

DOKUMENTACE K SEMESTRÁLNÍ PRÁCI  
Z PŘEDMĚTU KIV/SAR

# Analýza a Proof of Concept dekompozice systému CRCE

*Tomáš Ballák*

A19N0023P – ballakt@students.zcu.cz

*Martin Šebela*

A19N0046P – msebela@civ.zcu.cz

7. ledna 2021

# Obsah

<b>1</b>	<b>Zadání</b>	<b>2</b>
<b>2</b>	<b>Analýza architektury</b>	<b>3</b>
<b>3</b>	<b>Návrh nové architektury</b>	<b>4</b>
3.1	Sdílené moduly . . . . .	4
3.2	Microservices . . . . .	5
3.3	Bezpečnost . . . . .	6
3.4	Komunikace . . . . .	6
<b>4</b>	<b>Proof of concept</b>	<b>7</b>
4.1	Adresářová struktura . . . . .	7
4.2	<code>docker-compose</code> . . . . .	7
4.3	Dockerfile . . . . .	8
4.4	Maven Wrapper . . . . .	8
4.5	Flow sestavení . . . . .	9
4.5.1	Možnosti sestavení . . . . .	9
<b>5</b>	<b>Závěr</b>	<b>11</b>

# 1 Zadání

Tématem semestrální práce je analýza systému *CRCE* (*Component Repository supporting Compatibility Evaluation*) z pohledu změny softwarové architektury a *proof of concept* navržených změn.

Pro výzkumné účely bylo ve skupině *ReliSA* na *KIV FAV ZČU* vyvíjeno experimentální úložiště *CRCE* – <https://github.com/ReliSA/CRCE>, které je třeba dále rozvíjet. Architektura úložiště – komponentová a plug-in based, ale v zásadě monolitická co do výsledného nasazení – již nevyhovuje potřebám a komplikuje dlouhodobou údržbu.

Cílem práce proto je analyzovat možnosti změny architektury do podoby několika samostatných aplikací, volně provázaných datovými a/nebo servisními rozhraními, a realizovat alespoň dílčí funkční refactoring tímto směrem. Analýza může volně navázat na výsledky předchozích *KIV/SAR* a diplomových prací, které budou k dispozici.

Aplikace využívá technologie *Java*, *OSGi*, *REST API* a databázi *MongoDB*. Úvodní stránku aplikace ukazuje obr. 1.1.



Obrázek 1.1: Úvodní stránka systému *CRCE* po přihlášení uživatele.

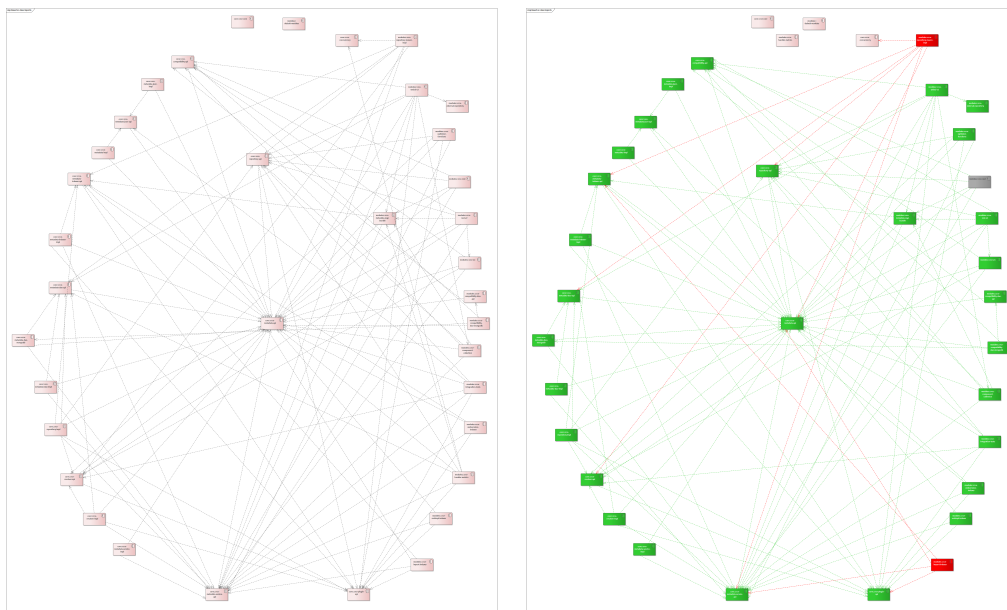
## 2 Analýza architektury

Nejprve došlo k analýze aktuální softwarové architektury systému *CRCE*, při které byly využity výstupy z předchozích *KIV/SAR* prací.

V současné architektuře je projekt rozdělen na tzv. **core** moduly a **no-core** moduly a knihovny třetích stran.

Jeden z diagramů předchozích *KIV/SAR* prací (z ak. roku 2018/19) ukazuje závislosti jednotlivých modulů (tj. **core** i **no-core** mezi sebou, a to na základě **importů** v jednotlivých *Java* souborech – viz levý diagram na obr. 2.1.

Protože mohlo dojít během doby od vytvoření původního diagramu k několika změnám v implementaci, celou analýzu závislostí modulů jsme provedli znovu (tzn. procházení každého *Java* souboru na **dev** větvi), přičemž jsme použili původně vytvořený diagram a ten následně obarvovali – viz pravý diagram na obr. 2.1. Červeně označené komponenty ukazují již smazané moduly (**crce-repository-maven-impl**, **crce-import-indexer**), šedě označený modul **crce-rest** jsme navrhli ke smazání z důvodu jeho nepoužití napříč aplikací.



Obrázek 2.1: Diagramy závislostí jednotlivých modulů na základě **importů** v jednotlivých souborech (vlevo diagram převzatý z předchozí *KIV/SAR* práce z ak. roku 2018/19, vpravo znovu ověření diagramu provedeného v rámci této práce)

## 3 Návrh nové architektury

Z pohledu zadání je cílem navrhnout architekturu tak, aby nevykazovala vlastnosti monolitické architektury. V následujících podkapitolách jsou popsány provedené změny, přičemž se vychází z analýzy provedené v kapitole 2.

### 3.1 Sdílené moduly

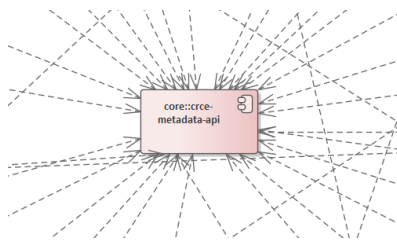
Jak ukazuje obrázek 2.1 se závislostmi jednotlivých modulů, je zde vidět hromadná závislost (viz obr. 3.1) na následujících modulech:

- `crce-metadata-api` (uprostřed diagramu na obr. 2.1),
- `crce-plugin-api` (vpravo v dolní části diagramu na obr. 2.1).

K těmto modulům lze přidat ještě následující tři moduly, které mají přímou závislost na modulu `crce-metadata-api` a nemají vazbu s žádným dalším modulem:

- `crce-metadata-impl`,
- `crce-metadata-json-api`,
- `crce-metadata-json-impl`.

Všech těchto pět modulů tak představuje tzv. *sdílené moduly* a nově se v ostatních modulech, které je vyžadují, používají formou *JAR* knihovny. Důvodem oddělení do samostatné knihovny bylo, abychom se zbavili zmíněných vazeb a mohli provádět další granularitu celého systému.



Obrázek 3.1: Ukázka závislosti ostatních modulů na modulu `crce-metadata-api`.

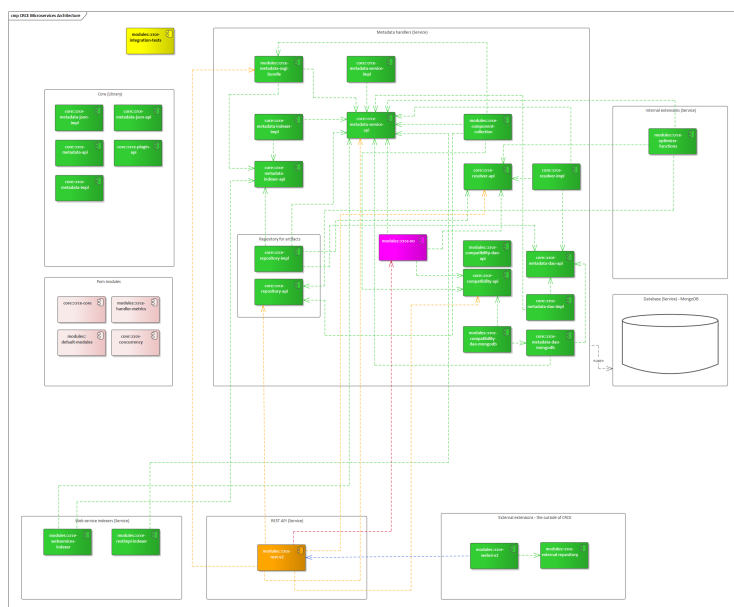
## 3.2 Microservices

Zbylé moduly byly rozděleny dle jejich určení a významu do jednotlivých *microservices* tak, jak ukazuje obr. 3.2. Došlo tak k vytvoření následujících pěti *microservices*:

1. *metadata handlers* – jádro systému
2. *web service indexers* – moduly určené pro indexaci
3. *REST API* – modul řešící komunikaci s vnějším světem
4. *internal extensions* – další interní rozšíření systému (např. výpočty *cost* funkce)
5. *external extensions* – webové rozhraní přímo komunikující pouze s *REST API*

Databáze *MongoDB* běží ve vlastním, dalším kontejneru (tedy také jako *microservice*, toho již bylo nicméně dosaženo díky předchozím *KIV/SAR* pracím).

Vazby s integračními testy (žlutý modul *crce-integration-tests*), které byly původně součástí aplikace, nebyly v důsledku změny architektury zachovány.



Obrázek 3.2: Nově navržená architektura systému *CRCE* rozdělená na jednotlivé *microservices* a sdílené knihovny (blok v levém horním rohu).

### 3.3 Bezpečnost

V původní architektuře existují přímé vazby mezi jádrem aplikace a mezi webovým rozhraním aplikace – tj. modulem `crce-webui-v2`. S jádrem aplikace ovšem komunikuje i modul `crce-rest-v2`. Vzniká tak možnost nebezpečí dvojí logiky aplikace (v modulu `crce-webui-v2` může existovat metoda pro kontrolu vstupu, zatímco v modulu `crce-rest-v2` to bude opomenuto apod.).

Z pohledu nově navržené architektury se přikláníme k řešení, že komunikace s jádrem aplikace – resp. mezi *metadata handlers microservice* a mezi *external extensions microservice* (tj. web) – bude probíhat pouze přes *REST API microservice*, abychom tuto dvojí logiku vyloučili. Zároveň tím dáváme možnost potenciálním zákazníkům aplikace, aby si vytvořili vlastní webové rozhraní pro obsluhu aplikace a nemuseli nutně používat webové rozhraní dodávané v systému *CRCE* (např. z důvodu, kdy by měla být funkčnost aplikace implementována do jiného, již existujícího systému).

### 3.4 Komunikace

Komunikace mezi jednotlivými *microservices* nebyla v rámci práce řešena, nicméně předpokládáme nasazení frameworku *Springu* v kombinaci s již nasazeným *OSGi* a následnou komunikaci pomocí *REST API* mezi *microservices*.

Alespoň tedy dodáváme zdroje, ze kterých by bylo možné vycházet:

- <https://github.com/dimmik/osgi-spring-boot-demo>
- <https://medium.com/@amitbhoraniya/spring-boot-with-osgi-25d2387a459e>
- <https://github.com/vmudigal/microservices-sample>

## 4 Proof of concept

Součástí práce je *proof of concept*, který ukazuje implementaci/refactoring alespoň dílčí části nově navržené architektury.

### 4.1 Adresářová struktura

Změnou architektury došlo ke změně adresářové struktury projektu tak, aby odpovídala jednotlivým *microservices* a *sdíleným modulům*. Adresářová struktura je tedy nyní (resp. ve větvi `sar-2020` v repozitáři projektu) následující:

- `build` – soubory zajišťující sestavení modulů třetích stran pro použití v *OSGi*
- `deploy` – další soubory zajišťující sestavení projektu včetně konfiguračních souborů
- `doc` – soubory s dokumentacemi prací z předmětu *KIV/SAR*
- `services` – jednotlivé *microservices*, každá z nich pak obsahuje podadresář:
  - `modules` – moduly, které spadají do dané *microservice*
  - `parent-aggregation` – rodič pro `pom.xml` soubory (nese informace o názvu a dalších informacích o *microservice*)
  - `third-party` – *OSGi bundles* třetích stran, které daná *microservice* vyžaduje
  - `metadata-modules.txt` – soubor obsahující seznam modulů, které daná *microservice* vyžaduje ke svému běhu a jsou umístěny v jádrové *metadata handlers microservice*
- `shared-modules` – obsahuje sdílené moduly, které vyžadují ostatní moduly napříč *microservices*, viz podkapitola 3.1

### 4.2 docker-compose

Součástí práce bylo vytvoření souboru `docker-compose.yml`, v němž jsou specifikovány jednotlivé *microservices*, které se mají sestavit a spustit.



Součástí tohoto nastavení je, na jakém portu bude daná *microservice* naslouchat (např. webové rozhraní *external extensions microservices* na portu 8080, *REST API microservice* na portu 8082 apod.), `connection string` k databázi, *docker volume*, na němž jsou uloženy sdílené moduly apod.

Všechny *microservices* navíc používají námi nadefinovanou vnitřní síť *crce*. Jejich běh tak neovlivňují požadavky z hostitelského operačního systému.

Součástí `docker-compose` je ještě další soubor `.env`, který obsahuje používané proměnné (názvy *docker volume*, cesty v souborovém systému apod.).

Využití `docker-compose` navíc umožňuje použít škálování, kdy lze pomocí argumentu `-scale SERVICE=NUM` nastavit počet spuštěných instancí konkrétní *microservice*.

Pro sestavení projektu je třeba mít nainstalován `docker-compose` minimálně ve verzi *v1.27.4*.

## 4.3 Dockerfile

Projekt nově obsahuje dva obsáhlé `Dockerfile` soubory, které společně se skriptem `build_shared_modules.sh` připraví *docker volume* se sdílenými moduly a v rámci `Dockerfile` souboru pak dojde k nakopírování potřebných souborů do každé překládané *microservice*.

Rozdělení na dva `Dockerfile` soubory je z toho důvodu, že jeden je použit pro jádrovou *metadata handlers microservice*, druhý pak pro zbylé *microservices*.

`Dockerfile` obsahuje navíc i konfiguraci *Apache Felix* z předchozí *KIV/SAR* práce.

## 4.4 Maven Wrapper

Pro sestavení se nově používá *Maven Wrapper*, který funguje ve formě samostatného skriptu, čímž odpadá nutnost instalace *Maven* do *docker container*. Navíc lze specifikovat, jaká konkrétní verze *Maven* má být použita.

V rámci `Dockerfile` byl pro *Maven Wrapper* vytvořen alias `mvn` namísto `mvnw`. Voláním `mvn` se tedy volá *Maven Wrapper*.

Každé volání příkazu `mvn` se provádí s parametrem `--no-transfer-progress`, který uživateli na výstup nevypisuje informace o stahování (zprávy typu *downloading...*). V současném stavu jsou během sestavování projektu přeskočeny testy (z důvodu urychlení celého

sestavení). V budoucnu by nicméně bylo vhodnější dát možnost provést sestavení opět včetně funkčních testů.

## 4.5 Flow sestavení

Pro sestavení projektu jsou nejprve vytvořeny všechny potřebné, sdílené *docker volumes*.

Například *docker volume shared-modules*, který obsahuje přeložené *sdílené moduly* ve formě *JAR* knihoven (viz podkapitola 3.1). K jeho vytvoření dojde díky skriptu `build.sh`. Dále pak skript `build_shared_modules.sh` stáhne a připraví potřebné závislosti, vytvoří požadovanou adresářovou strukturu na základě `package` a verze modulu a vytvoří výsledný *JAR* soubor v tomto adresáři. Nakonec dojde ještě k vytvoření `.pom` souboru, který obsahuje informace o daném modulu.

Další na řadu přichází sestavení `metadata-modules`. Ty jsou sestaveny jako klasická *microservice* narozdíl od `shared-modules`. Do vytvořeného *docker volume* se zkopíruje aktuální zdrojový kód a ten se následně sestaví pomocí *Maven*. Následuje spuštění příkazu `mvn clean pax:directory`. Ten shromáždí *JAR* soubory jednotlivých modulů do jednoho adresáře. Potom přijde na řadu z dřívějšíka existující soubor `prepare-bundles.sh`, který shromáždí všechny potřebné informace do *JAR* souborů pro každý modul a zkopíruje je do *Apache Felix* adresáře (do `bundles`).

Po sestavení `metadata-modules` se spustí jednotlivé *microservices*, které pro své sestavení využívají výše zmíněná *docker volumes*.

Po sestavení každé *microservice* se spustí *Apache Felix*, který načte jednotlivé `bundles` a vznikne tak funkční celek.

Celý tento proces je detailně popsán i v repozitáři projektu a stejně tak jsou okomentovány i skripty, které zajišťují sestavení.

### 4.5.1 Možnosti sestavení

#### `build all`

Sestaví jak `shared-modules`, tak `metadata-modules` a zároveň připraví všechny *docker volumes* a spustí jednotlivé *microservices*.

#### `build`

Funguje stejně jako `build all` s tím rozdílem, že nesestaví `shared-modules`.

**start.sh**

Spustí pouze jednotlivé *microservices*.

## 5 Závěr

Výstupem semestrální práce je nově navržená softwarová architektura aplikace *CRCE*, přičemž cílem bylo rozdělit aplikaci do několika dílčích modulů, resp. jednotlivých *microservices* tak, aby nevykazovala vlastnosti monolitické architektury. Toho bylo dosaženo rozdělením na pět *microservices* (resp. šest včetně databáze) a na sdílené knihovny.

Součástí práce je i *proof of concept*, který umožňuje aplikaci sestavit a spustit tak, aby odpovídala nově navržené architektuře. K tomu bylo třeba změnit *build* proces celého projektu a upravit/doplnit `pom.xml` soubory u každého z modulů.

Zároveň došlo k vytvoření dalších souborů `Dockerfile` a stejně tak souboru `docker-compose.yml`. Sestavení a spuštění celého projektu je po provedení těchto změn navíc jednodušší než bylo v původních verzích aplikace – stačí jen spustit skript `build.sh`, který připraví sdílené *docker volumes*, postupně přeloží zdrojový kód jednotlivých *microservices*, spustí databázi *MongoDB* a celou aplikaci poté spustí na zvolených portech.

Zachována a otestována byla i funkčnost webového rozhraní na portu 8080 (nyní tedy již ve formě *microservice*). *Microservice* obsluhující *REST-API* je dostupný na portu 8082 (viz soubor `docker-compose`) a také funguje dle očekávání (tzn. vrací požadované výsledky z databáze při zobrazení konkrétního *REST API endpoint*).

Zároveň došlo k opravení některých chyb z předchozích verzí projektu (např. opraveno vytváření *ZIP* souborů ve skriptu `prepare-bundles.sh`). Spolu s tím došlo k vyčištění celého projektu od nepoužívaných souborů. Adresářová struktura projektu je tak na první pohled odlišná od jiných větví projektu (způsobeno především změnou architektury na *microservices*).

Všechny nově vytvořené skripty byly okomentovány. Výsledek semestrální práce lze vidět na větvi `sar-2020` dostupné v repozitáři *GitHub* na URL adrese <https://github.com/ReliSA/CRCE/tree/sar-2020>.

Komunikace mezi jednotlivými *microservices* nebyla implementována, ale součástí dokumentace jsou zdroje, pomocí nichž by bylo možné tuto funkcionalitu doimplementovat.