testy psány vývojáři. U rozsáhlejších systémů, případně u složitější struktury procesu vývoje, bývá určen konkrétní pracovník, který je primárně odpovědný právě za sestavení aplikace a integrační testy [9].

4.1.3 Smoke testy

Smoke testy jsou sadou testovacích scénářů, které jsou spuštěny před zahájením systémového testování. Cílem smoke testů je ověřit, zda je aplikace nainstalovaná, spuštěná, přístupná a nakonfigurovaná pro potřeby testů. Pokud smoke testy selžou, tak je další testování aplikace pozastaveno a daná verze sestavení aplikace je odmítnuta [9].

4.1.4 End-to-end testování

End-to-end testování je komplexní testovací proces obsahující testování aplikace pomocí reálných uživatelských případů užití. End-to-end testování tedy ověřuje správnou funkčnost celého systému z pohledu uživatele.

Jako příklad end-to-end testování může být uveden následující scénář testu (e-mailová aplikace):

- přihlášení do aplikace,
- přístup do e-mailové schránky,
- vytvoření e-mailu,
- odeslání e-mailu,
- kontrola odeslané pošty
- odhlášení z aplikace.

4.1.5 Systémové testy

Základní metodikou systémového testování softwaru je FURPS[1]. Metoda FURPS byla vytvořena společností Hewlett-Packard na základě potřeby definovat, jak poznat a ověřit kvalitu dodávaného softwaru. Metoda FURPS nahlíží na testování kvality informačních systémů z pěti hledisek, a to funkcionality, použitelnosti, spolehlivosti, výkonu a podporovatelnosti [12] a [9].

Funkční testování

Funkční testování (někdy také black-box testování) je technika testování, která se zaměřuje na funkční požadavky softwaru a ověřuje, že software obsahuje všechny požadované funkcionality. Funkční testování také zahrnuje ověření funkčnosti systému na různých platformách a integraci s externími aplikacemi. Cílem je simulace prostředí koncových uživatelů [11].

Testování použitelnosti

Uživatelské testování je jednou z nejdůležitějších a nejčastěji používaných metod při testování použitelnosti. Princip spočívá v tom, že tato metoda simuluje chování samotných uživatelů, čímž lze odhalit chyby, které zůstaly vývojářům skryté. Toto testování je vhodné aplikovat již při vývoji softwaru, což vývojářům poskytuje cenné informace, se kterými snáze eliminují další problémy. Uživatelské testování tak může určovat správný směr při vývoji samotné aplikace a předcházet tak vznikajícím problémům [8].

Testování spolehlivosti

Cílem testování spolehlivosti je určit zda software splňuje požadavky na spolehlivost, které byly definovány zákazníkem. Základní typy testů spolehlivosti jsou [12]:

- **testy robustnosti systému** – testy odolnosti proti selhání po datové i funkční stránce,

[1]FURPS – Functionality, Usability, Reliability, Performance, Supportability

- **strukturní testy** – typicky testy webových rozhraní (chybné odkazy, chybějící stránky, osamocené stránky, ...),
- **stress testy** – ověření stability a funkčnosti systému při nestandardních podmínkách (snížení operační paměti aplikačního serveru, omezené sdílení HW i SW prostředků).

Výkonnostní testování

Při testování výkonnosti je ověřováno, zda bude software na podporovaném hardwaru fungovat optimálně i při vysoké zátěži (přístup hodně uživatelů současně, velké množství dat, atd.). Databáze může fungovat dokonale s malým počtem dat a dotazů, ale je-li vystavena vysoké zátěži běžného provozu, mohou se vyskytnout výkonnostní problémy.

Mezi základní typy výkonnostního testování patří [12]:

- **výkonnostní profilování** – měření odezvy systému, nalézání výkonnostně slabých míst systému (tzv. „úzká hrdla“),
- **testování konkurenčního přístupu ke zdrojům** – testy zachování funkčnosti systému při přístupu více uživatelů (popř. jiných systémů) ke sdíleným hardwarovým i softwarovým prostředkům,
- **zátěžové testování** – testy chování a výkonu systému pod vysokou zátěží (vysoký počet současně pracujících uživatelů),
- **srovnávací testy** – porovnání výkonnosti systémů na referenční platformě.

Testování podporovatelnosti

Testování podporovatelnosti ověřuje kompatibilitu s cílovými hardwarovými a softwarovými konfiguracemi a další vlastnosti související s údržbou systému. Mezi základní typy testování podporovatelnosti patří [12]:

- **testování konfigurací** – ověřuje, zda systém bude fungovat i na jiných přípustných hardwarových/softwarových konfiguracích, než jaké jsou nastaveny v současném testovacím (popř. produkčním) prostředí,

- **testování instalací** – test kompletní instalace a updatů na různé hardwarové a softwarové konfigurace.

4.1.6 Akceptační testy

Akceptační testy jsou poslední fází procesu testování softwaru. Cílem akceptačního testu je ověření úplnosti a funkčnosti informačního systému na základě akceptačních kritérií. Na software je nahlíženo jako na černou skříňku s cílem zjistit zda odpovídá specifikaci a neobsahuje kritické chyby. Podrobněji viz literatura [2].

5 Analýza řešení

V této kapitole bude popsána analýza aplikace implementované v rámci diplomové práce. V rámci analýzy bylo navrženo řešení nalezených nedostatků původní aplikace na zpracování chybových hlášení, které byly popsány v kapitole 2.7. Mezi hlavní nedostatky patří:

- **složité přiřazování chybových hlášení do skupin**,
  - přiřazování do skupin po jedné chybové hlášce,
  - nutnost provádět analýzu z důvodu specifických hlášení generovaných různými webovými prohlížeči,
- **vytváření e-mailů pro zákazníky**,
  - prohledávání dříve odeslané pošty a vyhodnocování, zda e-mail odesílat,
  - předávání vytvořených e-mailů osobě, která provádí samotnou komunikaci se zákazníkem,
- **znovuotevírání aplikace**,
  - nutnost opakovat nastavení aplikace po opětovném otevření aplikace,
- **návrh databáze**,
  - řada duplicitních hodnot v databázi,
  - pomalé načítání chybových hlášení (zobrazování všech výsledků až na konci zpracování dotazu).

5.1 Složité přiřazování chybových hlášení do skupin

Mezi nedostatky původní aplikace na zpracování chybových hlášení patří složité přiřazování hlášení do skupin. To je způsobeno nemožností přiřadit více hlášení do jedné skupiny najednou. Pokud chybová hlášení o jednom typu

chyby nepřišla ze stejného webového prohlížeče, je přiřazování hlášení ztíženo o jejich nutnou analýzu. Různé webové prohlížeče totiž generují rozdílné chybové hlášky. Stejný problém s chybovými hláškami prohlížečů také nastane, pokud byl v prohlížeči nastaven jiný jazyk (chybová hláška je generována v nastaveném jazyce).

5.1.1 Přidání více chybových hlášení najednou

V původní aplikaci lze přiřazovat chybová hlášení do skupin pouze po jednom. Přidání více hlášení najednou je umožněno pouze v případě absolutní shody, kdy lze přiřadit všechny nalezené absolutní shody, avšak bez možnosti jakéhokoliv zásahu do procesu přiřazení.

U ostatních typů shod již přiřazení více chybových hlášení najednou možné není, a to i když se například jedná o několik naprosto shodných hlášení, která však nesplňují absolutní shodu se skupinou. To způsobuje, že je nutné vykonávat zbytečnou činnost, kterou by bylo možné automatizovat.

Jinou možností, jak v původní aplikaci přiřadit více chybových hlášení najednou (hlášení nesplňují absolutní shodu, ale jsou navzájem naprosto shodná), je přiřazení prvního hlášení do skupiny, čímž se vytvoří alternativní skupina. Poté je potřeba znovu vyhledat absolutní shody. Tuto shodu již splní i ostatní hlášení, která jsou shodná s nově vytvořenou alternativní skupinou a lze je tedy přiřadit v rámci zpracování všech absolutních shod.

V analýze nového řešení bude navržena možnost přiřazení všech hlášení, která se shodují se stejnou skupinou na základě jiné shody než absolutní. To umožní zrychlení procesu přiřazování do skupin a omezí nutnost opětovného hledání na základě shody.

5.1.2 Specifika hlášení webových prohlížečů

Mezi nedostatky původní aplikace patří specifická hlášení webových prohlížečů. Pokud v aplikaci nastane chyba, tak je její chybová zpráva závislá na prohlížeči, ve kterém je aplikace spuštěna. Při zpracování chybových hlášení je nutné každou chybu analyzovat, zda je chybová zpráva prohlížeče rozdílná pouze z důvodu jiného typu prohlížeče, nebo zda se jedná o jiný typ chyby.

Snaha tedy byla navrhnout takové řešení, aby zpracování chybových hlášení nebylo závislé na webových prohlížečích a jejich specifických chybových zprávách.

Řešením tohoto problému může být zavedení tzv. unifikovaných zpráv, které budou reprezentovat chyby napříč webovými prohlížeči. Do databáze tedy bude přidána tabulka, kde budou uvedeny všechny verze chybových zpráv s informací o prohlížeči, ke kterému chyba patří. K takto identifikované dvojici (detekovaný prohlížeč – chybová zpráva) bude přiřazena unifikovaná zpráva, která bude popisovat chybu nezávisle na webovém prohlížeči, jak je například vidět v tabulce 5.1.

Detekovaný prohlížeč	**Chybový zpráva**	**Unifikovaná zpráva**
Chrome 35	TypeError: undefined is not a function	undefined is not a function
MSIE 8	Object doesn't support this property or method	undefined is not a function
Firefox 29	TypeError: input.charAt is not a function	undefined is not a function
Safari 6	TypeError: 'undefined' is not a function	undefined is not a function

Tabulka 5.1: Definování unifikované zprávy

Takto definovaná unifikovaná chybová zpráva by měla mít za důsledek přiřazení více chybových hlášení do skupin na základě přesnějších pravidel shod, a to hlavně na základě absolutní shody.

5.2 Vytváření e-mailů pro zákazníky

Velkou nevýhodou současného procesu zpracování chybových hlášení je náročnost ručního vytváření e-mailové komunikace. Pro každou chybovou hlášku, která obsahuje e-mailovou adresu zákazníka, je nutné vytvořit e-mail a poté jej odeslat.

Před vytvořením e-mailu se pro každé chybové hlášení musí ručně pro-

hledat složka odeslaných e-mailů a zjistit, zda se e-mail se stejnou chybou již neposílal. Pokud e-mail nalezen nebyl, je vytvořen a předán osobě komunikující se zákazníky.

Řešením současného stavu je navrhnutí automatického vytváření e-mailů při procesu zpracování hlášení. Při každém přiřazení hlášení do skupiny by mělo být možné vytvořit e-mail pro zákazníka, popřípadě přiřadit již existující e-mail z databáze. Dále bude aplikace umožňovat takto předvytvořené e-maily procházet, upravovat a přímo z aplikace odeslat.

Úpravu a odeslání e-mailů bude moci provádět osoba zodpovědná za komunikaci se zákazníky přímo v aplikaci, čímž odpadne nutnost předávání e-mailů mezi několika zaměstnaci.

Odesílání e-mailů bude provedeno přes API Kerio Connect, celá komunikace tak bude dostupná v e-mailové schránce v sekci odeslané pošty. Další komunikaci se zákazníkem bude možné realizovat z libovolného e-mailového klienta.

5.3 Znovuotevření aplikace

Drobným nedostatkem původní aplikace je nutnost opakovat nastavení aplikace po jejím opětovném otevření.

Jako řešení bude navrhnut způsob uložení aktuálního nastavení aplikace. Nastavení aplikace bude definováno v URI, a to v části nazvané fragment. Fragment je část URI, která je definovaná za hash znakem (znak #, detailní popis viz [21]).

5.4 Optimalizace databáze

Načítání chybových hlášení z databáze je časově náročné. Pro větší množství chybových hlášení (řádově stovky) se čas pohybuje v řádech desítek sekund. Pomalé vykonání dotazu je způsobeno strukturou databáze, jelikož je nutné použít složité dotazy přes více tabulek. V rámci optimalizace databáze byla navrhnuta nová struktura databáze, která by měla redukovat nutnost dotazů nad větším počtem tabulek.

5.4.1 Návrh nové databáze

V původním návrhu databáze bylo nutné pro vyhledání nezpracovaných chybových hlášení vykonávat dotaz nad větším počtem tabulek a velkým obsahem dat, obvzláště, bylo-li zvoleno delší časové období pro vyhledávání. Dotaz pro získání všech nepřiřazených hlášení pracoval celkem nad čtyřmi tabulkami, a to tabulkou se všemi chybovými hlášeními, tabulkou obsahující seznam všech přiřazených chybových hlášení, tabulkou produktů a tabulkou komponent, která obsahuje názvy modulů produktů s jejich verzemi sestavení.

Nová databáze bude navržena tak, aby bylo možné vyhledat chybová hlášení pouze nad dvěma tabulkami, a to tabulkou obsahující chybová hlášení a tabulkou komponent, která obsahuje názvy modulů produktů s jejich verzemi sestavení.

Další nevýhodou původní databázové struktury bylo, že pro každou alternativu chybového hlášení, která definovala stejnou chybu, se vytvořila nová skupina. Po první identifikaci chyby se založila skupina definující daný problém. Při každém dalším výskytu chyby neobsahující absolutně shodná data (například se lišil webový prohlížeč), byla vytvořena nová alternativní skupina, která se uložila do jiné tabulky databáze než skupina původní. Každá skupina byla definována daty zkopírovanými z chybové hlášky, což způsobovalo duplicitu dat. Po vytvoření skupiny se do přiřazené chybové hlášky vložila informace o skupině klasifikující hlášku.

Při hledání shody dalšího výskytu chybového hlášení se porovnávalo hlášení s daty uloženými v tabulkách se skupinami a alternativními skupinami. V nové struktuře databáze bude každá skupina identifikována přímo chybovým hlášením.

V nové struktuře databáze se odstraní tabulka s alternativními skupinami, která již nebude zapotřebí. Tabulka se skupinami bude obsahovat pouze informace specifické pro danou skupinu, jako je například komentář ke skupině nebo číslo „změnového požadavku“ ze systému Jira.

Po přiřazení chybového hlášení do skupiny se do dat chybového hlášení vloží číslo skupiny a informace o tom, zda se jedná o takzvanou unikátní chybovou hlášku. Tato informace odstraní nutnost vyhledávat nad větším množstvím dat než v původní databázi a zároveň umožní vyhledávat pouze nad jednou tabulkou.

O unikátní chybovou hlášku se bude jednat, jestliže se jedná o chybovou hlášku:

- na základě které byla vytvořena skupina (první výskyt chyby),
- která definuje stejnou chybu, ale není absolutně shodná s původní chybovou hláškou (nahrazení alternativní skupiny).

5.5 Dlouhé čekání při náročném dotazu

I přes optimalizaci struktury databáze zůstávají některé dotazy časově náročné, což je uživatelsky nepřívětivé. Výsledek je zobrazen až po zpracování dotazu a jeho vykonávání není možné přerušit.

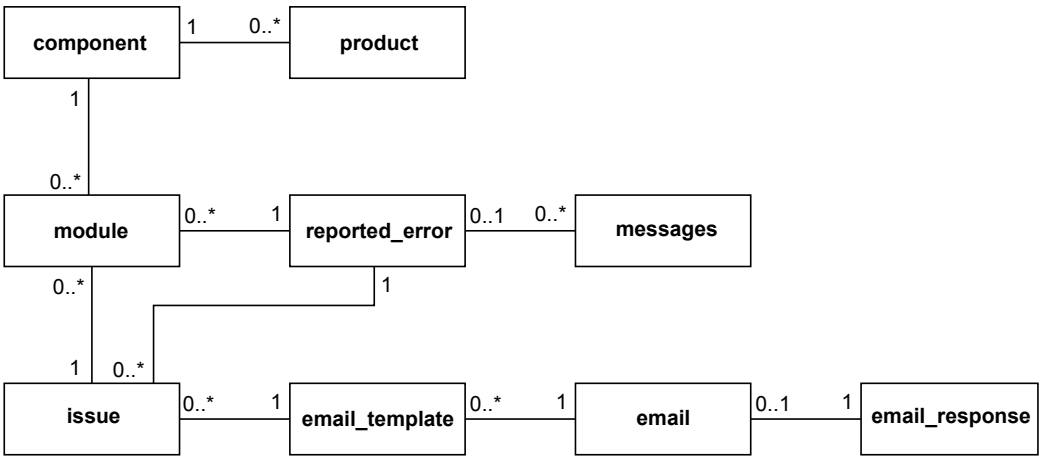
Řešením tohoto problému je rozdělení dotazu na více částí, na které se pak bude možné dotazovat z klienta. Tímto způsobem lze zobrazit první část výsledku a další části načítat postupně dle potřeby až do zaplnění obrazovky. Další data (mimo již zaplněnou obrazovku) je možné začít načítat až v případě potřeby.

Výhodou postupu vrácení výsledků po částech je, že po návratu první dávky dat z databáze již lze s daty v aplikaci pracovat. Na ostatní data není nutné čekat (dotaz je vykonáván na pozadí aplikace). Tím je docíleno rychlé odezvy na uživatelský příkaz (například nalezení všech absolutních shod). Vrácení výsledků po částech také umožní přerušit vyhledávání, nebo jej změnit před dokončením.

Nevýhodou výše zmíněného postupu je, že ihned neposkytuje uživateli informaci o celkovém počtu výsledků.

5.6 Struktura databáze

Kapitola popisuje strukturu databáze navrženou v rámci analýzy. Oproti původní databázi, která obsahovala devět tabulek, došlo k redukci na pět tabulek. Ovšem přibyly tři tabulky pro e-mailovou komunikaci, kterou původní aplikace neobsahovala. Dále přibyla jedna tabulka, která je určena pro unifikované zprávy (viz kapitola 5.1.2). ER model databáze je zobrazen na obrázku 5.1 a struktura všech tabulek je popsána níže.



Obrázek 5.1: ER model nové databáze

Tabulka „reported_error“

Tabulka „reported_error“ obsahuje všechna příchozí chybová hlášení nesoucí informace o chybě, která v aplikaci nastala. Tabulka „reported_error“ se skládá z následujících sloupců:

- **pkey_reported_error** – identifikátor položky tabulky (primární klíč),
- **fkey_product** – cizí klíč tabulky „product“,
- **fkey_component** – cizí klíč tabulky „component“,
- **fkey_container** – cizí klíč tabulky „container“,
- **userAgent** – text, který identifikuje internetový prohlížeč, poskytuje údaje o prohlížeči a operačním systému (jméno a verze prohlížeče, identifikátor operačního systému a jeho verze), například:
  `Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.2; Trident/4.0; .NET CLR 1.1.4322; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)`,
- **platform** – platforma, na které běžel server aplikace, například:
  `Win32`, `Mac OS X (10.9.4)`, `x86_64`,
- **serverOs** – operační systém, na kterém běžel server, například:
  `Windows Server 2003`,
  `CentOS release 6.5 (Final) x86_64`,

- **language** – jazyk nastavený ve sledované aplikaci, například `en, ru, cs`,
- **resolution** – rozlišení obrazovky, například `1366x768`,
- **zoom** – hodnota přiblížení obrazovky, například `0, 1, 100`,
- **detectedBrowser** – detekovaný webový prohlížeč a jeho verze, například `Chrome 19`,
- **unifiedMessage** – unifikovaný chybový zpráva webového prohlížeče (viz kapitola 6.2.4),
- **unifiedCallStack** – unifikovaný callstack (viz kapitola 6.2.4),
- **errorMessage** – chybová zpráva prohlížeče, například: `TypeError: head is undefined`,
- **url** – url skriptu, ve kterém nastala reportovaná chyba, například: `/webmail/webmail2.js`,
- **line** – číslo řádky skriptu, na které nastala reportovaná chyba,
- **callStack** – callstack je informace obsažená ve výjimce[1] obsahující sekvenci volaných funkcí až k řádce, která výjimku způsobila:

```
kerio.dom.destroyNode@app.js:296
.destroy@app.js:109588
.destroy@extjs-4.1.3/ext.js:1910
.destroy@extjs-4.1.3/ext.js:1909
.destroy@app.js:109652
.destroy@extjs-4.1.3/ext.js:1910
._clearFeed/<@app.js:109864
._clearFeed@app.js:109865
._changeFeed@app.js:109852
.fire@extjs-4.1.3/ext.js:2465
.continueFireEvent@app.js:492
.fireEvent@app.js:473
._onClick@app.js:56452
anonymous@app.js:46869
Ext.EventManager.createListenerWrap/wrap@app.js:46825,
```

[1]Chyba aplikace vygeneruje callstack ve všech současných nejvíce používaných webových prohlížečích: Chrome, IE, Firefox, Opera

- **activeDialogId** – ID dialogu, který byl aktivní v době chyby, například:
  `_k_messageBox`, `waitingDialog`, `view [new window]`,
- **navigationItemId** – aktivní navigační prvek, například:
  `mail`, `calendar`,
- **clientDnsName** – doménové jméno klienta (detailní popis DNS viz [14]),
- **userComment** – komentář uživatele, který odesílal hlášení,
- **userEmail** – e-mailová adresa uživatele,
- **creationTime** – čas zápisu chyby do databáze,
- **descrambledMessage** – chybová zpráva prohlížeče, například:
  `Uncaught TypeError: Cannot read property 'dom' of null`,
- **buildNumber** – pořadové číslo sestavení produktu, například `2224`, `2225`,
- **windowSize** – velikost okna prohlížeče, například `1920x989`,
- **runTime** – doba od načtení webové aplikace do webového prohlížeče do vzniku chyby v sekundách,
- **developerToolActive** – informace o tom, zda byly aktivní nástroje vývojáře v prohlížeči (0 – nástroje vývojáře nebyly aktivní, nebo 1 – nástroje vývojáře byly aktivní),
- **browserSpeed** – detekovaná rychlost prohlížeče,
- **previousWidgetId** – identifikátor předchozího ovládacího prvku, například `mail`, `calendar`.
- **issue_unique** – definuje, zda je issue, do kterého je chybové hlášení přiřazeno, unikátní; možné hodnoty jsou (YES – chybové hlášení je unikátní, NO – chybové hlášení není unikátní),
- **fkey_message** – cizí klíč tabulky „message“,
- **isHidden** – definuje, zda je chybové hlášení skryto (není zobrazeno v tabulce přijatých chybových hlášení),
- **fkey_module** – cizí klíč tabulky „module“.

Tabulka „issue“

Tabulka „issue“ obsahuje jedinečné informace o skupinách chybových hlášení. Na základě nového chybového hlášení se vytvoří skupina, do které se poté přiřazují všechna následující hlášení, která jsou shodná s vytvořenou skupinou. Tabulka „issue“ obsahuje následující sloupce:

- **pkey_issue** – identifikátor položky tabulky (primární klíč),
- **comment** – komentář skupiny,
- **bug** – číslo „změnového požadavku“ přiřazeného do skupiny,
- **fkey_module** – cizí klíč tabulky „module“.

Tabulka „messages“

Tabulka „messages“ obsahuje unifikované chybové zprávy. Skládá se z následujících sloupců:

- **id** – identifikátor položky tabulky (primární klíč),
- **browser** – webový prohlížeč,
- **message** – původní chybová zpráva,
- **unifiedMessage** – unifikovaná zpráva.

Tabulka „email_template“

Tabulka „email_template“ obsahuje vytvořené e-maily a tvoří ji sloupce:

- **pkey_email_template** – identifikátor položky tabulky (primární klíč),
- **subject** – předmět e-mailu,
- **content** – obsah e-mailu,
- **issueId** – identifikační číslo issue,
- **lang** – jazyk e-mailu.

Tabulka „email“

Tabulka „email“ spojuje vytvořené e-maily (položky tabulky „email_template“) s e-mailovými adresami. Obsahuje následující sloupce:

- **pkey_email** – identifikátor položky tabulky (primární klíč),
- **address** – e-mailová adresa,
- **fkey_email_template** – cizí klíč tabulky „email_template“,
- **sentDate** – datum odeslání e-mailu.

Tabulka „email_reponse“

Tabulka „email_reponse“ obsahuje přijaté e-maily a obsahuje následující sloupce:

- **pkey_email_reponse** – identifikátor položky tabulky (primární klíč),
- **subject** – předmět e-mailu,
- **content** – obsah e-mailu,
- **fkey_email** – cizí klíč tabulky „email“.

Tabulka „product“

Tabulka „product“ je pouhým seznamem produktů, který obsahuje následující sloupce:

- **pkey_product** – identifikátor položky tabulky (primární klíč),
- **name** – jméno produktu.

Tabulka „module“

Tabulka „module“ obsahuje seznam modulů produktů společnosti Kerio (například Kerio Connect Client, WebAdministration). Tabulka má následující strukturu:

- **pkey_module** – identifikátor položky tabulky (primární klíč),
- **name** – jméno modulu.

Tabulka „component“

Tabulka „component“ obsahuje moduly produktů ve spojení s jejich verzemi a tvoří ji sloupce:

- **pkey_component** – identifikátor položky tabulky (primární klíč),
- **fkey_product** – cizí klíč tabulky „product“,
- **version** – verze produktu,
- **module** – název modulu produktu.

6 Implementace aplikace

V této kapitole bude popsána implementace aplikace na zpracování chybových hlášení, která nahradí stávající řešení popsané v kapitole 2. Cílem je odstranit některé nedostatky a zlepšit produktivitu procesu zpracování chybových hlášení. Důvody pro nahrazení byly podrobně popsány v kapitole 2.7.

Implementace byla rozdělena do dvou hlavních částí, a to:

- **server** – poskytuje API a soubory klienta, stará se o komunikaci s databází a dalšími systémy,
- **klient** – komunikuje se serverem, řeší proces zpracování chybových hlášení.

6.1 Server

Pro implementaci serverové části byla použita technologie Node.js umožňující napsat server v programovacím jazyce JavaScript. Použití stejného programovacího jazyka, jak na serveru, tak na klientovi, dovoluje sdílet kód. Sdílením kódu mezi serverem a klientem lze delegovat činnost na server nebo na klienta dle potřeby.

Mezi základní funkce serveru patří:

- **poskytování statických souborů** – poskytuje soubory klienta webovému prohlížeči,
- **klientské API** – poskytování a ukládání dat,
- **komunikace s databází** – modul mysql (blíže viz [19]) platformy Node.js,
- **komunikace se systémem Jira** – restové API systému Jira,
- **komunikace se systémem Autobuild** – restové API systému Autobuild,
- **odesílání e-mailů** – odesílání e-mailů bylo implementováno dvěma způsoby:

– API Kerio Connect,
– Nodemailer – modul technologie Node.js.

6.1.1 Načítání dat z databáze

Komunikaci s MySql databází zajišťuje modul *mysql*. Po příchodu požadavku na data z klienta se zavolá metoda serveru, která sestaví příslušný databázový dotaz. Dotaz je vykonán a na server jsou vrácena nalezená data. Příklad načtění dat z databáze je uveden níže.

```
var connection = mysql.createConnection({
    host: 'hostname', // jméno identifikující počítač
    user: 'jmeno uzivatele',
    password: 'heslo',
    database: 'jmeno databaze'
});

connection.query('SELECT count(reported_error.pkey_reported_error) AS amount
        FROM reported_error WHERE fkey_issue = ?', [issueId],
        function(error, data) {

    /* Metoda query vyžaduje funkci zpětného volání, tzv. callback,
       která je spuštěna pokud je získán výsledek, nebo nastane
       chyba ve vykonávání dotazu. Výsledek dotazu je vrácen jako
       jeden datový tok.*/

});
```

6.1.2 Komunikace se systémem Jira

Pro komunikaci se systémem Jira je použito restové API určené pro integraci s jinými aplikacemi. Pro komunikaci se používá potokol HTTP, přičemž jednotlivé zdroje jsou adresovány použitím URI cest a standardních HTTP metod GET, PUT, POST a DELETE. Data jsou odesílána a přijímána ve formátu JSON. Restové API systému Jira je blíže popsáno viz [16].

Před dotazem do systému Jira je nejprve nutné provést autorizaci. URI pro přístup k datům vypadá následovně:

`http://hostname/rest/api/api-version/resource`,

kde:

- `hostname` – unikátní jméno identifikující počítač,
- `api-version` – verze restového API,
- `resource` – identifikace požadovaného zdroje.

Konkrétně může URI vypadat následovně:

`http://jira.kerio.local/rest/api/2/issue/CONNECT-1111`.

Ze systému Jira jsou pro potřeby zpracování chybových hlášení získávána následující data:

- `status` – status „změnového požadavku“ (nový, ve vývoji, v testování, uzavřen, ...),
- `fixVersion` – verze, do které se má „změnový požadavek“ opravovat, popřípadě byl opraven,
- `created` – datum, kdy byl „změnový požadavek“ vytvořen,
- `lastUpdated` – datum poslední modifikace „změnového požadavku“ (například datum, kdy byl bug uzavřen),
- `url` – url, přes které je „změnový požadavek“ přístupný v systému Jira.

6.1.3 Odesílání e-mailů

V rámci řešení byly implementovány dva způsoby, jakými lze odeslat z aplikace e-mail, a to prostřednictvím:

- **API Kerio Connect** – napojení na API produktu Kerio Connect,
- **Nodemailer** – modul technologie Node.js.

Mezi oběma implementovanými řešeními je možné přepínat změnou příznaku v konfiguraci e-mailového spojení. Jako výchozí hodnota je nastaveno odesílání e-mailů přes API Kerio Connect. Výhodou tohoto řešení je archivace odeslaných e-mailů.

API Kerio Connect

Komunikace s produktem Kerio Connect probíhá prostřednictvím JSON RPC 2.0[1].

Nejprve je nutné se přihlásit, k tomu slouží metoda 'Session.login'. Příklad dat pro přihlášení vypadá následovně:

```
{
    application: {
        name: 'webassist',
        vendor: 'Kerio'
    },
    userName: 'uzivatelske jmeno pro prihlaseni do Kerio Connect',
    password: 'heslo uzivatele'
}
```

Po úspěšném přihlášení získáme sezení, prostřednictvím kterého je možné volat metody API včetně odesílání e-mailů. Metoda pro odeslání e-mailu se jmenuje 'Mails.create', data pro odeslání e-mailu vypadají následovně:

```
{
    mails: [{
        from: {
            address: 'odesilatel@domena'
        },
        to: [{
            address: 'prijemce@domena'
        }],
        subject: 'predmet e-mailu',
        displayableParts: [{
            contentType: 'ctTextPlain',
            content: 'obsah e-mailu'
        }],
        send: true
    }]
}
```

[1]Specifikace protokolu a příklad odeslání požadavku lze nalézt v [17]

Nodemailer

Nodemailer je modul technologie Node.js umožňující odesílání e-mailů přes SMTP protokol. Příklad odeslání e-mailu vypadá následovně:

```
var nodemailer = require('nodemailer');
var transporter = nodemailer.createTransport({
    service: 'gmail',
    auth: {
        user: 'odesilatel@gmail.com',
        pass: 'heslo'
    }
});
transporter.sendMail({
    from: 'odesilatel@address',
    to: 'prijemce@address',
    subject: 'Predmet e-mailu',
    text: 'Obsah e-mailu'
});
```

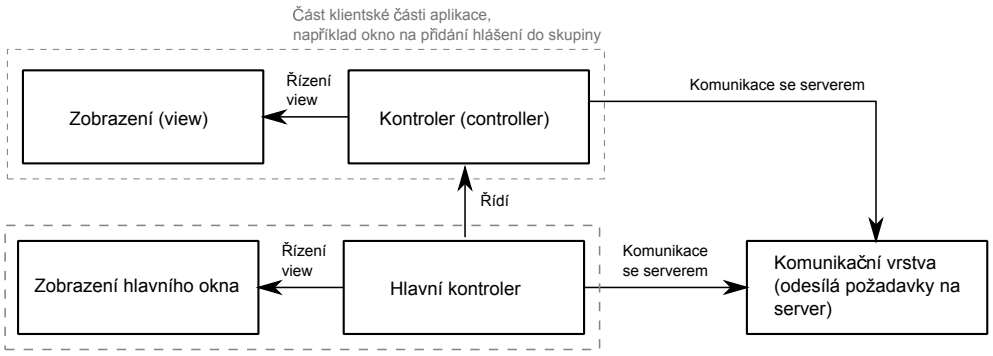
6.2 Klient

Klientská část aplikace je rozdělena na zobrazení (View) a kontrolery (Controller). Zobrazení zajišťuje vykreslení uživatelského rozhraní, kontrolery řídí zobrazení a komunikaci se serverem.

Každá část grafického uživatelského rozhraní má tak své zobrazení a svůj kontroler. Všechny kontrolery aplikace jsou řízeny jedním hlavním kontrolerem, který zajišťuje vytvoření všech kontrolerů a předávání kontextu mezi nimi. Komunikace v rámci klientské části aplikace je zobrazena na obrázku 6.1.

6.2.1 Zobrazení chybových hlášení

Po načtení aplikace ve webovém prohlížeči je nejprve nutné zvolit produkt, jehož chybová hlášení se budou zpracovávat. Po zvolení produktu jsou zobrazeny další možnosti:



Obrázek 6.1: Architektura aplikace

- **výběr modulu produktu** – například Kerio Connect Client,
- **výběr časového období** – časové období, pro které chceme zpracovávat chybová hlášení (například hlášení za poslední týden, měsíc, ...),
- **výběr požadované shody** – načtení nepřiřazených chybových hlášení, která splňují danou shodu s již přiřazeným hlášením,
- **odeslání vytvořených e-mailů** – umožňuje kontrolu a odeslání e-mailů vytvořených při zpracování chybových hlášení,
- **vyhledávání v nalezených hlášeních** – vyhledávání v chybových hlášení, která jsou aktuálně zobrazena,
- **vyhledávání v databázi** – možnost definování vlastností chybových hlášení, která se mají načíst z databáze (například chybová hlášení z prohlížeče Firefox, ...).

Po zvolení produktu, modulu a časového období jsou zobrazena nepřiřazená chybová hlášení, která splňují dané nastavení. Takto načtená hlášení je možné prohlížet a analyzovat (viz příloha B.1).

V detailním pohledu chybového hlášení jsou zobrazeny další informace:

- informace o „změnovém požadavku“ ze systému Jira, který je přiřazen do skupiny, se kterou se hlášení shoduje,
- počet aktuálně zobrazených hlášení, která se také shodují s danou skupinou,
- unifikovaný callstack.

6.2.2 Vytvoření skupiny

Skupina reprezentuje jednu chybu v aplikaci. Do skupiny lze přiřadit jedno nebo více chybových hlášení. Při vytváření skupiny (viz příloha B.4) lze zadat číslo „změnového požadavku“ ze systému Jira a komentář (pomáhající identifikovat chybu).

6.2.3 Přidání chybového hlášení do skupiny

Nevýhodou původního řešení byla nutnost přiřazení chybových hlášení po jednom. Při přiřazování chybových hlášení je nabídnuto přidání všech shodných nepřiřazených chybových hlášení (viz příloha B.2). Dále je při přiřazování umožněno:

- definování nové unifikované hlášky webových prohlížečů,
- vytvoření e-mailů pro zákazníky.

6.2.4 Unifikace

V rámci analýzy implementace nového řešení byla navržena unifikace chybových hlášek a callstacků, a to z důvodu odlišnosti webových prohlížečů. Unifikace umožní snadno porovnávat původně odlišná hlášení z různých prohlížečů.

Chybová hláška prohlížečů

Unifikace chybové hlášky (viz tabulka 5.1) umožňuje častěji přiřadit chybová hlášení pomocí absolutní shody, kdy místo hlášky prohlížeče se použije unifikovaná hláška. Při absolutní shodě je možné hlášení přiřadit do skupiny bez další analýzy.

Callstack

Unifikovaný callstack je automaticky vytovořen a ukládán do databáze a slouží pro porovnávání dvou hlášení. Každá řádka callstacku je převedena do

jednotného tvaru, a to:

metoda:`řádka in soubor`

V tabulce 6.1 je uveden příklad unifikace callstacků z různých webových prohlížečů, které způsobila stejná chyba.

6.2.5 Zobrazení callstacku

Pokud chybové hlášení obsahuje callstack, je možné jej zobrazit (viz příloha B.6). Pro zobrazení zdrojových kódů je ze systému Autobuild stažena verze aplikace, která odpovídá chybovému hlášení. Samotné zobrazení má pak dva módy:

- zobrazení části zdrojového kódu pro každou řádku callstacku,
- zobrazení celého souboru, kterého se řádka callstacku týká.

Mód zobrazení celého souboru může být použit při analýze větší části zdrojového kódu, což je výhodou oproti původní aplikaci, kde otevření celého souboru nebylo možné.

6.2.6 Zobrazení a modifikace skupin

Implementovaná aplikace umožňuje zobrazit seznam všech skupin chybových hlášení (viz příloha B.3). Ke každé skupině lze zobrazit všechny varianty chybových hlášení (některé mohou mít například rozdílný webový prohlížeč).

Dále je možné zobrazit podrobné informace o skupině, například kolik již bylo do skupiny přiřazeno chybových hlášení nebo e-maily, které byly ke skupině přiřazeny. V rámci skupiny je také možné modifikovat:

- číslo „změnového požadavku“, které je ke skupině přiřazeno,
- komentář popisující chybu,
- e-maily přiřazené do skupiny.

Původní callstack - MSIE 11	TypeError: Unable to get property 'isNode' of undefined or null reference at decorate (/webmail/lib/ext4/ext-all.js.2.2.2224:15094:96) at createNode (/webmail/lib/ext4/ext-all.js.2.2.2224:15115:2) at appendChild (/webmail/lib/ext4/ext-all.js.2.2.2224:15139:8) at fillNode (/webmail/lib/ext4/ext-all.js.2.2.2224:15503:6) at callParent (/webmail/lib/ext4/ext-all.js.2.2.2224:794:1) at fillNode (/webmail/webmail2.js.2.2.2224:53004:3) at onProxyLoad (/webmail/webmail2.js.2.2.2224:13589:4) at processResponse (/webmail/lib/ext4/ext-all.js.2.2.2224:7264:27) at Anonymous function (/webmail/webmail2.js.2.2.2224:27680:4) at _requestCallback (/webmail/webmail2.js.2.2.2224:7113:4)
Původní callstack - Chrome 31	TypeError: Cannot read property 'isNode' of undefined at Function.Ext.define.statics.decorate (/webmail/lib/ext4/ext-all.js.2.2.2224:15094:101) at createNode (/webmail/lib/ext4/ext-all.js.2.2.2224:15115:32) at appendChild (/webmail/lib/ext4/ext-all.js.2.2.2224:15139:12) at Ext.define.fillNode (/webmail/lib/ext4/ext-all.js.2.2.2224:15503:8) at b.callParent (/webmail/lib/ext4/ext-all.js.2.2.2224:794:13) at Ext.define.fillNode (/webmail/webmail2.js.2.2.2224:53005:18) at Ext.define.onProxyLoad (/webmail/webmail2.js.2.2.2224:13589:17) at Ext.define.processResponse (/webmail/lib/ext4/ext-all.js.2.2.2224:7264:29) at null.<anonymous> (/webmail/webmail2.js.2.2.2224:27680:7) at Object.kerio.api.impl.Ajax._requestCallback (/webmail/webmail2.js.2.2.2224:7113:18)
Původní callstack - Firefox 26	.statics.decorate@ext-all.js.2.2.2224:15094 .statics.getPrototypeBody/<.createNode@ext-all.js.2.2.2224:15115 .fillNode@ext-all.js.2.2.2224:15503 b.prototype.callParent@ext-all.js.2.2.2224:794 .fillNode@webmail2.js.2.2.2224:53005 .onProxyLoad@webmail2.js.2.2.2224:13589 .processResponse@ext-all.js.2.2.2224:7264 .createRequestCallback/<@webmail2.js.2.2.2224:27680 kerio.api.impl.Ajax._requestCallback@webmail2.js.2.2.2224:7113 .callback@ext-all.js.2.2.2224:1514 .onComplete@ext-all.js.2.2.2224:5213 .constructor/<.onStateChange@webmail2.js.2.2.2224:8077 Ext.Function.bind/<@webmail2.js.2.2.2224:69
Unifikovaný callstack	**decorate**:15094 in ext-all.js **getPrototypeBody**:15115 in ext-all.js **fillNode**:15503 in ext-all.js **callParent**:794 in ext-all.js **fillNode**:53005 in webmail2.js **onProxyLoad**:13589 in webmail2.js **processResponse**:7264 in ext-all.js **createRequestCallback**:27680 in webmail2.js **_requestCallback**:7113 in webmail2.js **callback**:1514 in ext-all.js **onComplete**:5213 in ext-all.js **onStateChange**:8077 in webmail2.js **bind**:68 in webmail2.js

Tabulka 6.1: Unifikace callstacku

6.2.7 Přesun chybových hlášení mezi skupinami

Implementovaná aplikace umožňuje přesun chybových hlášení mezi skupinami (viz příloha B.5). Důvodem pro přesun chybových hlášení může být:

- chybné přiřazení chybového hlášení při procesu zpracování,
- sjednocení skupin chybových hlášení, a to například, bylo-li zjištěno, že se skupiny týkají stejné chyby.

6.2.8 Kontrola a odesílání e-mailů

V rámci zpracování chybových hlášení je možné vytvářet e-maily pro zákazníky. Všechny vytvořené a neodeslané e-maily lze zobrazit, upravit a následně odeslat (viz příloha B.7).

Neodeslané e-maily jsou zobrazeny seskupeně, a to podle:

- **adresáta** – pokud je jednomu uživateli (na jednu e-mailovou adresu) odesíláno více e-mailů,
- **předmětu a obsahu** – pokud byl vytvořen pro několik e-mailových adres stejný e-mail (stejný předmět i obsah).

Seskupení se provádí pro usnadnění kontroly e-mailů. V případě seskupení dle adresáta je odesílán na jednu e-mailovou adresu pouze jeden e-mail. Seskupení dle obsahu je prováděno pouze pro zobrazení. Při odesílání jsou e-maily opět rozděleny a odeslány zvlášť pro každou e-mailovou adresu.

6.2.9 Zobrazení e-mailové komunikace

Pro každou e-mailovou adresu je uchována celá historie komunikace, kterou je také možné zobrazit (viz příloha B.8). Dále je možné uložit odpověď zákazníka.

6.2.10 Skrytí chybového hlášení při zpracování

Implementovaná aplikace umožňuje vyřadit některá nepřiřazená chybová hlášení z procesu zpracování. Důvodem vyřazení může být například nemožnost nalezení příčiny chyby.

Skrytí chybového hlášení je možné:

- **pro aktuální den** – informace je uložena ve webovém prohlížeči uživatele,
- **trvale** – informace o tom, že již chybu nechceme zobrazovat, je zapsána do databáze a takové chybové hlášení již není nikdy zařazeno do zpracování.

Implementovaná aplikace však umožňuje skrytá chybová hlášení zobrazit. Chybová hlášení lze také odstranit ze seznamu skrytých hlášení, přičemž budou zobrazena ihned při dalším zpracování.

7 Testování aplikace

V průběhu vývoje aplikace byla ověřována funčnost všech částí systému automatickými regresními testy. Testování bylo zaměřeno na end-to-end testování (viz kapitola 4.1.4), které ověřuje hlavní činnosti aplikace testováním běžných scénářů použití.

Pro testování aplikace byl použit testovací framework Protractor (viz kapitola 3.2), který je pro end-to-end testování webových aplikací určen.

7.1 Automatické regresní testy

Automatické regresní testy byly rozděleny do skupin podle části aplikace, kterou testují. Byly tedy sepsány scénáře použití aplikace, které poté byly převedeny do podoby testů. Mezi takové scénáře například patří:

- načtení nepřiřazených chybových hlášení,
- vyhledávání shod chybových hlášení se skupinami,
- vytvoření skupiny chybových hlášení,
- přidání chybových hlášení do skupiny, které zahrnuje,
  - vytvoření e-mailů pro zákazníky,
  - vytvoření unifikované zprávy,
- přesun chybových hlášení mezi skupinami,
- zobrazení skupin a jejich úprava,
- prohlížení odeslaných e-mailů a přidání odpovědi,
- prohlížení callstacku chybového hlášení.

Pro každou z výše zmíněných funkčností byly napsány automatické testy, které ověřují její funkcionalitu. Jako příklad scénáře testu může být uveden následující postup:

1. načtení aplikace do webového prohlížeče,
2. nastavení konfigurace aplikace (produkt, modul, časové období),
3. zvolení typu shody,
4. čekání na načtení chybových hlášení do aplikace,
5. ověření, zda byla zobrazena očekávaná data v očekávaném pořadí.

Ukázka zdrojového kódu regresního testu je v příloze D.

7.2 Uživatelské testování

Kromě automatických testů byla také aplikace otestována ručně. Pro testování byly použity scénáře, které simulují chování uživatele, který s aplikací bude pracovat. V rámci ručního testování byla hodnocena a testována nejen bezchybnost aplikace, ale také uživatelská přívětivost a s tím související rychlost odezvy aplikace.

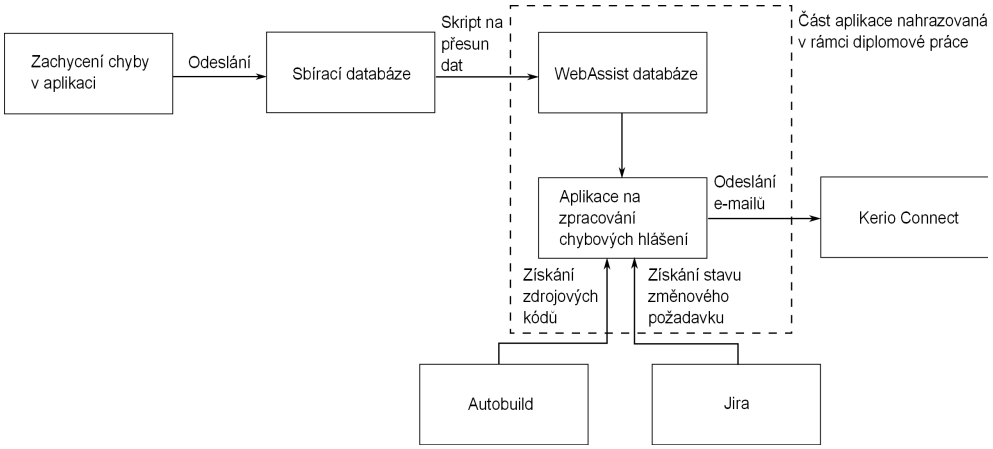
V rámci uživatelského testování bylo také ověřeno chování aplikace v případě selhání systémů, na které je aplikace napojena. Tedy například jak se aplikace zachová, pokud je některý ze systémů (Jira, Autobuild, Kerio Connect Client) nedostupný. Dále byla aplikace otestována pro případ, když selže spojení s databází.

8 Zhodnocení dosažených výsledků

V této kapitole budou popsána vylepšení implementované aplikace oproti původní aplikaci. Cílem implementované aplikace bylo zdokonalení procesu zpracování chybových hlášení a urychlení odezvy aplikace na uživatelské vstupy, zejména rychlost zpracování databázových dotazů.

8.1 Proces zpracování

Proces zpracování chybových hlášení je zobrazen na obrázku 8.1, kde je uveden proces od vzniku chyby v produktu až po komunikaci se zákazníky (diagram původního procesu zpracování je zobrazen na obrázku 2.1).



Obrázek 8.1: Proces zpracování v implementovaném řešení

V procesu zpracování bylo omezeno množství rutinních činností a počet hlášení, které vyžadují analýzu uživatelem. Část procesu byla zautomatizována, a to včetně e-mailové komunikace se zákazníky.

UML diagram tříd implementované aplikace je uveden v příloze A.

8.1.1 Analýza chybových hlášení

Některá chybová hlášení vyžadují analýzu pouze z důvodu odlišnosti webových prohlížečů. To je řešeno unifikací chybové zprávy a unifikací callstacku. Unifikací byl zvýšen počet chybových hlášení, která je možné zpracovat na základě absolutní shody. Absolutní shoda je jediným typem shody, kde není vyžadována analýza.

8.1.2 Přiřazování chybových hlášení do skupin

Proces přiřazování byl rozšířen o možnost přiřazení více chybových hlášení do jedné skupiny v jednom kroku. Při přiřazování chybového hlášení jsou nalezena všechna nepřiřazená hlášení, která jsou shodná se stejnou skupinou na základě stejné shody. Takto nalezená hlášení lze poté najednou přiřadit.

8.1.3 E-mailová komunikace se zákazníky

Jedním z podnětů pro vznik této práce byla integrace e-mailové komunikace se zákazníky. V rámci zpracování nových chybových hlášení je možné přiřadit e-mail jako odpověď zákazníkovi. E-mail je možné dále prohlížet, upravit a odeslat. Při odesílání se e-maily automaticky seskupují (viz kapitola 6.2.8). Odeslané e-maily je možné dohledat.

8.2 Výkonnost

V rámci implementované aplikace byla navržena nová struktura databáze a možnost zpracování složitých dotazů po částech. Tím bylo docíleno urychlení odezvy aplikace na vstupy uživatele.

8.2.1 Optimalizace struktury databáze

Změnou struktury databáze se docílilo zrychlení vykonávání dotazů. Databáze při testování běžela na počítači s následující konfigurací:

- operační systém: Windows 7 64-bit,
- procesor: Intel Core i7-3517U CPU 1,9 GHz,
- paměť: 8 GB RAM,
- typ disku: iSSD cache,
- průměrná přenosová rychlost při čtení z disku: 90,1 MB/s.

Při porovnávání byl vždy měřen čas potřebný pro zobrazení prvního chybového hlášení určeného k zpracování. V případě původní struktury se tedy jedná o čas, který je potřebný k načtení všech nepřiřazených chybových hlášení za dané období (tento dotaz je spuštěn při každé změně shody) a následné nalezení první shody hlášení s nějakou existující skupinou. V nově navržené struktuře se vždy jedná o čas potřebný k získání první části dat. Zrychlení oproti původní struktuře je uvedeno v tabulce 8.1.

Zrychlení nové struktury oproti původní není příliš patrné pro krátké časové období. Pokud je však zvoleno období delší, je zrychlení markantní. Například načtení nepřiřazených hlášení za období jednoho týdne již trvá deset sekund. Pokud se časové období prodlouží na měsíc, tak se dotaz zpracovává téměř minutu.

V nově navrhnuté struktuře databáze není období, pro které chceme hlášení zpracovávat, důležité, jelikož postupně načítá hlášení na pozadí aplikace a uživatel tak není dobou vyhledávání ovlivněn.

8.2.2 Odezva aplikace při zpracování náročných dotazů

Zrychlení odezvy aplikace bylo docíleno načítáním dat z databáze po částech. Ihned po návratu první části dat je možné začít proces zpracování. Další části dat jsou načítány na pozadí aplikace. Také je umožněno dotaz přerušit nebo jej změnit před dokončením.

	Doba trvání dotazu [s]		
Struktura databáze	**Původní struktura**		**Aktuální struktura**
Období	**Týden**	**Den**	**Stejné pro libovolné období (první část dat)**
Všechna nezpracovaná hlášení	9.792	0.332	0.006
Absolutní shoda	10.480	0.352	0.295
Rozdílná verze detekovaného prohlížeče	9.689	0.313	0.089
Rozdílný detekovaný prohlížeč	10.134	0.317	0.111
Rozdílná chybová hláška	10.090	0.342	0.233
Rozdílná chybová hláška a číslo sestavení	9.828	0.339	0.169
Rozdílný detekovaný prohlížeč a chybová zpráva	10.344	0.367	0.141
Rozdílné číslo sestavení a řádka	10.244	0.346	0.170
Shodná chybová hláška	9.883	0.332	0.177

Tabulka 8.1: Porovnání časů potřebných pro vyhledávání shod

9 Závěr

Cílem práce byl návrh a implementace aplikace na zpracování chybových hlášení z webových aplikací. Zadání kladlo na implementaci následující požadavky: automatizace části procesu, integrace s interními systémy a doplnění chybějící funkčnosti předchozího řešení. Na začátku práce byla uvažována možnost převzít z původního řešení určité moduly, například strukturu databáze, ale po důkladné analýze bylo rozhodnuto, že aplikace bude naprogramováná kompletně znovu.

Podrobné zhodnocení výsledků lze nalézt v předchozí kapitole, takže zde je možné jen stručně konstatovat, že se v rámci práce podařilo řádově urychlit načítání dat z databáze, a to změnou struktury databáze do podoby, která více odpovídá potřebám současného zpracování chyb. Dále se podařilo zjednodušit proces zpracování z uživatelského hlediska, což zahrnuje i redukci počtu akcí, které je nutné při zpracování nahlášených chyb provést.

Na počátku celé práce bylo velkým problémem detailní porozumění procesu zpracování chybových hlášení. Proces zpracování je komplikovaný a navržený tak, aby v maximální míře automatizoval analýzu jednotlivých chybových hlášení. Z tohoto důvodu obsahuje celou kaskádu nástrojů hledajících stejné chyby, které se vyskytly v jiných podmínkách. V původním řešení nebyla obsluha těchto nástrojů příliš intuitivní a byla často vícekroková. Toto je v novém řešení též odstraněno.

Ačkoliv byla před zahájením implementace provedena důsledná analýza, došlo v průběhu implementace k požadavkům i na zásadnější změny. Z toho důvodu bylo nutné implementaci opakovaně refaktorovat. Vzhledem k tomu, že je JavaScript slabě typovaný programovací jazyk, je rozsáhlý refaktoring časově náročný a náchylný k chybám. Správná funkčnost aplikace byla v průběhu rozsáhlých úprav ověřována použitím automatických regresních testů.

V současné době je aplikace plně funkční, zcela splňuje požadavky zadání a po migraci dat bude postupně nasazována ve firmě Kerio.

Do budoucna by bylo možné aplikaci rozšířit o podporu lokalizace. V první řadě by bylo vhodné lokalizovat e-mailovou komunikaci se zákazníky, která je v současné době vedena pouze v anglickém jazyce. Další možností v oblasti lokalizace by byl automatický překlad uživatelských komentářů do anglického jazyka, například použitím Google Translate API.

Literatura

[1] BLADES, Steve; FREDERICK, Shea; RAMSEY, Colin. *Learning Ext JS*. Packt Publishing, 2008. ISBN 1-8471-9514-8.

[2] GRAHAM, Dorothy; VAN VEENENDAAL, Erik; EVANS Isabel; BLACK Rex. *Foundations of Software Testing: ISTQB Certification*. Thomson, 2006. ISBN 1-8448-0355-4.

[3] TABAGBOGHI, Seyed; WILLIAMS, Hugh. *Learning MySQL*. O'Reilly Media, 2006. ISBN 0-5960-0864-3.

[4] WILLIAMSON, Ken. *Learning AngularJS*. O'Reilly Media, 2015. ISBN 1-4919-1675-3.

Další elektronické zdroje

[5] BONETTA, Daniele. *The Parallel Event Loop Model and Runtime* [online]. [cit.2015-4-28].
Dostupné z <http://design.inf.unisi.ch/sites/default/files/biblio/daniele-bonetta-phd-parallel-javascript.pdf>.

[6] CONDIT, Jeremy; VON BEHREN, Rob; BREWER, Eric. *Why Events Are A Bad Idea (for high-concurrency servers)* [online]. [cit.2015-4-27].
Dostupné z <https://www.usenix.org/legacy/events/hotos03/tech /full_papers/vonbehren/vonbehren.pdf>.

[7] DEBEK, Frank; ZELDOVICH, Nickolai; KAASHOEK Frans; MAZIÉRES, David, MORRIS, Robert. *Event-driven Programming for Robust Software* [online]. [cit.2015-4-27].
Dostupné z <http://www.scs.stanford.edu/~dm/home/papers/dabek:event.pdf>.

[8] KUMAR, Bimal Aklesh. *Evaluation of Fiji National University Campus Information Systems* [online]. [cit.2015-4-3].
Dostupné z <http://arxiv.org/ftp/arxiv/papers/1106/1106.3369.pdf>.

[9] MICHÁLEK, Lukáš. *Použití metod testování softwaru v praxi* [online]. [cit.2015-4-28].
Dostupné z <http://www.vse.cz/vskp/show_file.php?soubor_id=46>.

[10] PAŽOUREK, Tomáš. *Webové služby v architektuře REST na platformě .NET* [online]. [cit.2015-5-2].
Dostupné z <http://is.muni.cz/th/359464/fi_b/thesis.pdf>.

[11] WILLIAMS, Laurie. *Testing Overview and Black-Box Testing Techniques* [online]. [cit.2015-5-2].
Dostupné z <http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf>.

[12] ZELINKA, Bořek. *Testování softwaru* [online]. [cit.2015-4-2]. Dostupné z <http://d3s.mff.cuni.cz/teaching/commercial_workshops/previous/1213/zelinka-zajisteni_kvality_softwarovych_produktu.pdf>.

[13] *About Node.js®* [online]. [cit.2015-3-23]. Dostupné z <https://nodejs.org/about/>.

[14] *DOMAIN NAMES - CONCEPTS AND FACILITIES* [online]. [cit.2015-4-14]. Dostupné z <http://www.ietf.org/rfc/rfc1034.txt>.

[15] *Chapter 1. MySQL Architecture and History* [online]. [cit.2015-3-24]. Dostupné z <https://www.safaribooksonline.com/library/view/high-performance-mysql/9781449332471/ch01.html>.

[16] *JIRA 6.4.2 REST API documentation* [online]. [cit.2015-4-17]. Dostupné z <https://docs.atlassian.com/jira/REST/latest/>.

[17] *JSON-RPC 2.0 Specification* [online]. [cit.2015-4-17]. Dostupné z <http://www.jsonrpc.org/specification>.

[18] *Nodemailer* [online]. [cit.2015-3-23]. Dostupné z <http://adilapapaya.com/docs/nodemailer/>.

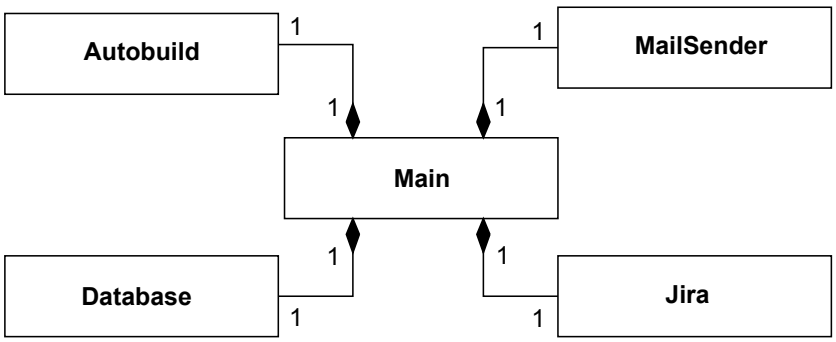
[19] *Node.js and MySQL tutorial* [online]. [cit.2015-3-23]. Dostupné z <http://codeforgeek.com/2015/01/nodejs-mysql-tutorial/>.

[20] *Protractor – How It Works* [online]. [cit.2015-3-23]. Dostupné z <http://angular.github.io/protractor/#/infrastructure>.

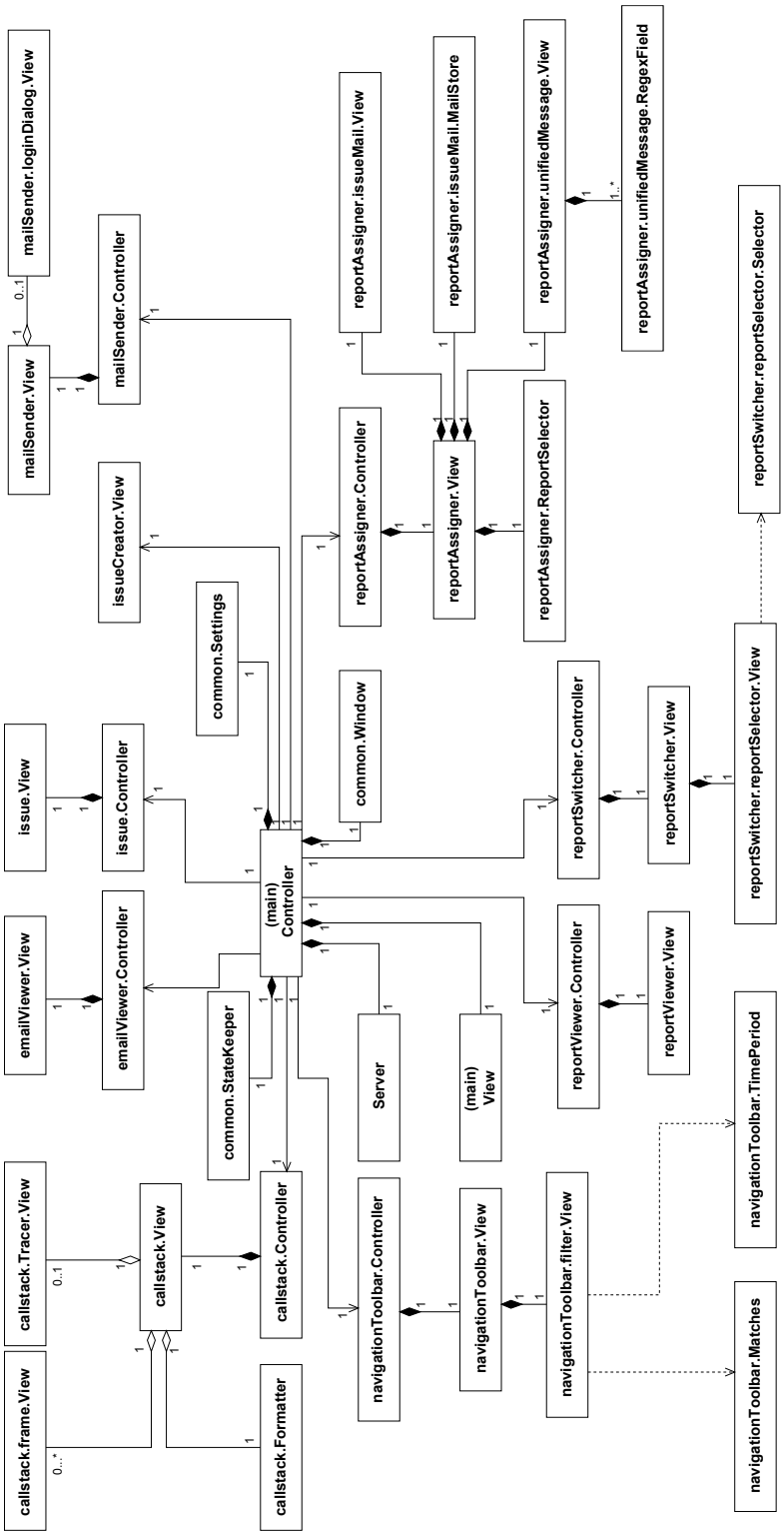
[21] *Uniform Resource Identifier (URI): Generic Syntax* [online]. [cit.2015-4-14]. Dostupné z <https://www.ietf.org/rfc/rfc3986.txt>.

A UML diagram tříd

UML diagram je rozdělen na serverovou (viz obrázek A.1) a klientskou (viz obrázek A.2) část. Klientská část obsahuje pouze třídy implementující aplikační logiku. Nejsou zde tedy obsaženy univerzální třídy, jako například třída, která slouží pro renderování tabulek v aplikaci nebo třída zobrazující dialogové okno.



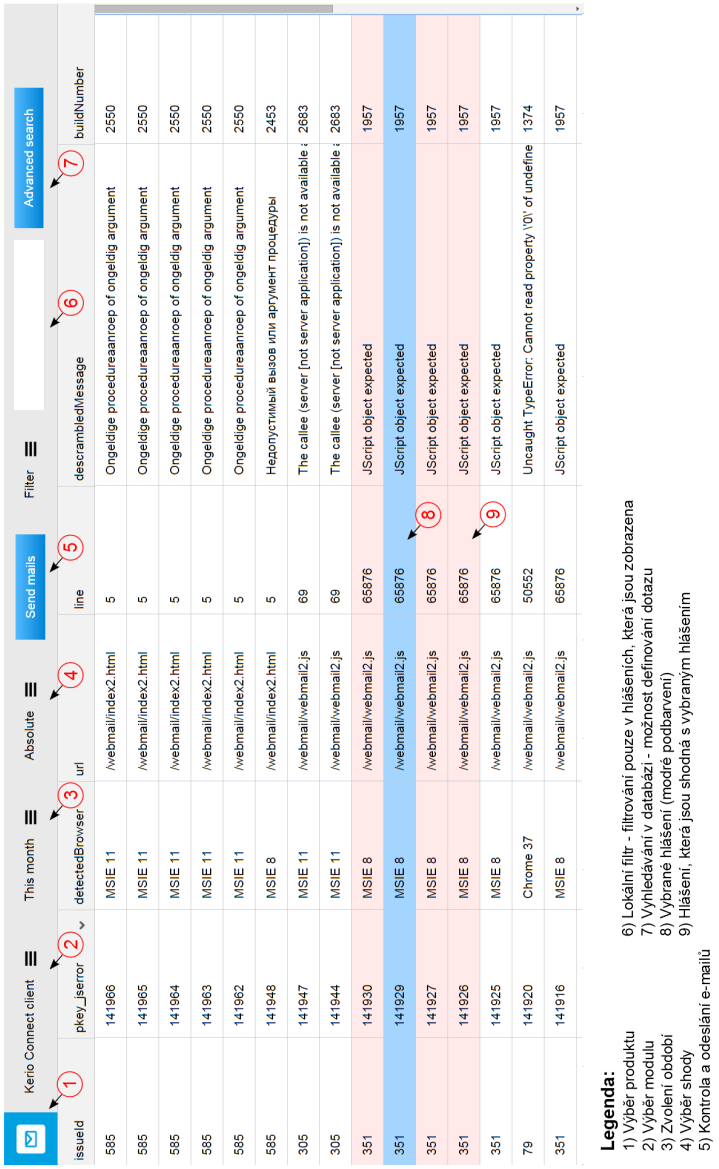
Obrázek A.1: UML diagram tříd serverové části implementace



Obrázek A.2: UML diagram tříd klientské části aplikace

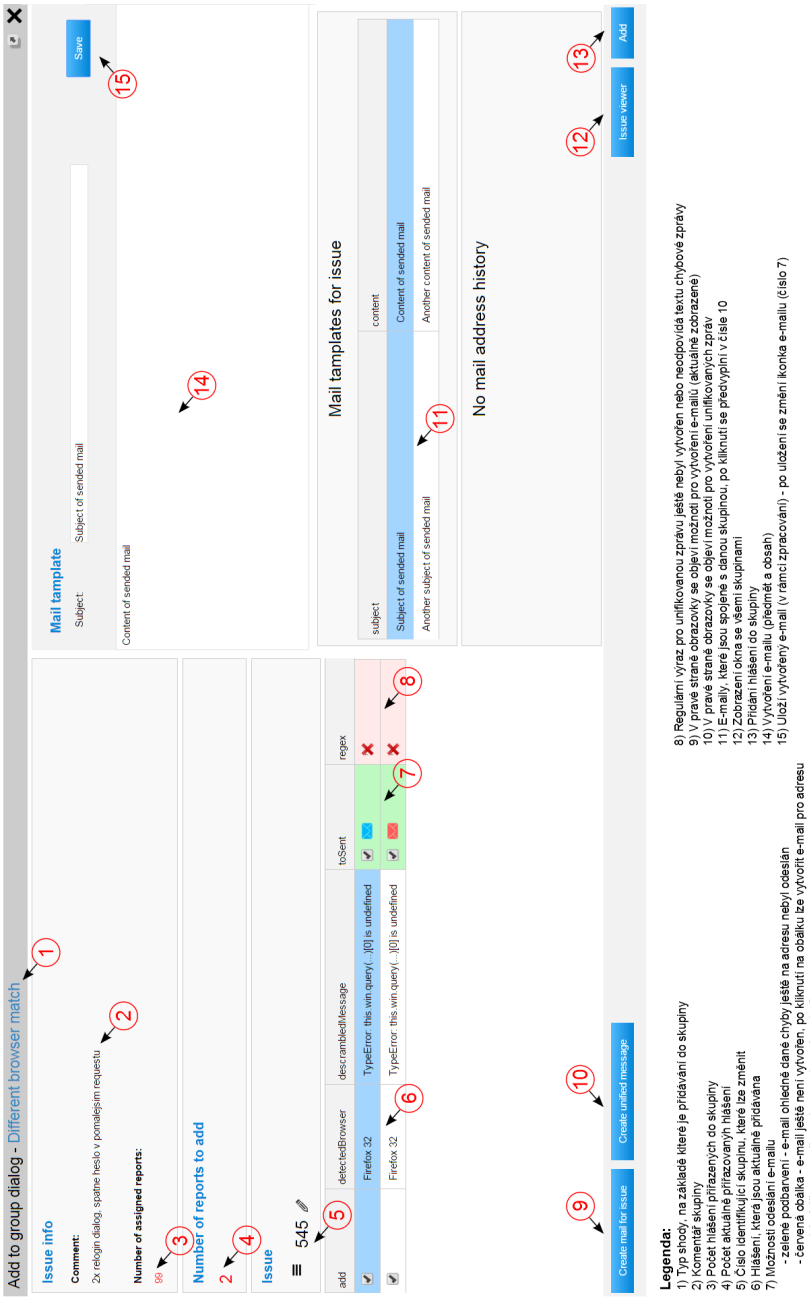
B Snímky obrazovek aplikace

B.1 Hlavní stránka aplikace



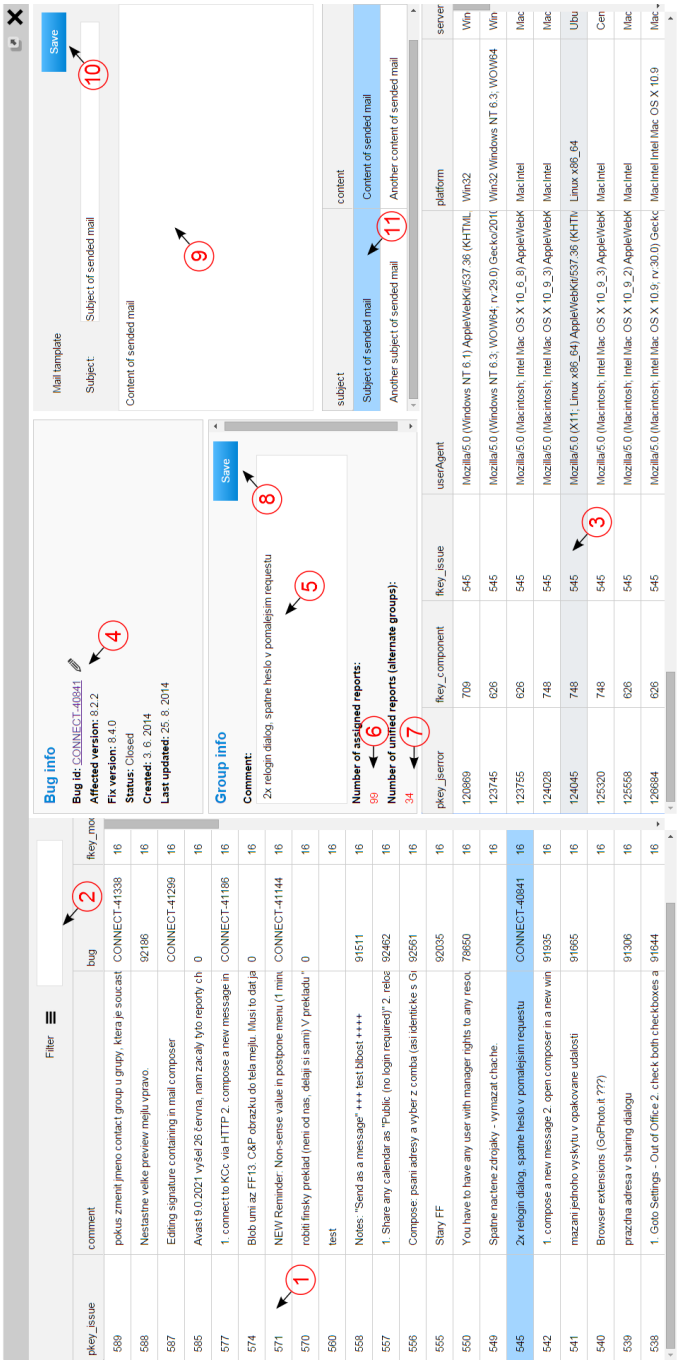
Obrázek B.1: Zobrazení chybových hlášení

B.2 Přidání hlášení do skupin



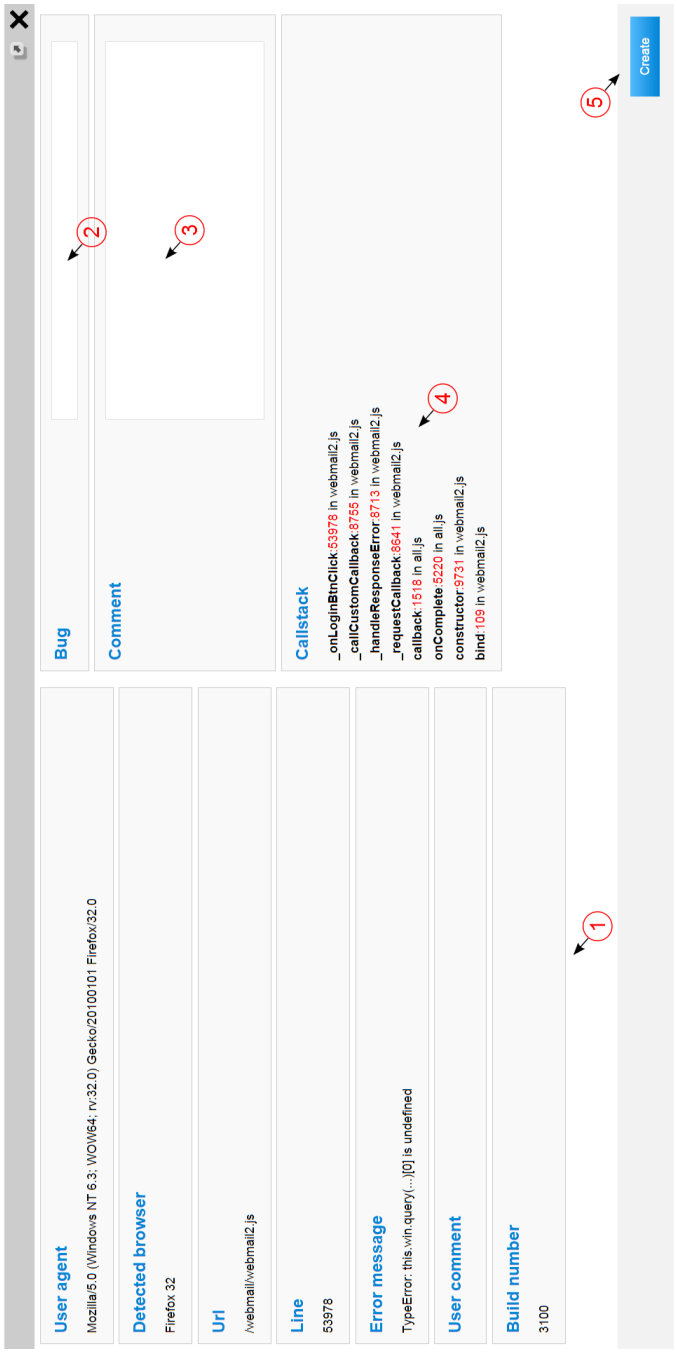
Obrázek B.2: Přidání hlášení do skupiny

B.3 Všechny skupiny



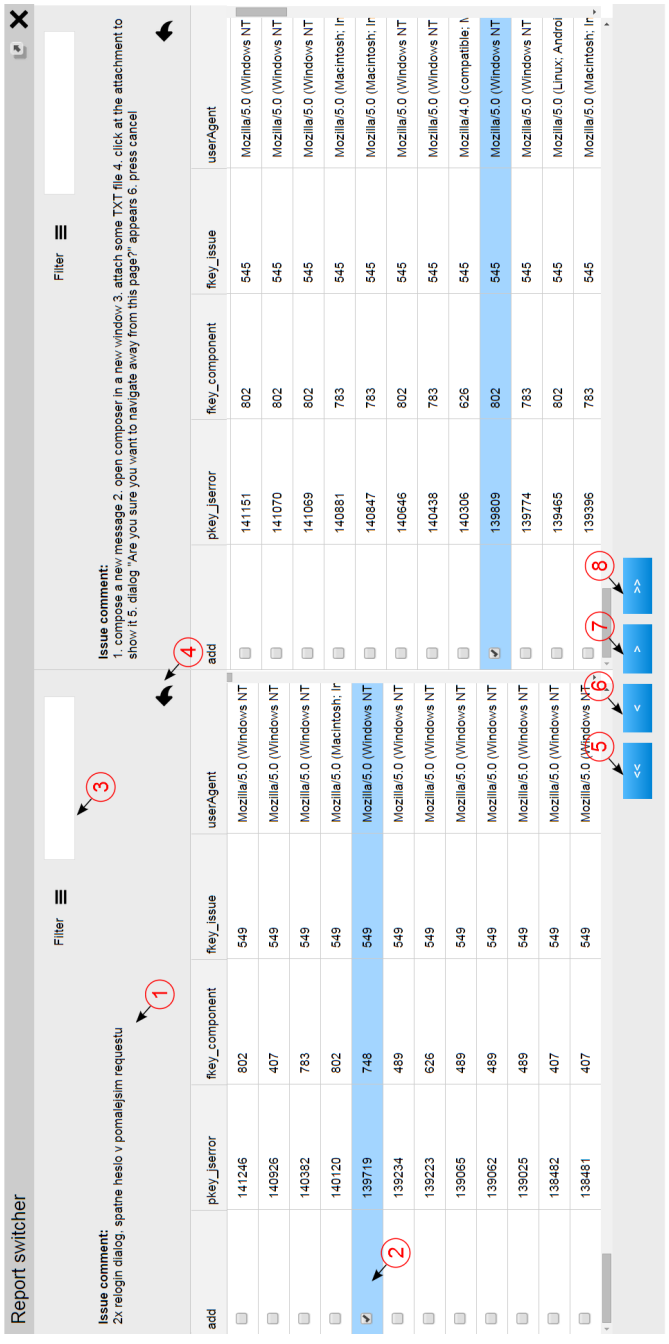
Obrázek B.3: Zobrazení všech existujících skupin

B.4 Vytvoření skupiny



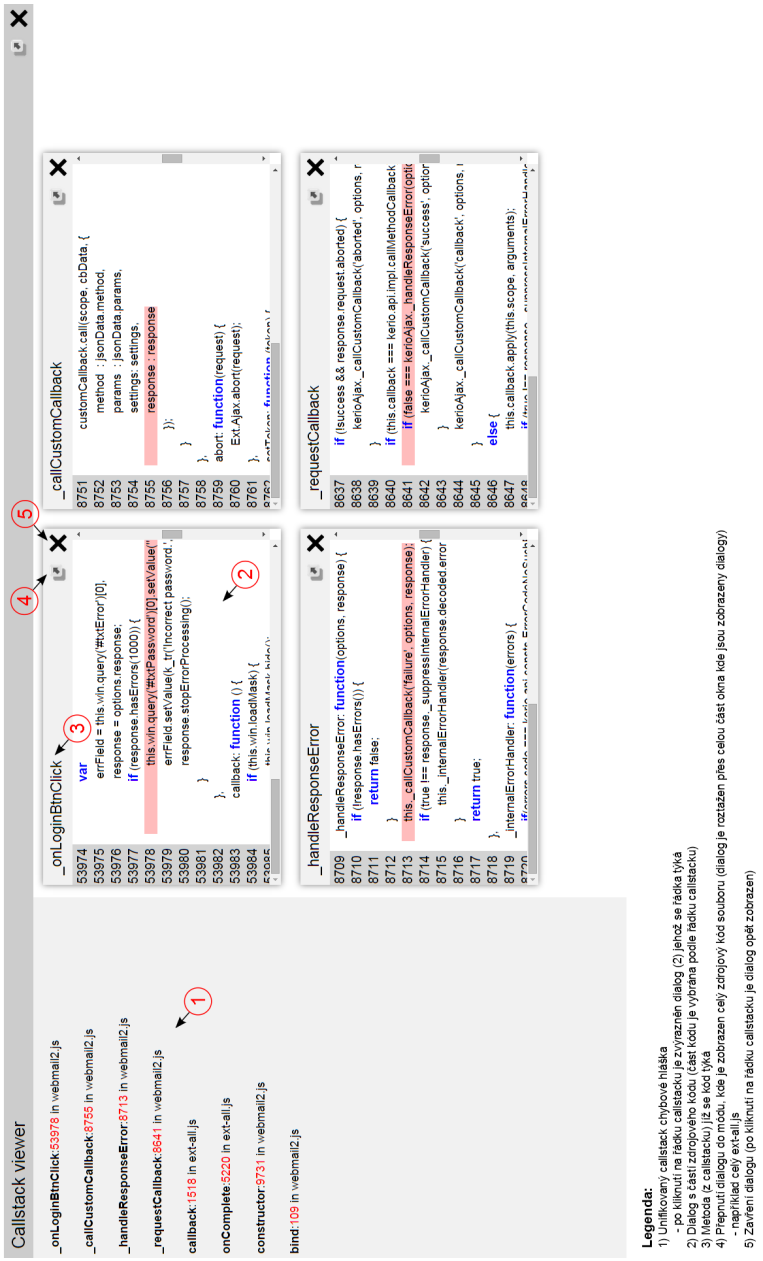
Obrázek B.4: Vytvoření skupiny

B.5 Přesun hlášení



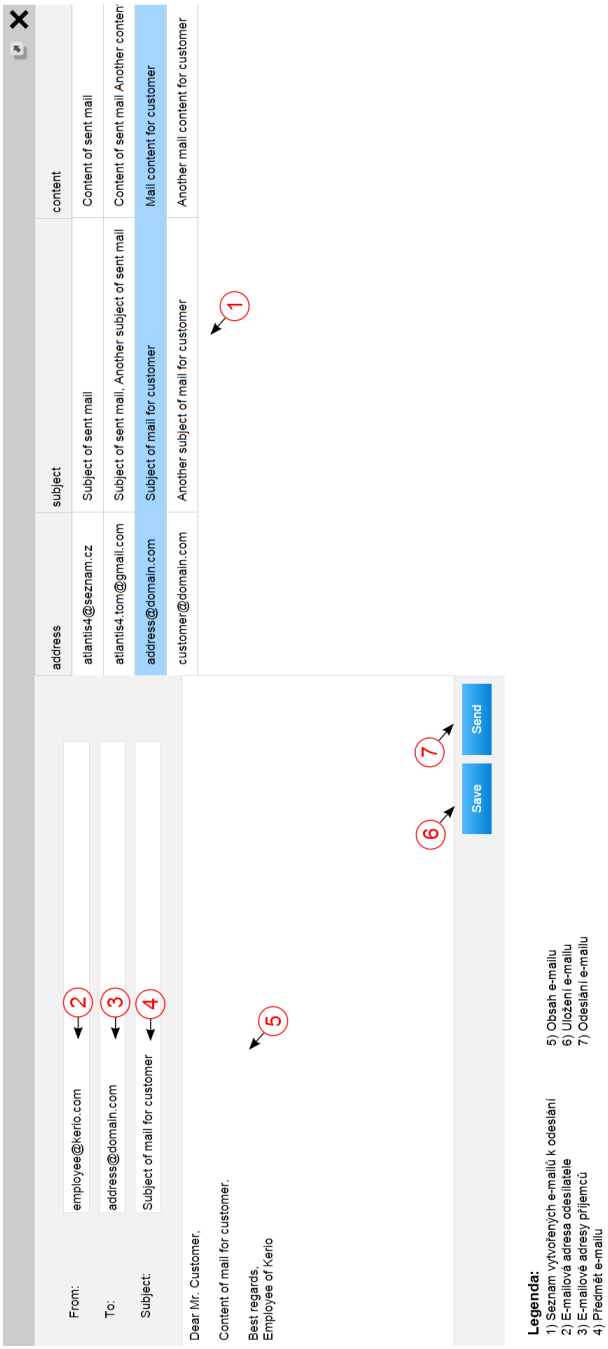
Obrázek B.5: Přesun hlášení mezi skupinami

B.6 Zobrazení callstacku



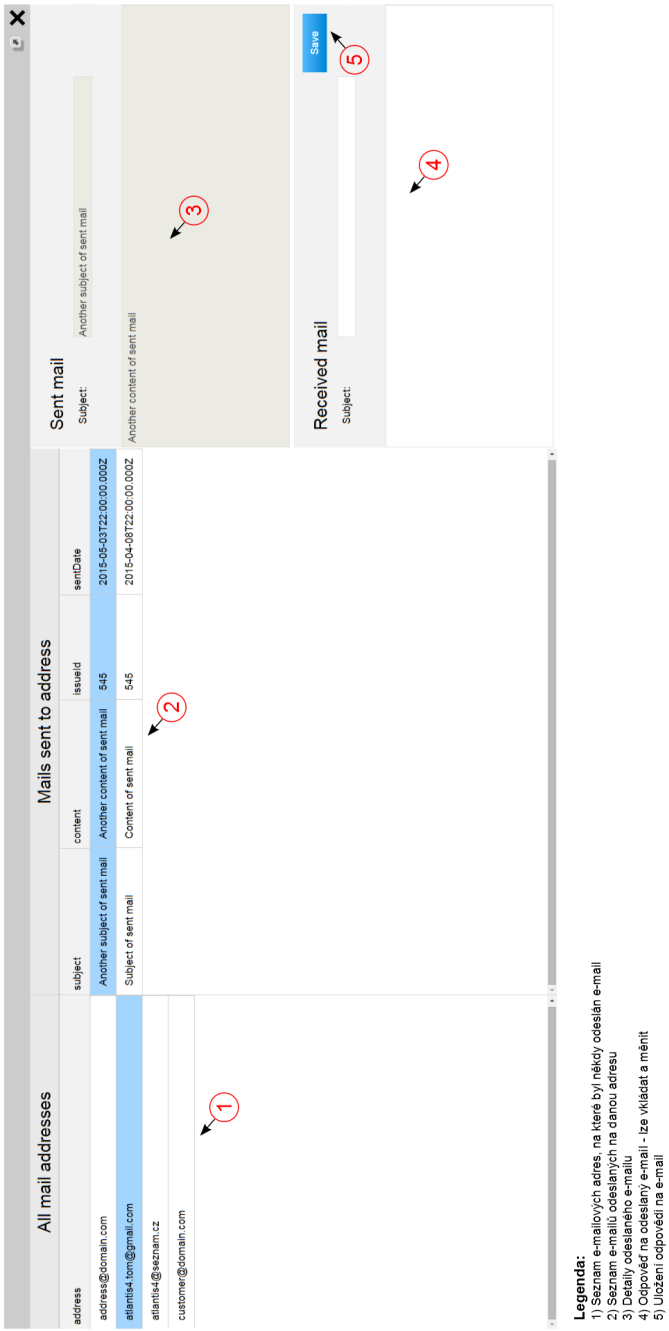
Obrázek B.6: Zobrazení callstacku chybové hlášky

B.7 Odeslání e-mailů



Obrázek B.7: Odesílání e-mailů

B.8 Prohlížení odeslaných e-mailů



Obrázek B.8: Prohlížení odeslaných e-mailů

C Ukázka kódu regresního testu

V této příloze je uveden kód regresního automatického testu pro testování obsahu tabulky. Testovací scénář je následující:

1. Načtení webové aplikace, která zobrazí chybová hlášení pro požadované nastavení,
2. ověření existence očekávaných chybových hlášení.

Kódu regresního automatického testu:

```
var driver = browser.driver;

describe('absolute table content', function () {
    it('absolute table content', function () {
        var data = [
            // testovaci data
        ];

        var i;
        driver.get('http://localhost:8080/#dateFrom={"from":1409436000000,"code":"This month"}
            &dateTo={}&productId=4&module=Kerio Connect client&match=Absolute');

        for (i = 0; i < data.length; i++) {
            var path = "(//*[contains(@class, 'reportTable')]//tr)[" + (i + 2) + "]"
                + "//td[normalize-space()='" + data[i].pkey_reported_error + "']"
                + "//following-sibling::td[normalize-space()='" + data[i].detectedBrowser + "']"
                + "//following-sibling::td[normalize-space()='" + data[i].url + "']"
                + "//following-sibling::td[normalize-space()='" + data[i].line + "']"
                + "//following-sibling::td[normalize-space()='" + data[i].userEmail + "']"
                + "//following-sibling::td[normalize-space()='" + data[i].creationTime + "']"
                + "//following-sibling::td[normalize-space()='" + data[i].descrambledMessage + "']"
                + "//following-sibling::td[normalize-space()='" + data[i].issueId + "']";
            driver.wait(function () {
                return driver.isElementPresent(by.xpath(path));
            }, 20000);
        }
    });
});
```

D Obsah CD

- **sources** – obsahuje zdrojové soubory aplikace a spouštěcí skripty,
- **sources/client** – zdrojové soubory klientské části aplikace,
- **sources/server** – zdrojové soubory serverové části aplikace,
- **tests** – zdrojové soubory regresních automatických testů a skripty potřebné pro spuštění testů,
- **text** – obsahuje text a obrázky diplomové práce.