

Projekt: DIC - Dependency Injection Container

***Abstrakt***

*Vytvořit Dependency Injection framework a ten ověřit na konstrukci objektového modelu vámi zvolené aplikace*

*POZOR: Kratší na vývoj, ale těžší a pro ty, kteří se na to cítí*

***Funkční požadavky***

1. Práce DIC je ovládána **pomocí anotací** uvnitř tříd
2. Zdrojové beany jsou uvnitř třídy anotovány pomocí ***@Configuration***
3. Injection instancí tříd do instance třídy lze provádět přes ***constructor*, *setter* nebo *property****.* To, kde se injekce použije a kde ne, specifikujte např: anotací***@Autowired****.*
4. Lze definovat seznam packages, jejichž třídy jsou spravovány DIC (DIC scanuje tyto packages)
5. V případě, že ve správě DIC existuje více než jedna třída splňující kontrakt pro injection, tak implementujte mechanismus, jak blíže specifikovat konkrétní třídu, která se použije.
6. U instancí spravovaných DIC lze nastavit 3 typy *scope* a to:
	* ***Singleton*** - DIC používá stále stejnou instanci
	* ***Prototype*** - DIC při každém požadavku na instanci vrátí novou instanci třídy
	* ***Clone*** - DIC při každém požadavku na instanci vrátí kopii posledně vytvořené instance. Podle konfigurace použitého *CloneProvideru* se provádí buď shallow copy nebo deep copy
7. DIC mohu nastartovat s různými konfiguracemi - konfigurační soubor např. Ve formátu yaml. V konfiguraci mohu specifikovat potřebné parametry (jako např parametry cache nebo např. i cesty k packages, které jsou scanovány DIC)
8. Implementujte tzv. **Response Cache**. Response cache se nastavuje pomocí anotace @Cached, která se přidá před metodu. Takto anotovaná metoda funguje tak, že při jejím druhém a dalším volání se vrací hodnota z cache a tělo metody se neprovolává. V případě, že je dosaženo TTL, tak se metoda provolá znova.
9. Implementujte **CachingProvider**, tedy interface pro speciální třídu, kterou DIC používá pro implementaci cache. Tento provider si automaticky bere konfiguraci z konfiguračního souboru. Tam se nastavuje např. TTL pro cache. Pozn. tzv. provider poskytuje hotovou instanci nektere komponenty, která má standardizované rozhraní např. pro caching, java persistence, logging atd. A různí dodavatelé dodávají jeho implementaci.
10. Implementujte **CloningProvider**, tedy interface pro speciální třídu, kterou DIC používá pro implementaci klonování tříd (viz různé scopes). Tento provider si automaticky bere konfiguraci z konfiguračního souboru.

***Požadované výstupy***

* Design ve formě class diagramů
* Javadoc vygenerovaný pro funkce, kterými uživatel pracuje s vaším software
* Unit testy
* Použití pro konstrukci vámi zvolené komplexní objektové struktury (alespoň 10 tříd, které mezi sebou mají kardinality 1:1, 1:N nebo M:N
* Realizujte pro dvě konfigurace: (1) pro reálné použití, (2) pro test - některé služby se neprovolávají a místo nich se používá mock

***Vhodné knihovny***

java.lang.annotation.\*

[java.lang.reflect](https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html).\*

org.burningwave.core.\*

***Doporučená literatur***a

<https://martinfowler.com/articles/injection.html>

<https://docs.spring.io/spring-framework/reference/core/beans.html>

<https://reflectoring.io/java-annotation-processing/>

<https://codegym.cc/groups/posts/208-dynamic-proxies>

<https://python-dependency-injector.ets-labs.org/introduction/index.html>