



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

DEPARTMENT OF INTELLIGENT SYSTEMS

**KLIENTSKÁ APLIKACE PRO JAZYKOVÝ SERVER
APACHE CAMEL**

CLIENT APPLICATION FOR APACHE CAMEL LANGUAGE SERVER

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

FILIP POSPÍŠIL

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. ONDŘEJ VAŠÍČEK

BRNO 2023

Zadání bakalářské práce



148348

Ústav: Ústav inteligentních systémů (UITS)
Student: **Pospišil Filip**
Program: Informační technologie
Specializace: Informační technologie
Název: **Klientská aplikace pro jazykový server Apache Camel**
Kategorie: Softwarové inženýrství
Akademický rok: 2022/23

Zadání:

1. Seznamte se s protokolem LSP (Language Server Protocol), s architekturou LSP klientů a s jazykovým serverem Apache Camel.
2. Nastudujte možnosti rozšíření vývojového prostředí Apache NetBeans o nové zásuvné moduly a jeho aktuálně dostupnou podporu protokolu LSP.
3. Navrhněte LSP klientskou aplikaci do vývojového prostředí Apache NetBeans pro jazykový server Apache Camel.
4. Implementujte LSP klientskou aplikaci navrženou v předchozím bodě.
5. Navrhněte způsob otestování implementovaného řešení a proveďte testování.
6. Shrňte dosažené výsledky a srovnajte je s alternativními dostupnými řešeními.

Literatura:

- Microsoft. Language Server Protocol. Dostupné na URL: <https://microsoft.github.io/language-server-protocol>
- Apache NetBeans. Dostupné na URL: <https://netbeans.apache.org>
- Apache Camel Language Server. Dostupné na URL: <https://github.com/camel-tooling/camel-language-server>

Podrobné závazné pokyny pro vypracování práce viz <https://www.fit.vut.cz/study/theses/>

Vedoucí práce: **Vašíček Ondřej, Ing.**
Konzultant: Jelínek Dominik, Ing.
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1.11.2022
Termín pro odevzdání: 10.5.2023
Datum schválení: 3.11.2022

Abstrakt

Předmětem této bakalářské práce je návrh a implementace zásuvného modulu poskytujícího jazykovou podporu pro *Apache Camel* v integrovaném vývojovém prostředí *Apache NetBeans*. Cílem práce je umožnit integraci jazykového klienta pro integrované vývojové prostředí *Apache NetBeans*. Klient ke své funkci využívá již existující jazykový server pro *Apache Camel* vytvořený společností *Red Hat*. Práce poskytuje základní úvod k jazykovému protokolu *Microsoft Language Protocol*, rámci *Apache Camel* a integrovanému vývojovému prostředí *Apache NetBeans*. Dále je popsán návrh, implementace a testování vytvořeného jazykového klienta. Výsledkem této práce je zásuvný modul poskytující komplexní jazykovou podporu pro rámec *Apache Camel* ve vývojovém prostředí *Apache NetBeans*. Zásuvný modul bude v budoucnu zveřejněn v repozitáři zásuvných modulů *Apache NetBeans Plugin Portal*.

Abstract

The goal of this Bachelors thesis is the design and implementation of a plug-in that will provide language support for *Apache Camel* in the integrated development environment *Apache NetBeans*. The aim of the thesis is to enable integration of the language client for the integrated development environment *Apache NetBeans*. The client utilizes an existing language server for *Apache Camel* created by *Red Hat*. The thesis provides a basic introduction to the *Microsoft Language Protocol*, the *Apache Camel* framework, and the integrated development environment *Apache Netbeans*. Next, the design, implementation, and testing of the created language client are described. The result of this thesis is a plug-in module that provides comprehensive language support for the *Apache Camel* framework in the *Apache Netbeans* development environment. In the future the plug-in will be published in the *Apache NetBeans Plugin Portal* repository.

Klíčová slova

Apache Camel, Apache NetBeans, JSON-RPC, Language Server Protocol, LSP

Keywords

Apache Camel, Apache NetBeans, JSON-RPC, Language Server Protocol, LSP

Citace

POSPÍŠIL, Filip. *Klientská aplikace pro jazykový server Apache Camel*. Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Ondřej Vašíček

Klientská aplikace pro jazykový server Apache Camel

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením vedoucího bakalářské práce. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Filip Pospíšil
6. května 2023

Poděkování

Rád bych poděkoval vedoucímu bakalářské práce panu Ing. Ondřeji Vašíčkovi a externímu odbornému konzultantovi z firmy Red Hat panu Ing. Dominikovi Jelínkovi za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Obsah

1	Úvod	2
1.1	Motivace	2
2	Přehled souvisejících technologií	3
2.1	Language Server Protocol	3
2.2	JSON-RPC	6
2.3	Apache Camel	8
2.4	Jazykový server pro Apache Camel	13
3	Vývojové prostředí Apache NetBeans	14
3.1	Architektura	14
3.2	Modulární architektura	15
3.3	Možnosti rozšiřování vývojového prostředí	16
3.4	Dostupné aplikační programové rozhraní	17
3.5	Alternativní dostupná řešení	17
4	Návrh jazykového klienta pro Apache Camel	19
4.1	Návrh architektury	19
4.2	Vlastnosti zásuvného modulu	20
4.3	Uživatelské rozhraní	20
5	Implementace zásuvného modulu	22
5.1	Postup vytváření jazykového klienta	22
5.2	Problém a jeho řešení	24
5.3	Publikace zásuvného modulu	27
6	Ověření funkcionality	28
6.1	Testování jazykového klienta	28
6.2	Srovnání s vestavěným jazykovým klientem	33
7	Závěr	34
	Literatura	35

Kapitola 1

Úvod

V době nepřehledného množství textových editorů a integrovaných vývojových prostředí je třeba zajistit komplexní podporu širokého množství programovacích jazyků. Taková podpora zahrnuje například rozšířené funkce editorů jako je zvýrazňování syntaxe, dokončování kódu a jiné. Aby bylo možné implementaci těchto funkcí usnadnit, byl vyvinut *Language Server Protocol*. *Language Server Protocol* umožňuje zjednodušení celého procesu do implementace jednoho jazykového serveru (*Language Server*) pro právě jeden programovací jazyk, který zajistí podporu daných funkcí za užití volání vzdálené procedury (*remote procedure call*). Pro implementaci na straně klienta je pak jen nutné zajistit komunikaci s jazykovým serverem pro poskytnutí jazykové podpory. Hlavním cílem bakalářské práce je navrhnout a implementovat jazykového klienta pro integrované vývojové prostředí *Apache NetBeans*.

1.1 Motivace

Již existující jazykový server vytvořený společností *Red Hat* pro *Apache Camel* [1] aktuálně nabízí podporu pro integrovaná vývojová prostředí *Eclipse*, *Microsoft Visual Studio Code*, *Eclipse Che* a *Atom*. Vytvoření jazykového klienta pro komunitní vývojové prostředí *Apache NetBeans* zajistí širší dostupnost pro uživatele. Zhotovení klienta povede k následnému zveřejnění v *Apache NetBeans Plugin Portalu* a nasazení v praxi.

Kapitola 2

Přehled souvisejících technologií

V dnešní době jsou technologie neustále se rozvíjejícím prvkem každodenního života. Nové a inovativní technologie se objevují každým dnem a pomáhají řešit různé problémy. Účelem této kapitoly je popsat technologie související s klientskou aplikací pro jazykový server *Apache Camel*. Kapitola se dále věnuje popisu samotného integračního rámce *Apache Camel*.

2.1 Language Server Protocol

Language Server Protocol (neboli *LSP*) [5] je otevřený standardizovaný protokol pro komunikaci mezi textovým editorem či integrovaným vývojovým prostředím (*IDE*) a serverem poskytujícím podporu pro rozšíření vlastností editoru pro daný programovací jazyk. Protokol byl původně vyvíjen společností *Microsoft* pro užití v *Microsoft Visual Studio Code*. V roce 2016 společnost *Microsoft* oznámila spolupráci s firmami *Red Hat* a *Codenvy* za účelem standardizace protokolu.

Mezi typická rozšíření vývojových prostředí patří například zvýrazňování syntaxe, nápověda a dokončování kódu, formátování kódu, odkazování a přechody na definice či zpracování upozornění a chyb. Implementace těchto vlastností do jednotlivých textových editorů a vývojových prostředí je náročná na zdroje, zejména na čas. Použití různých aplikačních programových rozhraní (*API*) v jednotlivých prostředích znemožňuje snadné integrování totožných funkcí. Hlavní myšlenkou je standard, který přesouvá veškerou logiku daných funkcí na stranu jazykového serveru. Ta je následně z editoru zpřístupněna pomocí *LSP* protokolu.

2.1.1 Princip funkce protokolu

LSP je provozován jako separátní (samostatně běžící) proces, během kterého editor komunikuje s jazykovým serverem za použití bezstavového protokolu *JSON-RPC*. Obrázek 2.1 znázorňuje komunikaci graficky. Komunikace probíhá následovně:

1. Uživatel otevřel soubor ve svém editoru.

Editor upozorní jazykový server, že došlo k otevření dokumentu pomocí notifikace `textDocument/didOpen`. Od tohoto okamžiku není obsah dokumentu pod správou souborového systému počítače, ale je uložen ve vnitřní paměti editoru, kde dochází k synchronizaci mezi editorem a jazykovým serverem za užití *LSP*.

2. Uživatel provedl změnu.

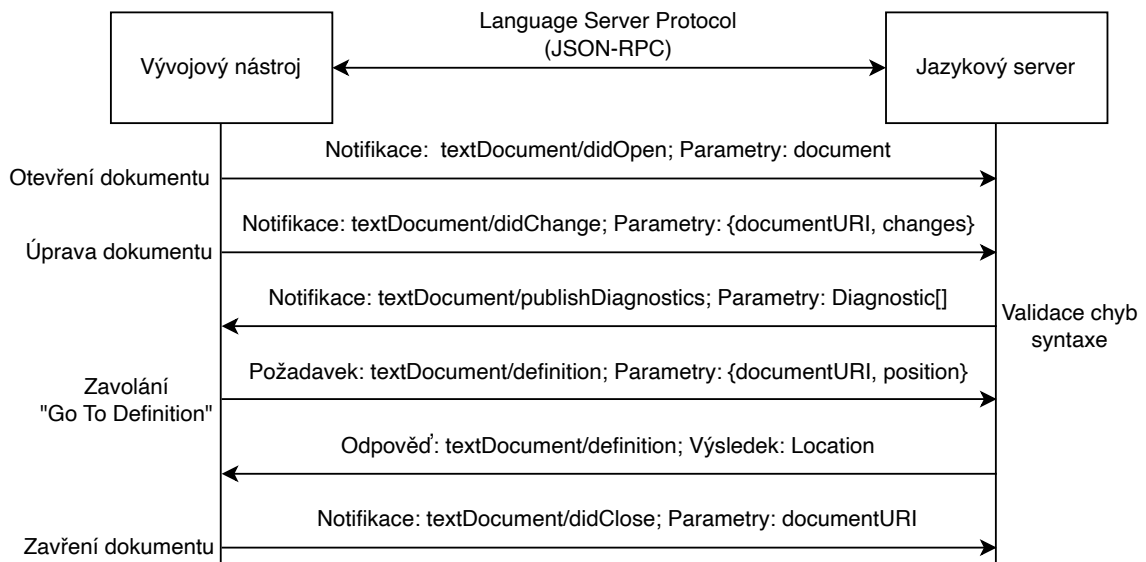
Editor upozorní jazykový server pomocí notifikace `textDocument/didChange`. Jazykový server provede analýzu nově obdržených informací a editoru odpovídá pomocí notifikace `textDocument/publishDiagnostics` obsahující diagnostiku včetně případných upozornění a chyb.

3. Uživatel vyžádal přechod na definici (definující část) symbolu.

Editor odešle notifikaci `textDocument/definition`, která přijímá dva parametry. Prvním parametrem je `documentURI`, nesoucí *URI* odkazovaného dokumentu. Druhý parametr `position` udává konkrétní pozici definice uvnitř souboru odkazovaného v *URI*. Editor jako odpověď na daný požadavek obdrží lokaci požadované definice.

4. Uživatel zavřel dokument.

Notifikace `textDocument/didClose` je zaslána z editoru na jazykový server. Ten je informován o uzavření dokumentu a tedy i o tom, že není třeba jej dále držet v paměti. Na zařízení se uloží poslední úpravy z paměti do souborového systému.



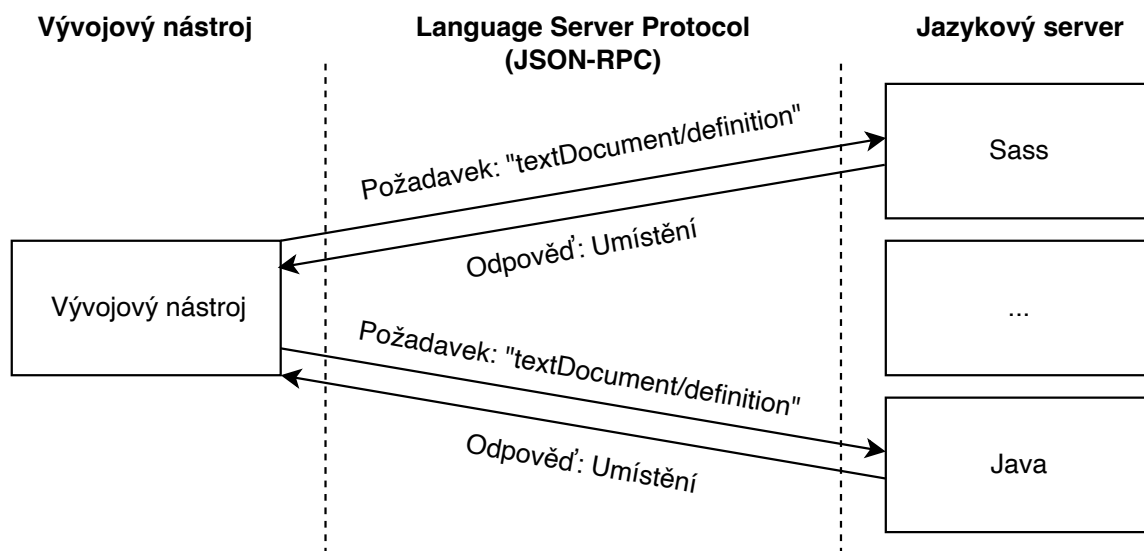
Obrázek 2.1: Grafické znázornění komunikace mezi editorem a jazykovým serverem. Převezato z [6].

2.1.2 Schopnosti protokolu

Ne každý jazykový server dokáže podporovat všechny funkce definované protokolem. *LSP* proto poskytuje výčet podporovaných funkcí. Schopnost (*Capability*) sdružuje sady podporovaných funkcí. Editor a server si oznamují podporované funkce právě použitím capabilities. Např. server oznamuje, že dokáže zpracovat požadavek `textDocument/definition` (přechod na definici), ale nedokáže zpracovat požadavek `workspace/symbol` (vyhledávání symbolů). Editor tak ví, které požadavky má smysl na server odesílat ke zpracování a které ne. Stejně tak je možné, že editor oznámí své schopnosti poskytovat notifikace typu `about to save`. Server tak může zpracovávat kód průběžně, ještě než je uložen a požadavek odeslán.

2.1.3 Současná práce s více jazyky

Když uživatel pracuje zároveň s více jazyky, editor naváže komunikaci s jazykovým serverem pro každý jazyk zvlášť. Obrázek 2.2 graficky znázorňuje proces komunikace s více jazykovými servery.



Obrázek 2.2: Schéma komunikace editoru a serveru při práci s více jazyky. Převzato z [6].

2.1.4 Specifikace protokolu

Základní protokol je tvořen hlavičkou a obsahem. Protokol je podobný *HTTP*. Hlavička a obsah jsou odděleny pomocí `\r\n`. V této práci bude použita verze protokolu 3.17, která je poslední dostupnou (kontrolováno dne 15.04.2023).

Hlavičková část

Skládá se z dvou hlavičkových polí. Každé pole je složeno ze jména a hodnoty oddělené `' : '` (dvojtečka následovaná mezerou). Struktura hlavičkových polí vyhovuje *HTTP* sémantice. Každé pole je ukončeno pomocí `\r\n`. Poslední pole v hlavičce je povinné. To znamená, že obsahové části zprávy vždy předcházejí dvě sekvence ukončovacích znaků `\r\n`. Tabulka 2.1 popisuje současně podporovaná pole.

Typ hlavičky	Datový typ	Popis
Content-Length	číslo	Délka obsahu v bytech. Toto pole je povinné.
Content-Type	řetězec	Samotný obsah. Implicitně nastaven na <code>application/vscode-jsonrpc; charset=utf8</code> .

Tabulka 2.1: Výčet současně podporovaných hlavičkových polí.

Obsahová část

Obsahuje vlastní obsah zprávy. Zpráva používá protokol *JSON-RPC*, aby popsala objekt, kterým může být požadavek, odpověď či notifikace. Více viz kapitola 2.2 věnující se *JSON-RPC*. Kódování obsahové části je definováno v poli `Content-Type`. Implicitně je využíváno kódování `utf-8`, které je jediné podporované v *LSP*. V případě, že klient nebo server obdrží požadavek s rozdílným kódováním, končí požadavek chybou. Starší verze používaly chybné označení – `utf8`. Správně implementovaný klient i server by měly toto označení kódování považovat za totožné s `utf-8`.

```
1 Content-Length: ... \r\n
2 \r\n
3 {
4   "jsonrpc": "2.0",
5   "id": 1,
6   "method": "textDocument/completion",
7   "params": {
8     ...
9   }
10 }
```

Výpis 2.1: Ukázka obsahu zprávy.

2.1.5 Architektura protokolu

Architekturu lze popsat několika základními prvky sloužícími pro komunikaci mezi editorem a jazykovým serverem. Prvky jsou následující:

- **Language Client** – Zprostředkovává komunikaci editoru a jazykového serveru.
- **Language Server** – Zpracovává požadavky klienta, komunikuje s editorem, využívá svých vlastností pro adekvátní odpověď klientovi.
- **Language Server Protocol** – Samotný protokol sloužící pro přenos dat. Pro přenos využívá *JSON-RPC*.

2.2 JSON-RPC

Je bezstavový protokol využívající vzdáleného volání procedur (*remote procedure call*, neboli *RPC*). To znamená, že požadovaná procedura je vykonána v jiném adresovém prostoru, ale zároveň vývojář nemusí explicitně řešit implementaci vzdálené integrace. Vytváří stejný kód bez ohledu na to, zda je daná interakce obsluhována lokálně nebo vzdáleně. Samotný protokol *JSON-RPC* byl navržen tak, aby byl co nejjednodušší. Obsahuje tedy pouze několik datových typů a operací nad nimi.

V rámci práce budeme pracovat s verzí protokolu *JSON-RPC 2.0*, která je poslední dostupnou (kontrolováno dne 15.04.2023),

2.2.1 Vlastnosti JSON-RPC

Protokol podporuje odesílání požadavků, odpovědi, události a chyby. Každý z těchto datových typů je pojmán jako objekt. Klient (editor) obvykle požaduje pouze jednu metodu ze vzdáleného serveru. Více vstupních parametrů je možné předávat serveru jako pole (array). Vzdálená metoda může podobným způsobem vracet více výstupních dat.¹ Objekty je možné serializovat (skládat za sebe) tak, jak umožňuje protokol *JSON*.

2.2.2 Objekty v JSON-RPC

Požadavek

Volání specifické metody poskytnuté vzdáleným serverem. Objekt obsahuje parametr `jsonrpc`, který určuje verzi *JSON-RPC* protokolu. Dále může obsahovat až tři další členy:

- **method** – Řetězec obsahující jméno metody, která má být vyvolána. Nesmí začínat předponou `rpc`. Takové metody jsou rezervovány pro vnitřní užití v rámci protokolu.
- **params** – Objekt nebo pole s hodnotami, které jsou předávány volané metodě. Pole musí obsahovat prvky v pořadí, které je očekávané druhou stranou. Objekt musí obsahovat přesný název, který očekává druhá strana. Použití tohoto parametru není vyžadováno.
- **id** – Řetězec nebo celé číslo užívané k identifikaci a provázání odpovědi a požadavku. Není-li očekávaná žádná odpověď, lze tento parametr vynechat.

Odpověď

Reakce na obdržené volání specifické metody. Objekt obsahuje parametr `jsonrpc`, který určuje verzi *JSON-RPC* protokolu. Dále může obsahovat až tři další členy:

- **result** – Data odpovídají návratové hodnotě vyvolané metodou. V případě úspěšného vykonání je formátován jako *JSON* objekt. V případě, že během zpracování došlo k chybě, musí tento člen zůstat nepoužitý.
- **error** – Návratová hodnota je objekt příslušné chyby. Viz tabulka 2.2. Dojde-li během zpracování požadavku k chybě, musí být tento člen přítomný. Nedojde-li k chybě během zpracování, není přítomnost členu nutná.
- **id** – Hodnota odpovídající `id` obdrženého požadavku.

Notifikace

Z hlediska protokolu se jedná o objekt požadavku, jeho parametr `id` je ale vždy `NULL`. Na tento typ objektu není očekávaná odpověď, není tedy přítomnost `id` vyžadována.

Chyba

Objekt chyby je struktura obsahující informace o chybě, která se vyskytla při zpracování požadavku. Tento objekt se používá k informování klienta o chybě, která se vyskytla v LSP serveru, aby mohl klient provést nápravné kroky.

¹Není dostupné ve starších verzích.

- **code** – Obsahuje celočíselnou hodnotu nesoucí hodnotu příslušné chyby. Tabulka 2.2 obsahuje výčet jednotlivých chybových kódů.
- **message** – Řetězec obsahující chybové hlášení.
- **data** – Podrobnější popis chyby. Parametr není povinný.

kód	zpráva	význam
-32700	Chyba syntaktické analýzy.	Server obdržel neplatný <i>JSON</i> . Chyba nastala během syntaktické analýzy na serveru.
-32600	Neplatný požadavek.	Odeslaný <i>JSON</i> není platným objektem Požadavku.
-32601	Metoda nebyla nalezena.	Požadovaná metoda neexistuje nebo není dostupná.
-32602	Neplatný parametr.	Neplatný parametr požadované metody.
-32603	Vnitřní chyba.	Vnitřní chyba protokolu <i>JSON-RPC</i> .
-320xx	Chyba serveru.	Kódy vyhrazené pro chyby definované implementovaným serverem.

Tabulka 2.2: Přehled chybových kódů protokolu *JSON-RPC*.

2.2.3 Hromadné volání

Protokol umožňuje hromadné volání metod. V takovém případě klient odesílá pole naplněné objekty požadavků. Server dané požadavky zpracuje, odpovědi plní do pole a po zpracování všech požadavků vrací pole odpovědí. Server požadavky může zpracovat v libovolném pořadí. To znamená, že pole odpovědí je naplněno také v libovolném pořadí. Klient musí jednotlivé odpovědi propojit na základě *id* požadavku.

V případě že dojde k chybě zpracování během hromadného volání, např. že požadavek není rozpoznán jako validní *JSON* nebo pole s alespoň jednou hodnotou, odpověď musí být pouze jeden objekt odpovědi. Pokud není server schopen zpracovat ani jeden z požadavků a pole odpovědí je tedy prázdné, nesmí toto pole server vrátit. V takovém případě není navraceno nic.

2.3 Apache Camel

Apache Camel je open-source integrační rámec (*framework*) vytvořený v jazyce *Java*. Je založený na *Enterprise Integration Patterns* popsaných v knize od Gregora Hohpe a Bobbyho Woolfa [4]. Základní koncept je tvořen integrací a interakcemi mezi různými aplikacemi, pro které *Camel* poskytuje trasování (routing), transformování a monitorování. *Camel* narozdíl od *ESB (Enterprise Service Bus)* nepodporuje kontejnery nebo spolehlivé zprávové sběrnice (*message bus*). Lze na jeho základě ale vytvořit kompletní integrační platformy jako je například *JBoss Fuse* [10] nebo *Apache ServiceMix* [13]. *Camel* užívá modulární architekturu, která umožňuje implementaci a použití zásuvných modulů pro podporu nových protokolů. Díky použití tohoto designu je architektura *Apache Camel* rychlá a snadno rozšiřitelná pro vývojáře.

2.3.1 Vlastnosti

Camel k zápisu integračních cest užívá stanovené konvence, které jsou dány doménově specifickým jazykem (*Domain Specific Language - DSL*) v deklarativní podobě. Nejčastější formou zápisu dané cesty je užití programovacího jazyka *Java*, který umožňuje za použití frameworků *Spring* [19] (ukázka viz výpis 2.2), *Blueprint* [16] nebo *Scala* [18] definovat konfigurační soubory *XML*. Dále je možné použít definici pomocí běžného kódu *Java* (*Java DSL* [17]) bez nutnosti použití konfiguračních souborů. Oba druhy zápisu jsou téměř totožné. *Blueprint* se typicky nasazuje v rámci *OSGi* (*Open Services Gateway initiative*) [3].

```
1 <routes xmlns="http://camel.apache.org/schema/spring">
2   <route>
3     <from uri="timer:tick"/>
4     <setBody>
5       <constant>Hello World!</constant>
6     </setBody>
7     <to uri="log:info"/>
8   </route>
9 </routes>
```

Výpis 2.2: Ukázka syntaxe pro *Spring DSL*.

2.3.2 Trasování

Základní vlastností *Apache Camel* je jeho trasovací a zprostředkovací engine (*routing and mediation engine*). Trasovací engine přesouvá data z jedné lokace do jiné. Při tomto přesunu využívá nakonfigurovaných cest. Uživatel si může pro svoje trasování nadefinovat vlastní pravidla, přidávat metody pro zpracování a modifikaci dat a data filtrovat. Na konci cesty je také potřeba zvolit cílovou lokalitu, kam budou umístěna výstupní data.

2.3.3 Transformace

Transformace (přeměna vstupní informace na výstupní) přes definovanou cestu probíhá po objektech zprávy.

Objekt Zpráva

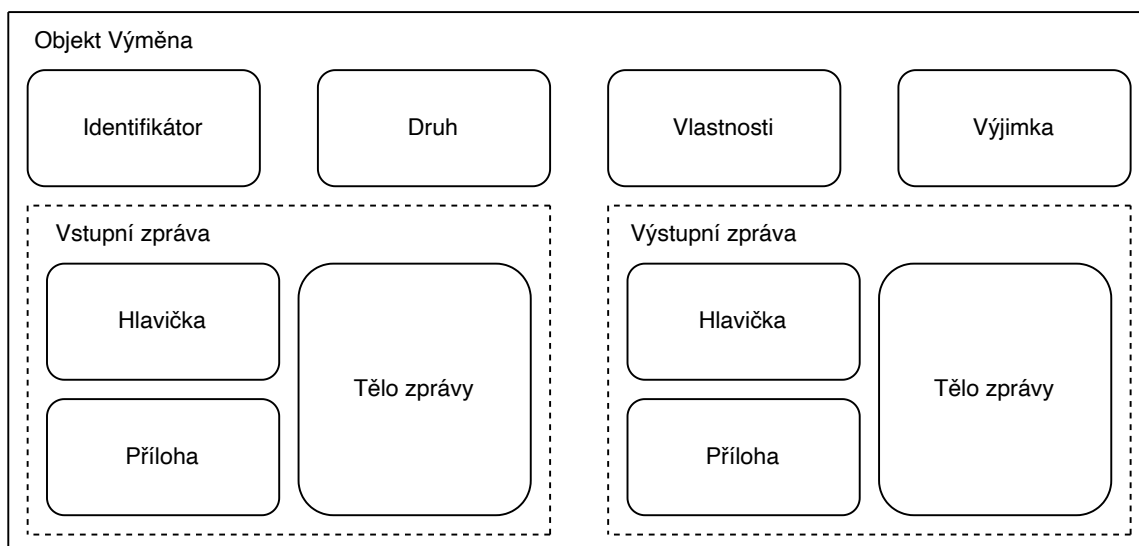
Objekt *Zpráva* (*Message*) [`org.apache.camel.Message`] reprezentuje data, která jsou používána systémem ke vzájemné komunikaci. Komunikace probíhá od odesílatele k příjemci. Objekt *Zprávy* se skládá z hlavičky, těla zprávy a nepovinných příloh. Všechny *Zprávy* jsou identifikovány unikátním identifikátorem. Tvar identifikátoru je definován konvencí protokolu. V případě, že protokol tvar identifikátoru nedefinuje, je užitá výchozí konvence integračního rámce.

Hlavička *Zprávy* je obdobná hlavičce protokolu *HTTP*. Obsahuje informace o odesílateli, kódování, typ obsahu a autentizační parametry. Tělo reprezentuje obsah samotné zprávy. Typicky se jedná o *Java* objekt, zpráva může uložit jakýkoliv typ obsahu, pouze by měla být zajištěna schopnost příjemce danou zprávu akceptovat.

Objekt Výměna

Objekt *Výměny* je definován jako kontejner zapouzdřující objekt *Zprávy* během jeho cesty. Mezi objektem *Výměny* a systémem probíhá množství interakcí. Grafické znázornění objektu je popsáno obrázkem 2.3. Objekt obsahuje:

- **ID výměny** (*Exchange ID*) – Unikátní identifikátor dané výměny, může být explicitně daný, jinak je automaticky generovaný rámcem *Apache Camel*.
- **Message Exchange Pattern** (*MEP*) – Definuje styl zprávy.
- **Výjimka** (*Exception*) – Pro případ že by během cesty nastala chyba.
- **Vlastnosti** (*Properties*) – Obdoba hlavičky v objektu *Zprávy*, vlastnosti jsou ale narozdíl od hlavičky neměnné po celou dobu trvání výměny.
- **Vstupní zpráva** – Povinná vstupní zpráva obsahující požadavek.
- **Výstupní zpráva** – Nepovinná zpráva obsahující odpověď (pouze pro *Message Exchange Pattern* typu *InOut*).



Obrázek 2.3: Znázornění objektu *Výměny* a jeho komponent.

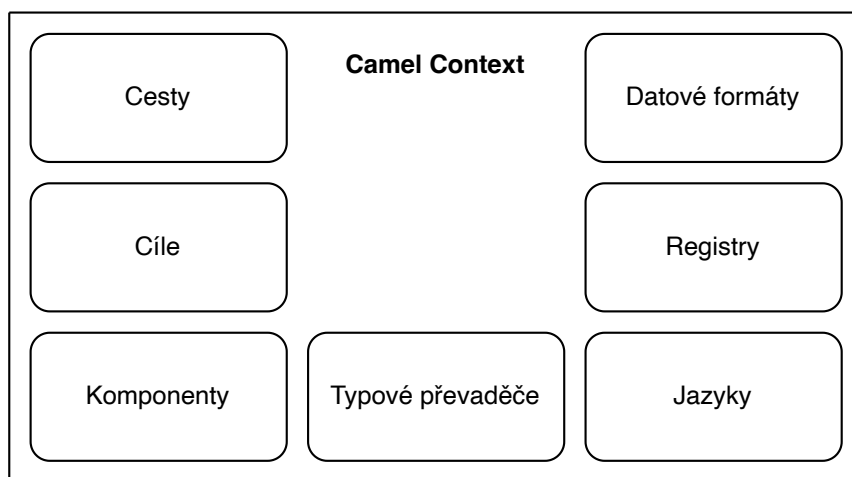
2.3.4 Architektura

Architektura *Apache Camel* je založena na *Camel Context*, který propojuje prvky architektury běhového systému.

Camel Context

Obdoba kontejneru určená pro běhové prostředí (*runtime*) systému *Camel*. Kontext udržuje vše pohromadě a spravuje následující služby během spuštěného běhového prostředí (graficky viz obrázek 2.4):

- **Cesty** (*Routes*) – nadefinované cesty mezi komponenty přidané do kontextu.
- **Cíle** (*Endpoints*) – Cíle jednotlivých cest vytvořených v kontextu.
- **Komponenty** (*Components*) – Komponenty používané aplikacemi.
- **Typové převaděče** (*Type Converters*) – Možné změny datového typu načtené kontextem.
- **Datové formáty** (*Data formats*) – Datové formáty načtené do kontextu.
- **Jazyky** (*Languages*) – *Camel* podporuje různé jazyky použité ve výrazech, ty jsou načteny do kontextu.
- **Registry** (*Registers*) – Implicitně je užitý *Java Naming and Directory Interface* [9], v případě specifických požadavků technologie může být použit i jiný.



Obrázek 2.4: Jednotlivé prvky architektury, ze kterých je složený rámec *Camel Context*.

2.3.5 Komponenty

Jedním z klíčových prvků *Apache Camel* jsou komponenty. Komponenty v *Apache Camel* jsou moduly reprezentující systémy a protokoly, které lze integrovat dohromady. Každá komponenta má specifické rozhraní pro komunikaci s daným systémem a umožňuje snadné propojení s *Apache Camel*. Komponenty mohou být použity jak pro odesílání, tak pro příjem zpráv. Každá komponenta má svůj vlastní konfigurační soubor, který umožňuje nastavit vlastnosti pro daný systém. Například konfigurace komponenty pro práci s databázemi umožňuje nastavit připojení k databázi, *SQL* dotazy a další vlastnosti. V současné době *Apache Camel* podporuje více než 300 různých komponent.

Komponenty v *Apache Camel* lze použít pro vytváření integračních cest, které umožňují propojovat různé systémy a aplikace. Jednotlivé cesty jsou tvořeny soubory komponent. Tyto cesty lze vytvářet pomocí konfiguračních souborů nebo v kódu.

2.3.6 Katalogy komponent

Katalogy komponent v *Apache Camel* jsou soubory, které obsahují seznam komponent, které jsou dostupné pro použití. Tyto soubory jsou obvykle ve formátu *XML* nebo *JSON* a obsahují informace o každé komponentě. Mezi tyto informace patří například název, třída, verze a další. Katalogy komponent jsou užitečné pro vývojáře, kteří chtějí použít *Apache Camel* pro integraci s konkrétním systémem. Díky tomu je možné provést snadnou instalaci a konfiguraci potřebných komponent. Katalogy komponent jsou součástí *Apache Camel* a jsou distribuovány společně s ním. Existují také externí katalogy komponent, které obsahují komponenty vyvinuté třetími stranami a které lze použít s *Apache Camel*. Katalogy se mezi jednotlivými verzemi *Apache Camel* mohou lišit.

2.3.7 Běhová prostředí

Běhové prostředí *Apache Camel* je softwarová komponenta umožňující běh integračních cest a řešení. Existuje mnoho různých poskytovatelů běhového prostředí pro *Apache Camel*. Vzájemně se liší svými vlastnostmi a použitím. Díky tomu lze optimalizovat běh *Camel* aplikací pro konkrétní použití, prostředí a scénáře. Nejpoužívanější běhová prostředí jsou:

Spring Boot

Spring Boot je běhové prostředí, které je založeno na *Spring Frameworku* [19]. Prostředí poskytuje všechny výhody a funkce *Spring Frameworku* a umožňuje snadnou integraci s jinými *Spring* projekty. Jeho konfigurace je snadná, lze k tomu použít *Spring XML* nebo *Java* konfigurace. Běhové prostředí *Spring Boot* je velmi flexibilní a vhodné pro větší a složitější projekty.

Karaf

Karaf je běhové prostředí, které je založeno na *Apache Karaf* [21]. *Karaf* využívá *OSGi* [3] kontejnerů a umožňuje snadnou správu modulů a závislostí v běhu aplikace. Jeho architektura je modulární. V rámci běhového prostředí jsou integrovány další technologie a frameworky jako například *Apache CXF* [20] nebo *Apache ActiveMQ* [14]. Běhové prostředí *Karaf* je vhodné pro velké a složité projekty, kde je potřeba spravovat mnoho modulů a závislostí.

Quarkus

Quarkus je běhové prostředí založené na open-source frameworku pro kontejnerizované mikroslužby. Jednou z nejvýraznějších vlastností běhového prostředí *Quarkus* je jeho rychlost a nízká paměťová náročnost, což umožňuje běh *Camel* aplikací v malých kontejnerech a nízkou latenci. Dále umožňuje snadnou kombinaci různých technologií v jedné aplikaci a poskytuje vývojářům mnoho možností pro vytváření moderních cloudových a bezserverových aplikací. Běhové prostředí *Quarkus* podporuje různá prostředí, jako jsou *Kubernetes* [23], nebo *OpenShift* [11]. Běhové prostředí je vhodné pro vytváření efektivních, škálovatelných a výkonných mikroslužeb, které jsou nasazeny v kontejnerech.

2.3.8 Apache Kafka

Apache Kafka je open-source streamovací platforma navržená pro zpracování a ukládání velkého množství dat v reálném čase. *Kafka* se zaměřuje na vysokou spolehlivost, škálovatelnost a nízkou latenci a je vhodná pro zpracování dat z různých zdrojů, jako jsou například senzory, aplikace, webové stránky a další. *Kafka* se často využívá pro zpracování velkých objemů dat v aplikacích pracujících v reálném čase. *Kafka* používá *publish-subscribe* vzorec, kde producenti zapisují data do *topiců*² a konzumenti je odebírají k dalšímu zpracování.

2.4 Jazykový server pro Apache Camel

Implementace jazykového serveru pro *Apache Camel* byla vytvořena společností *Red Hat* za účelem poskytnutí jazykové podpory. Jedná se o open-source projekt, jehož zdrojový kód je volně dostupný na *GitHub* repozitáři [1]. Jazykový server je implementovaný v jazyce *Java*, využívá knihovny *LSP4J* a *LSP4E* a jeho provozování je nezávislé na cílové platformě.

Poslední dostupnou verzí ke dni 16.4.2023 je verze 1.9.1. Vytvořený jazykový klient bude podporovat všechny současně podporované funkční vlastnosti. Mezi ty patří:

- Dokončování kódu pro jednotné identifikátory zdroje (*URI*) *Camel*. To zahrnuje dokončování *Camel* komponent, atributů a jejich hodnot.
- Zobrazování dokumentace pro jednotlivé *Camel* komponenty.
- Diagnostika a validace jednotných identifikátorů zdroje *Camel*.
- Použití specifické verze katalogu komponent *Camel*.
- Použití specifického poskytovatele běhového prostředí katalogu komponent *Camel*.
- Přidání vlastních komponent *Camel*.
- Propojení pro dokončování komponent *Apache Kafka*.

Jazykový server pro *Apache Camel* stále prochází vývojem. Následující funkční vlastnosti jsou plánované a budou implementovány v budoucnu:

- Pokročilé dokončování kódu.
- Zobrazení struktury zdrojového kódu.
- Navigace ve zdrojovém kódu.
- Reference.
- Zvýrazňování.
- Formátování kódu.

²Topic je kanál pro ukládání a přenášení zpráv, které jsou organizovány do oddělených a nezávislých toků dat.

Kapitola 3

Vývojové prostředí Apache NetBeans

Apache NetBeans je open source integrované vývojové prostředí (*IDE*) vytvořené firmou *Oracle*. Vzniklo v roce 1996 pod názvem *Xelfi*. V současné době je pod správou *Apache Software Foundation*. Primárně je určeno pro vývoj v programovacím jazyce *Java*. Jeho architektura ovšem umožňuje programování i v jiných jazycích jako je např. *PHP*, *HTML5*, *C/C++* či *JavaScript*. Samotné prostředí je vytvořeno v jazyce *Java*. Je možné jej spustit na operačních systémech *Windows*, *Linux*, *macOS* a *Solaris*. Díky modulární architektuře lze vývojové prostředí snadno rozšiřovat o další zásuvné moduly, případně integrovat s dalšími nástroji a technologiemi.

3.1 Architektura

Architektura vývojového prostředí *Apache NetBeans* je modulární. To umožňuje snadné rozšíření funkcí a integraci s dalšími nástroji a technologiemi. Hlavní části architektury *Apache NetBeans* jsou následující:

Platforma NetBeans

Tvoří základní část integrovaného vývojového prostředí a zajišťuje společné služby a infrastrukturu pro všechny moduly. Nejdůležitější součásti architektury platformy *Apache NetBeans* jsou:

- Jádru *Apache NetBeans*,
- sada knihoven,
- aplikační programové rozhraní (*API*),
- uživatelské rozhraní,
- systém pro správu a nastavení pluginů,
- nástroje pro správu a práci s projekty,
- a další funkce.

Moduly

Tvoří základní stavební jednotku. Poskytují specifické funkce, jakými může být podpora pro konkrétní jazyk, integrace s verzovacími systémy, editory kódu a další. Moduly jsou samostatné jednotky a mohou být do vývojového prostředí instalovány nebo z něj odebrány dle potřeby.

Zásuvné moduly (plug-iny)

Jsou externími moduly. Rozšiřují funkce a chování vývojového prostředí. Jsou tvořeny uživateli (třetími stranami) a stejně jako moduly je lze volně přidávat či odebírat z vývojového prostředí.

Uživatelské rozhraní

Uživatelské rozhraní je založeno na technologii *Swing*. Díky tomu lze vytvořit přizpůsobitelné a intuitivní uživatelské rozhraní. Díky modulární architektuře lze vzhled přizpůsobit i pomocí modulů a pluginů.

Podpora pro různé jazyky a technologie

Apache NetBeans poskytuje podporu pro mnoho jazyků a technologií, včetně *Java*, *PHP*, *C++*, *HTML*, *JavaScript*, *Ruby*, *Python* a dalších. Každý jazyk má svůj vlastní modul, který poskytuje specifické nástroje a funkce pro vývoj v tomto jazyce.

3.2 Modulární architektura

Modulární architektura je způsob návrhu softwaru, který umožňuje rozdělení aplikace na jednotlivé moduly, které jsou na sobě nezávislé. Díky tomu je možné jednotlivé moduly vyvíjet, testovat a nasazovat samostatně. Každý modul obsahuje sadu funkcí, které jsou vzájemně úzce spjaty a mohou být snadno nahrazovány nebo aktualizovány bez ovlivnění jiných modulů. Využití modulární architektury umožňuje vývojářům snadné rozšiřování o nové funkce či poskytovat aktualizace bez potřeby měnit jiné části aplikace. To také umožňuje přizpůsobit aplikace pro různé účely a různé cílové platformy.

Typické použití modulární architektury lze nalézt zejména v rozsáhlých aplikacích a systémech, např. operační systémy, databáze nebo právě vývojová prostředí. Architekturu lze implementovat různými způsoby. Nejběžnějším způsobem implementace využívá dynamického načítání či stahování pluginů, knihoven a modulů v době běhu aplikace. Mezi hlavní výhody modulární architektury patří:

- Snadná správa aplikace – moduly jsou nezávislé a mohou být aktualizovány samostatně.
- Rozšiřitelnost aplikace bez nutnosti měnit stávající kód.
- Optimalizace jednotlivých modulů pro cílové platformy.
- Možnost sdílet jednotlivé moduly mezi různými aplikacemi.
- Snadnější testování a ladění jednotlivých modulů.

Modulární architektura v NetBeans

Modulární architektura vývojového prostředí *Apache NetBeans* je jedním z klíčových prvků. Každá funkce v *Apache NetBeans* je implementovaná jako samostatný modul, který lze instalovat a odinstalovat dle potřeby. Jednotlivé moduly jsou navrhovány tak, aby byly vzájemně co nejméně závislé a mohly být použity v různých konfiguracích či kombinacích.

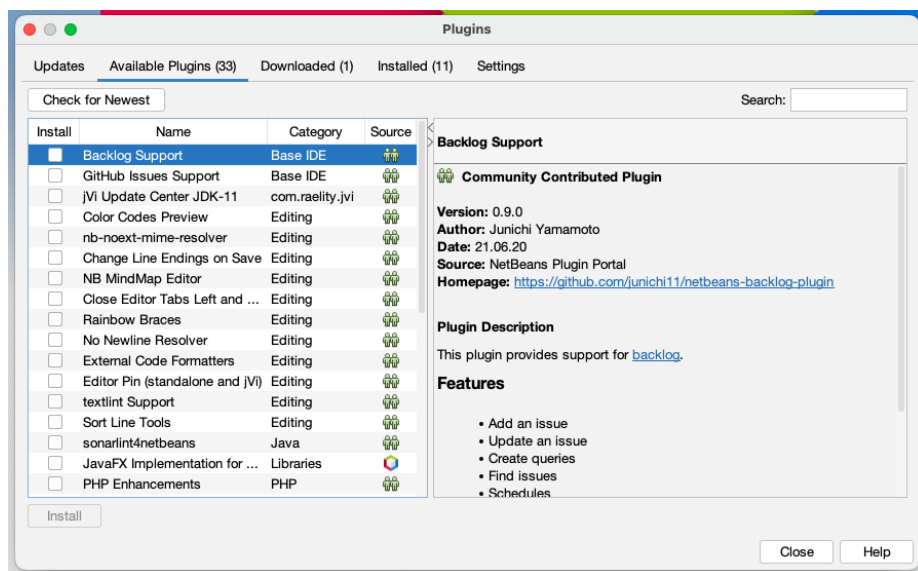
Moduly se skládají ze tříd, zdrojových kódů, konfiguračních souborů a případných dalších zdrojů. Zdroje jsou umístěny v samostatných adresářích a jsou sdružovány do archivů s příponou **JAR**. Zásuvné moduly určené pro vývojové prostředí *Apache NetBeans* mají příponu **NPM**.

3.3 Možnosti rozšiřování vývojového prostředí

Vývojové prostředí *Apache NetBeans* lze rozšířit více různými způsoby. K rozšíření je potřeba zvolit vhodný zdroj pro získání zásuvného modulu. Modul musí být kompatibilní se zvolenou verzí vývojového prostředí *Apache NetBeans*.

Instalace zásuvných modulů z centrálního repozitáře

Repozitář obsahující zásuvné moduly nese název *Apache NetBeans Plugin Portal*. Lze v něm libovolně vyhledávat, stahovat a instalovat již existující zásuvné moduly. Instalace je možná přímo z uživatelského rozhraní *Apache NetBeans* pomocí funkce správy zásuvných modulů, kterou lze najít pod záložkou **Tools** → **Plugins**. Repozitář je dostupný na adrese <https://plugins.netbeans.apache.org/> a ke dni 15.4.2023 obsahuje 69 zásuvných modulů, které mohou být instalovány. Pro verzi 17 vývojového prostředí *Apache NetBeans* je verifikováno 33 zásuvných modulů, které mohou být přímo instalovány. Správce zásuvných modulů zobrazuje obrázek 3.1.



Obrázek 3.1: Funkce správy zásuvných modulů.

Použití zásuvného modulu vytvořeného třetí stranou

Existuje mnoho zásuvných modulů, které se v oficiálním repozitáři nenachází. Pro instalaci takového modulu potřebujeme mít soubor s příponou `NBM` určenou pro verzi použitého vývojového prostředí. Instalaci je pak možné provést pomocí funkce správy zásuvných modulů. Dále je možné instalovat moduly *OSGi* [3] s příponou `JAR`.

Vytvoření vlastního zásuvného modulu

K tvorbě vlastního zásuvného modulu lze využít již existující aplikační programové rozhraní *Apache NetBeans*. Takový modul lze pak následně volně sdílet či zveřejnit na *Apache NetBeans Plugin Portalu*.

3.4 Dostupné aplikační programové rozhraní

Aplikační programové rozhraní ve vývojovém prostředí *Apache NetBeans* poskytuje způsob, jakým lze vytvářet jazykové klienty za použití jazyku *Java*. Aplikační programové rozhraní je implementováno za použití knihovny *LSP4J* [2]. *LSP4J* je open-source knihovna poskytující implementaci protokolu *LSP* pro integrovaná vývojová prostředí psaná jazykem *Java*, jako je například *Eclipse* nebo *Apache NetBeans*. Knihovna podporuje všechny funkce protokolu *LSP*. Knihovna je distribuovaná pod licencí *Eclipse Public Licence 2.0*¹. Správa knihovny je v režii aktivní komunity, která poskytuje nová vylepšení a aktualizace.

Implementovaný klient k navázání spojení využívá třídu `LanguageServerProvider` modulu `org.netbeans.modules.lsp.client.spi`. Pro práci s editorem kódu je k dispozici modul `org.netbeans.api.editor`, který obsahuje i metody podstatné pro inicializaci jazykového klienta při otevření podporovaného souboru – `MimeRegistration` a `MimeRegistrations`. Pro zpracování uživatelských preferencí je třeba implementovat vlastní třídu.

3.5 Alternativní dostupná řešení

Vývojové prostředí *Apache NetBeans* disponuje vestavěným jazykovým klientem s možností omezené konfigurace vlastního jazykového serveru. Při použití vestavěného klienta ale nelze specifikovat potřebné parametry spuštění jazykového serveru. Důsledkem toho není vestavěný klient s jazykovým serverem *Apache Camel* vytvořeným společností *Red Hat* kompatibilní. Přímé alternativní řešení tedy neexistuje. Vestavěného jazykového klienta lze použít dvěma způsoby popsanými níže.

3.5.1 Registrace vlastního jazykového serveru

Registrace vlastního jazykového serveru je možná po vyvolání průvodce nacházejícího se v `Options` → `Editor` → `Language Servers`. Průvodce umožňuje specifikovat přípony, pro které se má jazykový server spouštět. Kromě samotného jazykového serveru lze připojit soubor se syntaktickou gramatikou². Není možné ovšem specifikovat konkrétní instrukce pro spuštění jazykového serveru (jazykový server pro *Apache Camel* vytvořený společností *Red Hat* se spouští pomocí příkazu `java -jar`). Ke spuštění jazykového serveru tedy nedojde. Manuální spuštění jazykového serveru s následným propojením není možné.

¹Je zdarma k užití a může být upravována dle potřeb.

²Určuje pravidla, podle kterých jsou zapisovány programové konstrukce a výrazy v daném jazyce.

3.5.2 Propojení s jazykovým serverem

Druhým způsobem, jak lze vestavěného jazykového klienta propojit s vývojovým prostředím *Apache NetBeans*, je propojení pomocí **Connect to Language Server** z nabídky **Tools**. Tato možnost vyžaduje již spuštěný jazykový server komunikující na definovaném³ síťovém portu. Jazykový server *Apache Camel* vytvořený společností *Red Hat* touto možností nedisponuje. Pro případné propojení lze vytvořit prostředníka, který bude přesměrovávat vstupní a výstupní toky z běžícího jazykového serveru na síťový port. Takové řešení je uživatelsky velmi nepřívětivé. Implementace funkčních vlastností nad rámec povinného základu je nemožná.

³Číslo portu může být libovolné.

Kapitola 4

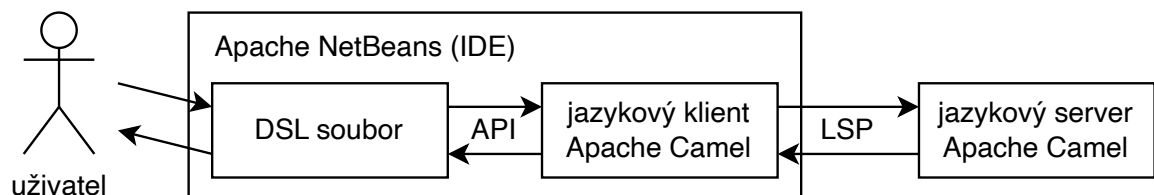
Návrh jazykového klienta pro Apache Camel

Návrh klienta je stěžejní součástí při procesu tvorby klientské aplikace. Kapitola se věnuje popisu klíčových prvků při návrhu samotného modulu včetně podporovaných funkčních vlastností vytvářeného jazykového klienta. Dále kapitola popisuje návrh uživatelského rozhraní.

4.1 Návrh architektury

Architektura zásuvného modulu se skládá ze dvou částí. První částí je samotná klientská aplikace pro jazykový server *Apache Camel*. Ta zajišťuje komunikaci mezi integrovaným vývojovým prostředím a jazykovým serverem *Apache Camel*. Druhou částí je samotný zásuvný modul pro vývojové prostředí. Zásuvný modul integruje vytvořenou klientskou aplikaci přímo do vývojového prostředí. Zároveň přidává uživatelské rozhraní pro správu a nastavení modulu.

Diagram 4.1 zobrazuje návrh architektury. Integrované vývojové prostředí (*IDE*) *Apache NetBeans* reprezentuje rozhraní pro editaci kódu. Uživatel otevře podporovaný soubor (libovolné *Camel DSL*¹) uvnitř vývojového prostředí. Vývojové prostředí pomocí aplikačního programového rozhraní (*API*) nejprve spustí instanci jazykového klienta, následně s ním za použití stejného rozhraní komunikuje. Jazykový klient naváže spojení s jazykovým serverem *Apache Camel*. Instance jazykového serveru je spuštěna jazykovým klientem po otevření podporovaného souboru, zásah uživatele není vyžadovaný. Jazykový klient a jazykový server spolu komunikují za užití *Language Server Protokolu* popsaného v kapitole 2.1.



Obrázek 4.1: Diagram zobrazující návrh architektury jazykového klienta.

¹Doménově specifický jazyk (Domain Specific Language), viz kapitola 2.3.1.

4.2 Vlastnosti zásuvného modulu

Jazykový klient byl navržen a implementován tak, aby využil veškerý potenciál funkcí, které jazykový server vytvořený společností *Red Hat* pro *Apache Camel* nabízí. Mezi funkcionality jazykového serveru patří:

- **Dokončování kódu pro jednotné identifikátory zdroje (URI) Camel** – Vývojové prostředí automaticky napovídá a dokončuje dostupné *Camel* komponenty, atributy komponent a jejich hodnoty na základě dotazů odesílaných na server.
- **Diagnostika a validace jednotných identifikátorů zdroje Camel** – Vývojové prostředí automaticky zaslá požadavek na server, který kontroluje, zda jsou vložené *Camel* komponenty, atributy komponent a jejich hodnoty validní.
- **Volba specifické verze katalogu komponent Camel** – Uživatel může specifikovat, jakou verzi katalogu komponent *Camel* má jazykový server použít.
- **Volba specifického poskytovatele běhového prostředí katalogu komponent Camel** – Uživatel může specifikovat, jakého poskytovatele běhového prostředí katalogu komponent *Camel* má jazykový server použít.
- **Přidání vlastních komponent Camel** – Uživatel může rozšířit katalog komponent o vlastní komponenty *Camel*.
- **Propojení pro dokončování komponent Apache Kafka** – Uživatel může specifikovat adresu běžící instance *Apache Kafka*, ze které jazykový klient přebírá informace o běžících topicích, které následně nabízí při automatickém dokončování kódu.
- **Použití specifického jazykového serveru** – Uživatel může specifikovat vlastní cestu k jazykovému serveru, který je použitý. Není-li vlastní cesta specifikovaná, jazykový klient automaticky inicializuje poslední dostupnou verzi jazykového serveru bez nutnosti zásahu uživatele.

Všechny funkce jsou podporovány pro doménově specifické jazyky *Java*, *XML* a *YAML*.

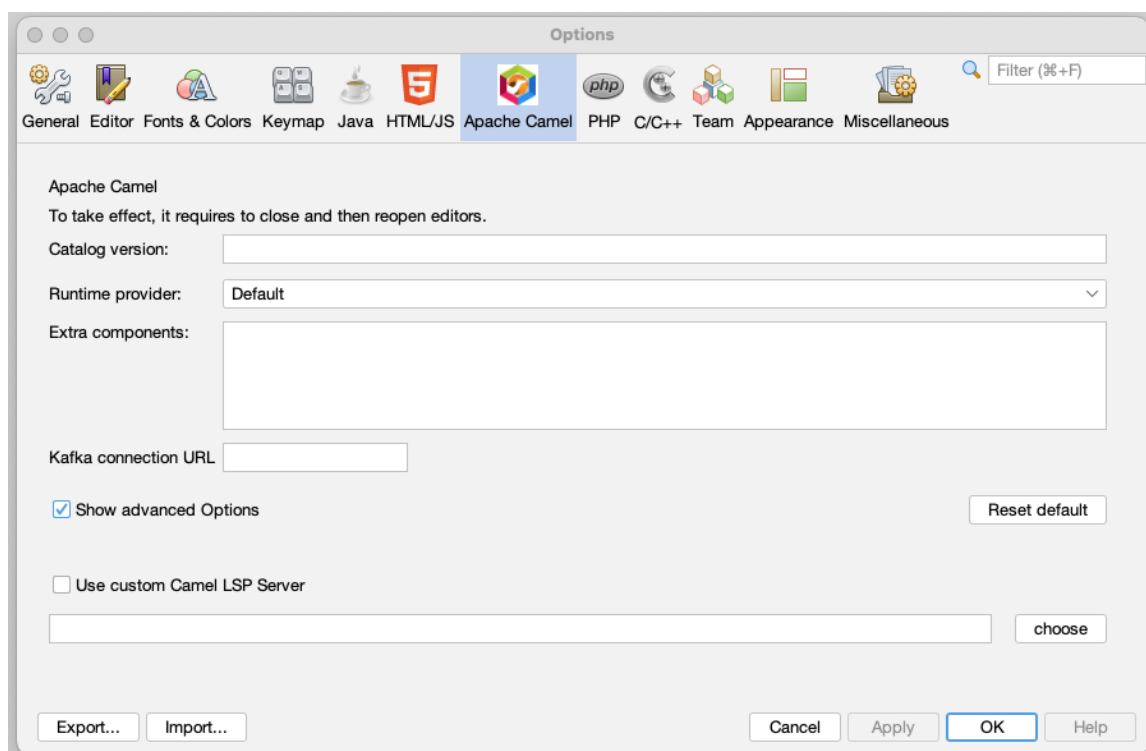
4.3 Uživatelské rozhraní

Samotný jazykový klient využívá výchozího uživatelského rozhraní *Apache NetBeans*. Pro cílového uživatele je nenápadný a poskytuje mu implicitní podporu. Návrh uživatelského rozhraní se tedy omezuje pouze pro nastavení klienta.

Při návrhu rozhraní bylo prioritou zachování intuitivnosti. V rámci vývojového prostředí lze za použití vhodného aplikačního rozhraní umístit nastavení na libovolné místo. Za nejvhodnější umístění je považováno přímo hlavní nastavení editoru. Pro modul bylo zvoleno nastavení v samostatné záložce, oddělené od předinstalovaných jazykových klientů. Vzhled uživatelského rozhraní zobrazuje obrázek 4.2. Samotné nastavení jazykového klienta je intuitivní a především nepovinné. Každý z atributů má své výchozí hodnoty.

Uživatel si může nastavit následující:

- **Verze katalogu s komponentami *Camel*** – Výchozí verzi stanovuje jazykový server, klient v takovém případě odesílá hodnotu `DEFAULT`.
- **Poskytovatel běhového prostředí katalogu komponent *Camel*** – Výchozí hodnotu definuje jazykový server, klient odesílá hodnotu `DEFAULT`. Uživatel má na výběr mezi *Spring Boot*, *Karaf* a *Quarkus*.
- **Přidání vlastní komponenty *Camel*** – Komponenty jsou vkládány jako validní *JSON* list.² Výchozí hodnota je `null`. Příklad viz výpis 6.2.
- **Adresu pro propojení s běžící instancí *Apache Kafka*** – Výchozí hodnota je `localhost:9092`, což je výchozí adresa *Apache Kafka*. Jazykový klient tedy hledá běžící instance i bez nutnosti zásahu uživatele.
- **Vlastní jazykový server** – Při zobrazení pokročilých možností si může uživatel definovat cestu k vlastnímu jazykovému serveru *Apache Camel*. Klient ve výchozím nastavení používá verzi 1.9.1.



Obrázek 4.2: Možnosti nastavení jazykového klienta.

²<https://camel.apache.org/manual/writing-components.html>

Kapitola 5

Implementace zásuvného modulu

Na základě vytvořeného návrhu jazykového klienta je možné vytvořit zásuvný modul s klientskou aplikací implementující jazykového klienta pro *Apache Camel*. Proces implementace lze rozdělit do několika fází, které tato kapitola popisuje. Kapitola dále popisuje i nalezený nedostatek v aplikačním programovém rozhraní a návrh jeho řešení.

5.1 Postup vytváření jazykového klienta

Prerekvizity

Pro vytvoření nového jazykového klienta pro *Apache NetBeans* je zapotřebí připravit lokální prostředí a mít k dispozici vývojové prostředí *Apache NetBeans*, manažer balíčků *Apache Maven* [22] a jazykový server včetně nezbytných informací pro jeho spuštění. Dále je vhodné mít k dispozici verzovací nástroj *Git* [12].

Založení nového projektu

Prvním krokem pro vytvoření jazykového klienta je založení nového projektu. Pomocí **File** → **New Project...** lze vyvolat průvodce pro založení projektu. Vývojové prostředí disponuje předdefinovanou šablonou pro tvorbu zásuvných modulů. Využívá k tomu manažera balíčku *Apache Maven*.

Možností nastavení projektu jsou triviální a lze je později změnit. Implementovaný jazykový klient je tvořen s využitím verze *Apache NetBeans 17.0*, která odpovídá označení `RELEASE170`.

Konfigurace založeného projektu

Samotné založení projektu pro tvorbu zásuvného modulu žádným způsobem nedefinuje jeho předurčení. Tvořený jazykový klient by měl být spouštěn pouze na vyžádání. Tedy v situacích, kdy je v editoru aktivně otevřený soubor, kterému náš server poskytuje jazykovou podporu. Na existujícím projektu lze pomocí předdefinované šablony takovou funkci vytvořit. Vyvoláním nabídky nad projektem **New** → **Other** → **Module Development** → **File Type** dojde k otevření průvodce pro automatické rozpoznávání typů souborů.

Nastavení je třeba věnovat zvýšenou pozornost, jeho pozdější změna je obtížná. K identifikaci typu souboru je využíván standard MIME [7]. Ten se skládá ze dvou částí – typu a podtypu. Tyto části jsou oddělené lomítkem. Pro tvořeného jazykového klienta používáme

typ `text`. Jazykový server *Apache Camel* pracuje celkem se třemi podtypy – *XML*, *Java* a *YAML*. Jazykový klient tedy bude rozpoznávat následující definice:

- `text/xml`,
- `text/java`,
- `text/yaml`.

Dále je třeba definovat, zda požadovaný typ souboru má specifickou příponu nebo je tvořen kořenovým elementem *XML*. Tvořený jazykový klient pro *Apache Camel* bude rozpoznávat přípony `xml`, `java` a `yaml`.

Po ukončení průvodce bude projekt obsahovat nové soubory. Kromě rozpoznání typu souboru lze tímto způsobem také do vývojového prostředí přidat zcela nový typ souboru. Soubor `xmlTemplate.xml` obsahuje šablonu, která by se vložila do nově vytvořeného souboru s daným typem. Soubor `package-info.java` obsahuje informace spojené právě s vložením této šablony. Protože v rámci implementace jazykového klienta šablony použity nebudou, nejsou tyto soubory relevantní a z projektu mohou být odstraněny. Stejný postup je třeba opakovat pro typy souborů `java` a `yaml`.

Třída s jazykovým klientem

Rozpoznávání typu souborů je nezbytnou součástí implementace jazykového klienta, samo o sobě ale nepřináší žádnou funkcionalitu. Aby bylo možné klienta implementovat, je třeba vytvořit novou třídu v projektu. Třidu je možné pojmenovat zcela libovolně.

Nově vytvořená třída bude implementovat `LanguageServerProvider` z aplikačního programového rozhraní knihoven *Apache NetBeans*. Třidu je třeba importovat. Dále je třeba importovat funkci `MimeRegistration` z programového rozhraní editoru *Apache NetBeans*.

`MimeRegistration` zajišťuje stěžejní část nově vznikajícího klienta. Propojí integrované vývojové prostředí s jazykovým klientem ve vhodné chvíli. Konkrétně v situaci, kdy je otevřen konkrétní soubor využívající definovaný *MIME*. Jazykový klient pro *Apache Camel* pracuje s třemi různými doménově specifickými jazyky. Takovou situaci je možné řešit dvěma způsoby. Pro každý doménově specifický jazyk mohou vytvořit samostatnou třídu nebo mohou využít třídu `MimeRegistrations`, která mi umožní zaštitit jednu třídu s jazykovým klientem více typy *MIME*.

Tvorba uživatelského rozhraní

Integrované vývojové prostředí *Apache NetBeans* využívá k tvorbě grafického prostředí nástrojovou sadu *Swing UI*. Příkladem užití ve vývojovém prostředí je například editor kódu, kde sada *Swing UI* zajišťuje zvýrazňování syntaxe. Vývojové prostředí *Apache NetBeans* obsahuje základní šablony pro tvorbu uživatelských rozhraní, které je možné použít.

Na existujícím projektu lze vyvolat průvodce pro využití šablon pomocí `New → Other → Module Development`. K dispozici je více druhů šablon. Pro implementaci uživatelského nastavení přímo v rozhraní vývojového prostředí je vhodná pouze šablona `Options Panel`. Průvodce je velmi intuitivní a jeho nastavení je snadné. Nastavení jazykového klienta pro *Apache Camel* je implementováno jako primární panel, tedy nastavení je zcela oddělené na samostatné záložce. Průvodce vygeneruje soubory `Panel.form`, `Panel.java`, `OptionsPanelController.java`.

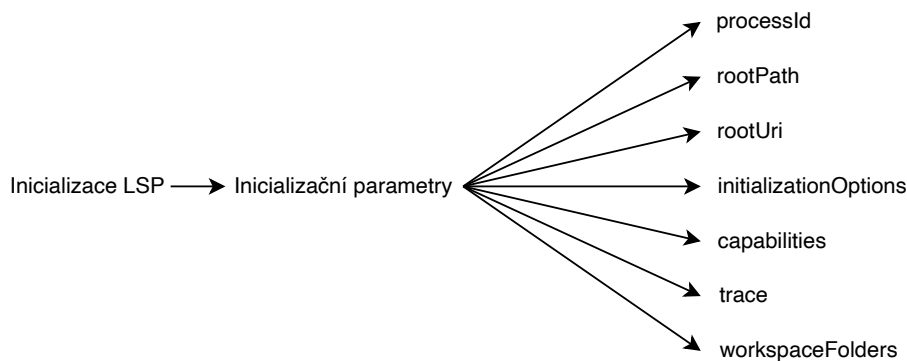
`Panel.form` a `Panel.java` reprezentují grafické rozhraní modulu. `Panel.java` lze otevřít pomocí grafického editoru `Design`. Uvnitř grafického editoru lze snadno navrhovat

uživatelské rozhraní. Soubor `Panel.form` je generován automaticky a není nutné do něj zasahovat. `OptionsPanelController.java` představuje třídu sdružující veškeré operace vykonané nad vytvořeným uživatelským rozhráním. Veškerá implementace spojená se tvorbou a zpracováním uživatelských preferencí se nachází zde.

5.2 Problém a jeho řešení

Přítomnost uživatelského nastavení je stěžejní přidanou hodnotou dedikovaného jazykového klienta. K předávání spouštěcích parametrů klienta dochází ihned po spuštění jazykového serveru. Parametry jsou předávány ve formátu *JSON-RPC* během první zprávy směřované na jazykový server. Složení první zprávy zobrazuje obrázek 5.1. V průběhu implementace těchto přidaných funkcionalit bylo nalezeno nedostatečné pokrytí ze strany aplikačního programového rozhraní *Apache NetBeans*. Po kompletním přezkoumání dokumentace i zdrojových kódů byl utvořen závěr, že tyto parametry se současnou implementací aplikačního rozhraní nelze serveru předat. Na dané téma byla otevřena v komunitě *Apache NetBeans* diskuze¹, která i po měsíci zůstala bez odpovědi.

Aplikační programové rozhraní *Apache NetBeans* využívá knihovnu *LSP4J*. Knihovna *LSP4J* poskytuje základní implementaci nízkourovňového aplikačního rozhraní *Language Server Protocolu* pro prostředí využívající jazyk *Java*. Implementace této knihovny obsahuje i funkci `setInitializationOptions`, která zajišťuje předání inicializačních parametrů jazykovému serveru. Pro zachování objektivnosti je třeba zmínit, že tyto parametry nejsou povinné a komunikaci lze úspěšně zahájit i bez jejich přítomnosti.² I přes to danou absenci považuji za hrubý nedostatek. Na základě daného zjištění jsem založil nový požadavek na rozšíření aplikačního rozhraní,³ včetně návrhu mého řešení, které je popsáno v následujících odstavcích.



Obrázek 5.1: Složení první zprávy odesílané na server.

Analýza problému

Pro možnost navržení alternativního řešení bylo nezbytné nejprve dostatečně pochopit způsob, jakým aplikační rozhraní *Apache NetBeans* pracuje s nízkourovňovou knihovnou *LSP4J*.

¹<https://github.com/apache/netbeans/discussions/5776>

²Funkce jsou v takovém případě omezené na povinné minimum.

³<https://github.com/apache/netbeans/issues/5822>

Komunikace probíhá následovně:

1. Při vytvoření nové instance jazykového klienta je volán konstruktor `LanguageServerDescription`. Konstruktor je součástí třídy `LanguageServerProvider` a akceptuje tři parametry – tok vstupních dat, tok výstupních dat a proces jazykového serveru.
2. Třída `LanguageServerProvider` při své inicializaci využívá třídy `LSPBindings`, která zajišťuje vazbu mezi jazykovým klientem a jazykovým serverem. První volanou metodou je metoda `getBindings`, která zajišťuje získání potřebných vazeb. Implementována je ve funkci `getBindingsImpl`. Ta volá metodu `buildBindings`, která vytváří vazbu mezi klientem a jazykovým serverem.
3. K předání parametrů pro inicializaci serveru dochází pomocí metody `initServer`, která je inicializována v průběhu metody `buildBindings`. V této části také chybí možnost přidání vlastních parametrů typu `initializationOptions`.
4. Jazykový server je spuštěn s předanými parametry.

Řešení

Stěžejní problém je tvořen skutečností, že nelze při inicializaci jazykového serveru metodou `initServer` předat vlastní parametry. Konstruktor dané metody jsem tedy rozšířil o další parametr `initOptions`. Parametr je typu `Object`, přijímá tedy libovolný objekt.

```
1 private static InitializeResult initServer(Process p, LanguageServer server,
    FileObject root, Object initOptions);
```

Výpis 5.1: Konstruktor funkce `initServer` s přidáním parametrem.

Nově přidáný parametr je uvnitř funkce `initServer` zpracován pouze je-li jeho hodnota odlišná od `null`, v opačném případě není hodnota `setInitializationOptions` inicializována.

```
1 private static InitializeResult initServer(...) {
2     InitializeParams initParams = new InitializeParams();
3     ...
4     if(initOptions != null){
5         initParams.setInitializationOptions(initOptions);
6     }
7     ...
8 }
```

Výpis 5.2: Zpracování parametru `initOptions`.

Metoda `initServer` je volána z metod `buildBindings` a `addBindings`. Druhá zmíněná metoda není pro rozšíření prostředí podstatná, pro zachování funkčnosti je pouze třeba doplnit čtvrtý parametr s hodnotou `null`.

```
1 public static void addBindings(..){
2     ...
3     InitializeResult result = initServer(null, server, root, null);
4     ...
5 }
```

Výpis 5.3: Rozšíření metody `addBindings`.

Metoda `buildBindings` je pro řešení podstatná. Tato metoda přebírá hodnoty z `LanguageServerProviderAccessor`, který zajišťuje instanci jazykového serveru. Tuto třídu bude také nezbytné upravit.

```
1 private static LSPBindings buildBindings(...) {
2     ...
3     try {
4         ...
5         Object initOpt = LanguageServerProviderAccessor.getInstance().
            getInitOptions(desc);
6         ...
7         InitializeResult result = initServer(p, server, dir, initOpt);
8         ...
9     }
10    ...
11 }
```

Výpis 5.4: Rozšíření metody `buildBindings`.

Třídu `LanguageServerProviderAccessor` zajišťující běh instance jazykového serveru rozšířím o abstraktní metody pro nastavení a získání hodnot inicializačních hodnot.

```
1 public abstract class LanguageServerProviderAccessor {
2     ...
3     public abstract Object getInitOptions(LanguageServerDescription desc);
4     public abstract void setInitOptions(LanguageServerDescription desc,
        Object initOptions);
5 }
```

Výpis 5.5: Rozšíření třídy `LanguageServerProviderAccessor`.

Samotnou implementaci abstraktních metod provádím v rozhraní `LanguageServerProvider` a třídě `LanguageServerDescription`. Rozhraní je třeba rozšířit hned o několik vlastností. Je nutné založit privátní proměnnou zajišťující Objekt `initOptions`. Dále je potřeba přidat nový konstruktor pro vytvoření instance jazykového klienta přijímající parametr `initOptions`. Po rozšíření o nový konstruktor je třeba ošetřit zachování zpětné kompatibility při volání původního konstruktoru.

```
1 public interface LanguageServerProvider {
2     ...
3     public static final class LanguageServerDescription {
4         public static @NonNull LanguageServerDescription create(@NonNull
            InputStream in, @NonNull OutputStream out, @Nullable Process
            process) {
5             return new LanguageServerDescription(in, out, process, null);
6         }
7         public static @NonNull LanguageServerDescription create(@NonNull
            InputStream in, @NonNull OutputStream out, @Nullable Process
            process, @Nullable Object initOptions) {
8             return new LanguageServerDescription(in, out, process, initOptions
                );
9         }
10    ...
11 }
```

Výpis 5.6: Rozšíření třídy `LanguageServerDescription` o druhý konstruktor.

Oba konstruktory při předávání své návratové hodnoty vracejí novou instanci `LanguageServerDescription`. Konstruktor s třemi parametry na místo očekávaného nového parametru dosazuje hodnotu `null`. Tím je zachována zpětná kompatibilita při volání původního konstrukturu.

```
1 public interface LanguageServerProvider {
2     ...
3     public static final class LanguageServerDescription {
4         ...
5         private Object initOptions;
6         ...
7         private LanguageServerDescription(InputStream in, OutputStream out,
8             Process process, Object initOptions) {
9             ...
10            this.initOptions = initOptions;
11        }
12    }
13 }
14 }
```

Výpis 5.7: Zpracování nových parametrů třídou `LanguageServerDescription`.

Takto upravené aplikační rozhraní je potřeba zkompletovat pomocí nástroje `ant` [15]. Nově vzniklý soubor s upraveným aplikačním rozhraním je nutné připojit jako závislost do implementace jazykového klienta. Zdrojové kódy k upravenému aplikačnímu rozhraní jsou součástí přílohy této bakalářské práce.

5.3 Publikace zásuvného modulu

Zásuvný modul vytvořený v této práci je v plánu publikovat v repozitáři zásuvných modulů *Apache NetBeans Plugin Portal*. To nyní není na základě nutné úpravy aplikačního programového rozhraní pro zajištění kompletní funkčnosti jazykového klienta možné. Po rozšíření aplikačního rozhraní o funkci inicializace parametrů bude možné jazykového klienta přizpůsobit a následně zveřejnit. Zásuvný modul bude procházet posouzením ověřených dobrovolníků z komunity *Apache NetBeans*. Po jejich schválení bude zásuvný modul publikován.

Kapitola 6

Ověření funkcionality

Nedílnou součástí procesu vývoje aplikací je testování. Automatizace testování na úrovni uživatelského rozhraní je obtížná, při absenci vhodného integračního rozhraní až nemožná. Proto bylo zvoleno manuální testování na základě testovacích scénářů. Tato kapitola popisuje testovací scénáře pro jednotlivé funkční vlastnosti jazykového klienta. Součástí jsou i jednotlivé kroky pro manuální testování.

6.1 Testování jazykového klienta

Pro ověření funkčnosti klienta slouží následující manuální testovací scénáře. Pro testování je třeba použít instanci *Apache NetBeans* obsahující upravené aplikační programové rozhraní. Spuštění je popsáno v manuálu obsaženém v příloze této bakalářské práce. Pro testování je možné použít následující kód. Po každé změně v nastavení klienta je třeba restartovat vývojové prostředí, aby se změny projevíly.

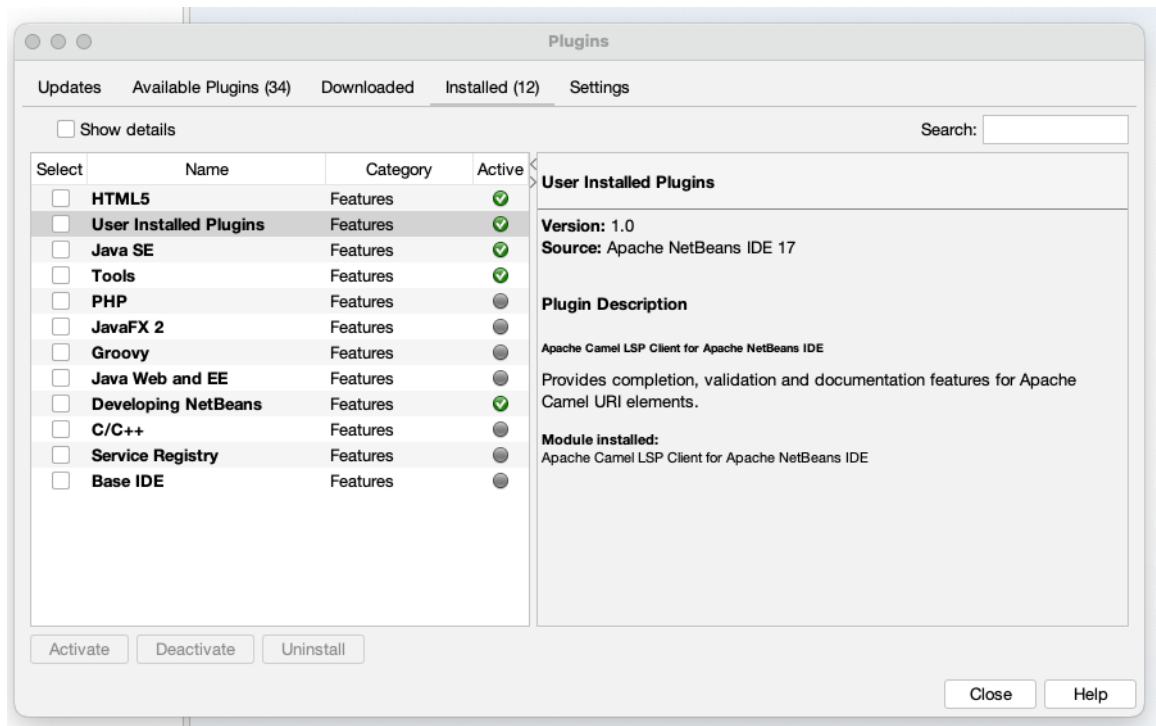
```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <camelContext id="cbr-example-context"
3   xmlns="http://camel.apache.org/schema/spring">
4   <route id="cbr-route">
5     <from id="_from1" uri=""/>
6   </route>
7 </camelContext>
```

Výpis 6.1: Kód použitý v testovacích scénářích.

6.1.1 Přítomnost klienta ve vývojovém prostředí

Cílem testovacího scénáře je ověřit, že je klient ve vývojovém prostředí *Apache NetBeans* nainstalován a je aktivní.

1. Z nabídky rozhraní prostředí *Apache NetBeans* otevřít menu **Tools**.
2. V nabídce **Tools** zvolit vnořenou položku **Plugins**.
3. Otevřít záložku **Installed** a ověřit, že zásuvný modul je nainstalován a je aktivní, viz obrázek 6.1.

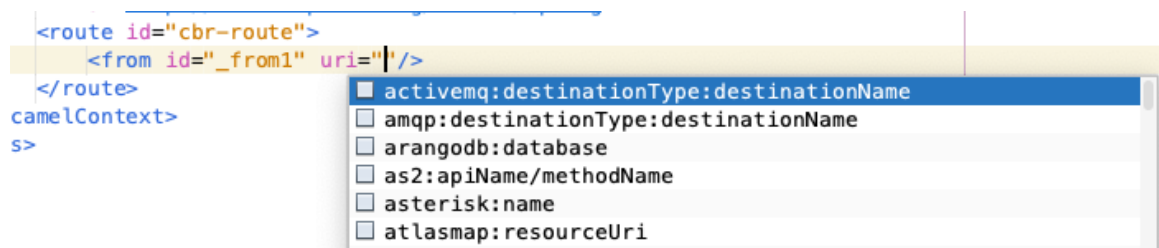


Obrázek 6.1: Nainstalovaný a aktivní zásuvný modul s klientem.

6.1.2 Dokončování kódu

Testovací scénář ověřuje základní funkční vlastnost v podobě automatického dokončování kódu včetně doplňování atributů a jejich hodnot.

1. V libovolném projektu vytvořit soubor pojmenovaný například `camel-route.xml`.
2. Jako obsah souboru vložit kód z výpisu 6.1.
3. Umístit kurzor mezi uvozovky na 5. řádku u atributu `uri` a vyvolat kontextovou nabídku použitím klávesové zkratky `Control + mezera`. Viz obrázek 6.2.



Obrázek 6.2: Kontextová nabídka zobrazující dostupné komponenty.

4. Z nabídky vybrat komponentu `timer:timerName`.
5. Doplnit za komponentu znak `?` a vyvolat kontextovou nabídku, jestliže se nevyvolala automaticky.

6. Ze seznamu atributů vybrat `delay`.
7. Ověřit, že došlo k doplnění výchozí hodnoty `'=1000'` a řádek obsahující komponentu vypadá následovně:

```
1 <from id="_from1" uri="timer:timerName?delay=1000"/>
```

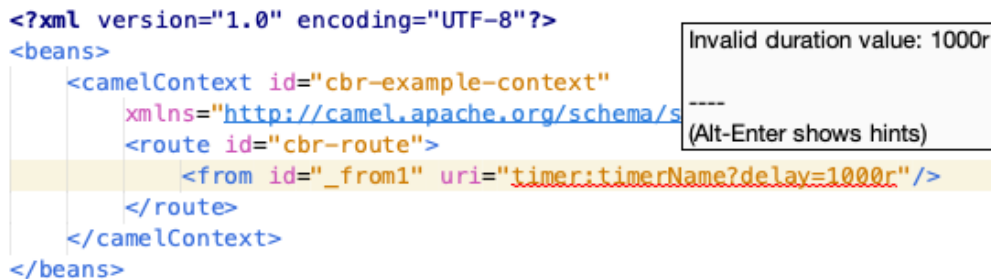
6.1.3 Diagnostika a validace

Testovací scénář ověřuje dostupnost diagnostiky a validace atributů u komponent. Vývojové prostředí automaticky zasílá požadavek na server, který kontroluje, zda jsou vložené *Camel* komponenty, atributy komponent a jejich hodnoty validní¹.

1. Opakovat postup testování dokončování kódu.
2. Za hodnotu `'delay=1000'` doplnit znak `r`, řádek tedy dostane tvar:

```
1 <from id="_from1" uri="timer:timerName?delay=1000r"/>
```

3. Zkontrolovat, že je řádek zvýrazněný a klient hlásí diagnostickou chybu. Viz obrázek 6.3.



```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
  <camelContext id="cbr-example-context"
    xmlns="http://camel.apache.org/schema/spring" >
    <route id="cbr-route">
      <from id="_from1" uri="timer:timerName?delay=1000r" />
    </route>
  </camelContext>
</beans>
```

Obrázek 6.3: Diagnostika chybných atributů komponenty.

6.1.4 Použití specifické verze katalogu

Testovací scénář ověřuje, že klient umí přepínat verze katalogů. Starší verze katalogů neobsahují stejné komponenty jako verze novější. Testovací scénář této skutečnosti využívá. Výchozí katalog ve verzi 2.15.1 neobsahuje komponentu `file-watch`. Při specifikování této konkrétní verze tedy komponenta v kontextové nabídce nemůže být dostupná.

1. Nastavit verzi katalogu na 2.15.1.
2. Mezi uvozovky na 5. řádku u atributu `uri` vložit `file` a vyvolat kontextovou nabídku.
3. Zkontrolovat, že nabídka neobsahuje komponentu `file-watch`.
4. Změnit verzi katalogu zpět na výchozí.
5. Opakovat krok 2.
6. Ověřit, že s výchozím katalogem je komponenta `file-watch` v kontextové nabídce dostupná.

¹Komponenta a její atributy existují. Hodnoty atributů mají platný datový typ a formát.

6.1.5 Použití specifického poskytovatele běhového prostředí

Testovací scénář ověřuje, že klient umí přepínat různé poskytovatele běhového prostředí. Různí poskytovatelé běhových prostředí neposkytují podporu pro všechny komponenty. Stejně tedy jako u testování specifických verzí katalogů lze této skutečnosti využít pro ověření korektnosti nastavení. Katalog komponent běhového prostředí **Quarkus** neobsahuje komponentu **JMX**, při vyvolání kontextové nabídky tedy komponenta nemůže být dostupná. Na rozdíl od starší verze katalogu je ale nutné počítat se skutečností, že se jednotlivé katalogy o komponenty stále rozšiřují, testovaná komponenta tedy může být později dostupná a testovací scénář bude vyžadovat změnu komponenty. Dostupnost komponent lze ověřit na stránkách jednotlivých poskytovatelů běhových prostředí.

1. Nastavit poskytovatele běhového prostředí na hodnotu **Quarkus**.
2. Mezi uvozovky na 5. řádku u atributu `uri` vložit `jmx` a vyvolat kontextovou nabídku.
3. Zkontrolovat, že nabídka neobsahuje komponentu `jmx:serverURL`.
4. Změnit poskytovatele běhového prostředí zpět na výchozí.
5. Opakovat krok 2.
6. Ověřit, že s výchozím běhovým prostředím je komponenta `jmx:serverURL` v kontextové nabídce dostupná.

6.1.6 Přidání vlastních komponent

Libovolný katalog komponent lze rozšířit o nabízení vlastních komponent. Testovací scénář ověřuje, zda-li jsou vlastní komponenty k dispozici v kontextové nabídce jazykového klienta.

1. V nastavení vložit vlastní komponentu.

```
1  [  
2  {  
3    "component":{  
4      "kind":"component",  
5      "scheme":"abcd",  
6      "syntax":"abcd:xyz"  
7    }  
8  }  
9  ]
```

Výpis 6.2: Kód vlastní komponenty.

2. Mezi uvozovky na 6. řádku u atributu `uri` vložit `abc` a vyvolat kontextovou nabídku.
3. Zkontrolovat, že nabídka obsahuje komponentu `abcd:xyz`.

6.1.7 Propojení pro dokončování komponent Apache Kafka

Testovací scénář ověřuje dostupnost atributu obsahujícího název topiců běžící instance *Apache Kafka* pro *Camel* komponentu `kafka`. Prerekvizity: Běžící instance *Apache Kafka* na adrese odlišné od `localhost:9092`. Tato adresa je výchozí a jazykový klient se s ní pokouší navázat spojení bez nutnosti zásahu uživatele. Vhodná je například adresa

```

<?xml version="1.0" encoding="UTF-8"?>
<beans>
  <camelContext id="cbr-example-context"
    xmlns="http://camel.apache.org/schema/spring">
    <route id="cbr-route">
      <from id="_from1" uri="abcd"/>
    </route>
  </camelContext>
</beans>

```

Obrázek 6.4: Kontextová nabídka obsahující vlastní komponentu.

localhost:9093. Instance *Apache Kafka* již musí obsahovat alespoň jeden běžící topic, například `quickstart-events`.

1. Nastavit adresu *Apache Kafka* dle běžící instance, například na `localhost:9093`.
2. Mezi uvozovky na 6. řádku u atributu `uri` vložit `kafka:` a vyvolat kontextovou nabídku.
3. Ověřit, že je k dispozici atribut obsahující název topicu, například `quickstart-events`, viz obrázek 6.5.

```

Start Page x route.xml x
<?xml version="1.0" encoding="UTF-8"?>
<camelContext id="cbr-example-context"
  xmlns="http://camel.apache.org/schema/spring">
  <route id="cbr-route">
    <from id="_from1" uri="kafka:"/>
  </route>
</camelContext>

```

Obrázek 6.5: Doplnění atributu topicu *Apache Kafka*.

6.1.8 Použití specifického jazykového serveru

Pro případ, že by uživatel nechtěl použít výchozí jazykový server, může si zvolit cestu k vlastnímu jazykovému serveru. Testovací scénář ověřuje, zda-li je použit specifikovaný jazykový server.

1. V nastavení aktivovat přepínač pokročilého nastavení (`Advanced options`).
2. Ze souborového systému vybrat cestu k jinému jazykovému serveru *Apache Camel*.
3. Otevřít soubor obsahující doménově specifický jazyk.
4. Pomocí nástroje `jps` [8] ověřit, že se spustila požadovaná verze serveru.

```
→ ~ jps
41157 Launcher
41926 Jps
41674 camel-lsp-server-1.5.0.jar
41405 Main
└─
```

Obrázek 6.6: Výpis nástroje jps.

6.2 Srovnání s vestavěným jazykovým klientem

Jak bylo uvedeno v kapitole 3.5, vestavěný jazykový klient ve vývojovém prostředí *Apache NetBeans* není možné pro jazykový server *Apache Camel* použít. Srovnání tedy může probíhat pouze v teoretické rovině a za předpokladu, že by nakonfigurovaný vestavěný jazykový klient byl s jazykovým serverem *Apache Camel* kompatibilní.

Největší výhodou vlastní implementace jazykového klienta je uživatelská přívětivost. Instalace takového zásuvného modulu z repozitáře zásuvných modulů *Apache NetBeans Plugin Portal* či přímá instalace ze souboru se zásuvným modulem nevyžaduje žádnou dodatečnou konfiguraci a pro uživatele instalace obnáší jen pár kliknutí. Vestavěný jazykový klient takovou konfiguraci vyžaduje. Ze strany vývojového prostředí jsou navíc požadavky nejasné, konfigurace tedy není příliš přívětivá.

Vestavěný jazykový klient by také uměl využít pouze určitou část funkčních vlastností jazykového klienta. Zcela jistě by bylo možné provozovat základní funkční vlastnosti jako je dokončování komponent, atributů a hodnot atributů. V případě, že klient nepředává serveru žádné specifické požadavky, server pracuje se svými výchozími hodnotami. Podpora pro typy souborů `xml`, `yaml` a `java` by byla zachována i v případě použití vestavěného klienta. Nebylo by tedy nutné vytvářet tři konfigurace. Vestavěný klient umožňuje specifikovat více než jeden atribut definující typ souboru.

Vestavěný klient by neumožňoval využít přidanou hodnotu implementovaného klienta v podobě pokročilých nastavení, která jsou dostupná z uživatelského rozhraní a popsána v kapitole 4.3. Nebylo by tedy možné použít jiné verze katalogu, jiné poskytovatele běhového prostředí či vlastní komponenty.

Kapitola 7

Závěr

Cílem bakalářské práce byl návrh zásuvného modulu poskytujícího jazykovou podporu pro rámec *Apache Camel* v integrovaném vývojovém prostředí *Apache NetBeans* a následná implementace navrženého zásuvného modulu postaveného nad aplikačním programovým rozhraním vývojového prostředí.

V rámci analýzy potřeb k řešení zadání bakalářské práce bylo nejprve nutné se blíže seznámit s jazykovým protokolem *Language Server Protocol* a s jeho architekturou. Dále bylo zapotřebí nastudovat rámec *Apache Camel* a jeho již existující dostupnou implementaci jazykového serveru. V poslední části teoretického seznámení bylo nastudováno integrované vývojové prostředí *Apache NetBeans*, možnosti jeho rozšíření a aplikační programové rozhraní užívané k vývoji zásuvných modulů.

Základním předpokladem jazykového klienta byla uživatelská přívětivost. Při návrhu jazykového klienta bylo zapotřebí nastudovat rámec *Swing*, sloužící pro tvorbu grafických rozhraní v prostředí *Java*. Při návrhu architektury bylo také nezbytné specifikovat jednotlivé funkční vlastnosti tvořeného jazykového klienta.

Výstupem bakalářské práce je funkční jazykový klient. Zásuvný modul s jazykovým klientem bylo zapotřebí implementovat na základě návrhu, který byl vytvořen. Během implementace zásuvného modulu byl objeven zásadní nedostatek v aplikačním programovém rozhraní integrovaného vývojového prostředí *Apache NetBeans*. Na základě této skutečnosti byl vypracován návrh řešení a otevřen požadavek na začlenění vlastnosti. Požadavek nyní čeká na vyřízení komunitou *Apache NetBeans*. I přes tuto skutečnost bylo možné jazykového klienta implementovat v plném rozsahu.

Po úspěšné implementaci jazykového klienta byly provedeno testování. Pro proces testování byly navrženy manuální testovací scénáře, vycházející z funkčních vlastností implementovaného klienta. Vytvořený klient byl také porovnán s alternativními dostupnými řešeními.

Všechny zadané cíle této práce byly úspěšně splněny, ověřeny a výsledný jazykový klient vyplnil požadavky firmy *Red Hat* pro použití v praxi a budoucí vývoj klienta. Ten je v současné době limitován aplikačním rozhraním vývojového prostředí *Apache NetBeans*. Po začlenění navrhovaných úprav do aplikačního programového rozhraní bude možné vzniklého klienta publikovat v repozitáři zásuvných modulů *Apache NetBeans Plugin Portal*. Dále bude možné klienta v budoucnu rozšiřovat o nové funkční vlastnosti, které jazykový server pro *Apache Camel* plánuje poskytovat.

Literatura

- [1] CAMEL TOOLING. *Camel Language Server* [online]. 2023 [cit. 2023-04-20]. Dostupné z: <https://github.com/camel-tooling/camel-language-server/>.
- [2] ECLIPSE FOUNDATION. *Eclipse LSP4J™* [online]. 2023 [cit. 2023-03-10]. Dostupné z: <https://github.com/eclipse-lsp4j/lsp4j>.
- [3] ECLIPSE FOUNDATION. *OSGi Working Group* [online]. 2023 [cit. 2023-02-02]. Dostupné z: <https://www.osgi.org/>.
- [4] HOHPE, G. a WOOLF, B. *Enterprise integration patterns: designing, building, and deploying messaging solutions*. 1. vydání. Boston: Addison-Wesley, 2004. ISBN 978-0-321-20068-6.
- [5] MICROSOFT. *Microsoft Language Server Protocol* [online]. 2022 [cit. 2023-02-03]. Dostupné z: <https://microsoft.github.io/language-server-protocol/>.
- [6] MICROSOFT. *Microsoft Language Server Protocol Overview* [online]. 2022 [cit. 2023-02-08]. Dostupné z: <https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/>.
- [7] MOZILLA CORPORATION. *Common MIME Types* [online]. 2023 [cit. 2023-03-15]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Common_types.
- [8] ORACLE CORPORATION. *JPS - Java Virtual Machine Process Status Tool* [online]. 2020 [cit. 2023-04-17]. Dostupné z: <https://docs.oracle.com/javase/7/docs/technotes/tools/share/jps.html>.
- [9] ORACLE CORPORATION. *The Java Naming and Directory Interface* [online]. 2022 [cit. 2023-03-20]. Dostupné z: <https://docs.oracle.com/javase/tutorial/jndi/overview/index.html>.
- [10] RED HAT, INC.. *Red Hat Fuse* [online]. 2023 [cit. 2023-03-16]. Dostupné z: <https://www.redhat.com/en/technologies/jboss-middleware/fuse>.
- [11] RED HAT, INC.. *Red Hat OpenShift* [online]. 2023 [cit. 2023-02-20]. Dostupné z: <https://www.redhat.com/en/technologies/cloud-computing/openshift>.
- [12] SOFTWARE FREEDOM CONSERVANCY, INC.. *Git* [online]. 2023 [cit. 2023-03-20]. Dostupné z: <https://git-scm.com/>.
- [13] THE APACHE SOFTWARE FOUNDATION. *Apache ServiceMix* [online]. 2018 [cit. 2023-02-05]. Dostupné z: <https://servicemix.apache.org/>.

- [14] THE APACHE SOFTWARE FOUNDATION. *Apache ActiveMQ®* [online]. 2023 [cit. 2023-02-05]. Dostupné z: <https://activemq.apache.org/>.
- [15] THE APACHE SOFTWARE FOUNDATION. *The Apache Ant Project* [online]. 2023 [cit. 2023-02-18]. Dostupné z: <https://ant.apache.org/>.
- [16] THE APACHE SOFTWARE FOUNDATION. *Apache Camel: Blueprint support* [online]. 2023 [cit. 2023-02-06]. Dostupné z: <https://camel.apache.org/components/2.x/others/blueprint.html>.
- [17] THE APACHE SOFTWARE FOUNDATION. *Apache Camel: Java DSL support* [online]. 2023 [cit. 2023-02-06]. Dostupné z: <https://camel.apache.org/manual/java-dsl.html>.
- [18] THE APACHE SOFTWARE FOUNDATION. *Apache Camel: Scala DSL support* [online]. 2023 [cit. 2023-02-05]. Dostupné z: <https://camel.apache.org/components/2.x/others/scala.html>.
- [19] THE APACHE SOFTWARE FOUNDATION. *Apache Camel: Spring framework support* [online]. 2023 [cit. 2023-02-04]. Dostupné z: <https://camel.apache.org/manual/spring.html>.
- [20] THE APACHE SOFTWARE FOUNDATION. *Apache CXF™: An Open-Source Services Framework* [online]. 2023 [cit. 2023-02-07]. Dostupné z: <https://cxf.apache.org/>.
- [21] THE APACHE SOFTWARE FOUNDATION. *Apache Karaf* [online]. 2023 [cit. 2023-02-12]. Dostupné z: <https://karaf.apache.org/>.
- [22] THE APACHE SOFTWARE FOUNDATION. *Apache Maven Project* [online]. 2023 [cit. 2023-01-28]. Dostupné z: <https://maven.apache.org/>.
- [23] THE LINUX FOUNDATION. *Kubernetes* [online]. 2023 [cit. 2023-02-22]. Dostupné z: <https://kubernetes.io/>.