Contexts and Dependency Injection (CDI) & Enterprise Java Beans (EJB)

16. prosince 2020

V úvodních seminářích jsme si představili mechanismus Contexts and Dependency Injection (CDI), který umožňuje jednoduchým a typově bezpečným způsobem provázat komponenty, které jsou postaveny na principu JavaBeans. Vedle toho platforma Java EE nabízí ještě Enterprise JavaBeans (EJB), což je mnohem pokročilejší přístup k tvorbě aplikacích postavených na komponentách.

1 Contexts and Dependency Injection

Mechanismus Contexts and Dependency Injection (CDI) umožňuje pomocí anotací `@RequestScoped`, `@SessionScoped` apod. definovat jednotlivé komponenty a jejich rozsah platnosti, a následně pomocí anotace `@Inject` umožňuje definovat propojení jednotlivých komponent. Primárně jsme CDI využívali k provázání JSF stránek s logikou aplikace a k vzájemnému provázání jednotlivých komponent obstárávajících logiku aplikace. Avšak CDI lze použít i společně s webovými službami (ať už s RESTful API nebo Web Services) nebo dokonce přímo se servlety.

Na mechanismu CDI je zajímavé to, že přináší do programovaní deklarativní přístup. Vskutku, máme-li následovně deklarovaný atribut

```
@Inject private Foo foo;
```

říkáme, že v atributu `foo` očekáváme komponentu typu `Foo`. Jakým způsobem tato komponenta bude získána, jaké bude mít vlasnosti (např. rozsah platnosti), jak bude deklarována nebo inicializována, neřešíme, a přenecháváme to aplikačnímu serveru.

Ve všech scenářích, kdy jsme CDI použili, byl k identifikaci jednotlivých komponent použit typ[1] a jednotlivé komponenty byly vytvořeny s pomocí neparametrického konstruktoru. S ohledem na to, že typový systém Javy je relativně omezený a někdy může být potřeba komponentu adekvátním způsobem inicializovat, ukážeme si nyní některé rozšiřující možnosti práce s komponentami.

1.1 Rozlišení komneponet

Je běžné, že máme několik komponent, které implementují stejné rozhraní, např. pro potřeby produkčního nasazení a testování, nebo podle jednotlivých zákázníků. V takovém případě potřebujeme určit, která

[1] nebo jméno komponenty v případě JSF

implementace se má použít.

Tento problém si budeme demonstrovat na ukázkových komponentách, které obstarávají „zkrášlení textu".

Předpokládejme, že máme obecné rozhraní pro „zkrášlování textu"

```
public interface Prettifier {
    public String prettify(String str);
}
```

A chce mít dvě komponenty implementující toto rozhraní

- `BasicPrettifierBean`, která zvýrazní text vložením mezer, tj. `"Hello"` → `"H e l l o"`,
- `FancyPrettifierBean`, která zvýrazní text vložením hvězdiček, tj. `"Hello"` → `"H*e*l*l*o"`.

Pokud bychom chtěli použít komponentu na základě obecného rozhraní, např.

```
@Inject private Prettifier prettifier;
```

dostane se nám varování, resp. chyby, že není jednoznačně určené, která komponenta se má použít.

Abychom rozlišili jednotlivé komponenty, musíme použí tzv. *kvalifikátory*, což jsou anotace upřesňující jednotlivé komponenty, resp. jejich vlastnosti.

Kvalifikátor je anotace označená anotací `@Qualifier` s tím, že je nutné, aby tato vlastnost byla k dispozici za běhu programu, tj. `@Retention(RUNTIME)`.

Pro potřeby našeho příkladu anotaci ještě vybavíme atributem `desc` typu `String`, který bude indikovat způsob „zkrášlení textu".

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Style {
    String desc();
}
```

Tento kvalifikátor použijeme jednak na straně samotné komponenty, abychom specifikovali typ komponenty, a taktéž na straně s anotací `@Inject`, abychom určili, jakého typu má komponenta být. Toto použití nám ukazuje následující kód.

```
// samotna deklarace
@RequestScoped
@Style(desc = "basic")
public class BasicPrettifierBean implements Prettifier { }

@RequestScoped
@Style(desc = "fancy")
```

```
public class FancyPrettifierBean implements Prettifier { }

// pouziti
@Inject @Style(desc = "basic")
private Prettifier prettifier;
```

Ve skutečnosti, pokud nepoužijeme žádný kvalifikátor (ať už na straně komponenty, či straně s anotací `@Inject`), tak se implicitně doplní kvalifikátor `@Default`. Chtěli-li bychom udělat některou z komponent jako implicitní, můžeme ji vybavit touto anotací, tj. následovně.

```
@RequestScoped
@Style(desc = "basic")
@Default
public class BasicPrettifierBean implements Prettifier { }
```

V takovém případě bude fungovat i použití

```
@Inject private Prettifier prettifier;
```

protože v tomto případě se implicitně doplní kvalifikátor `@Default`.

1.2 Vytvoření a inicializace komponent

Dosud jsme využívali toho, že jednotlivé komponenty potřebovali inicializovat pouze v rámci svého konstrukturu, který navíc byl neparametrický a často implicitní. Jsou situace, kdy toto řešení není dostatečné, protože v rámci konstruktoru dochází pouze k inicializaci objektu, nikoliv však komponenty. Nemáme tedy k dispozici přístup k hodnotám atributů, které jsou označeny anotací `@Inject`, protože to provede aplikační server až po doběhnutí konstruktoru. Pokud potřebujeme inicializovat komponentu, slouží k tomu anotace `@PostConstruct`, kdy takto označená metoda slouží k řádné inicializaci komponenty.

Dalším způsobem, jak se vypořádat s inicializací, je přenechat to na specializovaných metodách, které se postarají o vytvoření a inicializaci celé komponenty. Tyto metody jsou označeny anotací `@Produces` a jejich návratový typ společně s kvalifikátory určuje, jaké komponenty vytváří.

Následující příklad nám ukazuje třídu, která generuje různé implementace rozhraní `Prettifier`, přičemž sdílí kód mezi jednotlivými komponentami.

```
@RequestScoped
public class PrettifierSource {

    @Produces @Style(desc = "basic")
    public Prettifier getBasicPrettifier() {
        return create(" ");
    }
```

```
    @Produces @Style(desc = "decent")
    public Prettifier getDecentPrettifier() {
        return create("-");
    }

    private Prettifier create(String decoration) {
        return (String str) -> {
          // ...
        };
    }
}
```

2 Enterprise JavaBeans

CDI představuje jednoduchý a pohodlný nástroj pro provázání komponent, avšak řadu dalších důležitých činností neřeší. Zejména se to týká synchronizace, kdy hlavní břímě spojené se synchronizací nese programátor, který musí u jednotlivých tříd identifikovat metody, jež mohou být volány z více vláken a které pracují se sdílenými prostředky, a označit je jako `synchronized`. To může být, a často je, zdrojem špatně odhalitelných chyb. Další důležitou oblastí, kterou CDI neřeší, je škálování, tj. distribuce výpočtu přes více virtuálních[2] potažmo fyzických strojů.

Tyto a celou řadu dalších problémů řeší Enterprise JavaBeans (EJB). EJB umožňují škálovat napříč více stroji, AS se postará o korektní vytvoření instancí a distribuci napříč více virtuálními stroji. EJB podporují transakční zpracování a soubežný přístup ke sdíleným objektům. A v neposlední řadě EJB podporují striktní oddělení klientské části kódu a business logiky, takže jsou vhodné v situacích, kdy máme více různých klientů pracujících se sdílenou logikou.

Zakomponovaní EJB do platformy Jave EE ilustruje Obrázek 1. Všimněte si, že EJB mohou být spravovány v rámci samostatného Enterprise Bean Containeru.

2.1 Typy EJB

EJB se dělí na dva základní typy

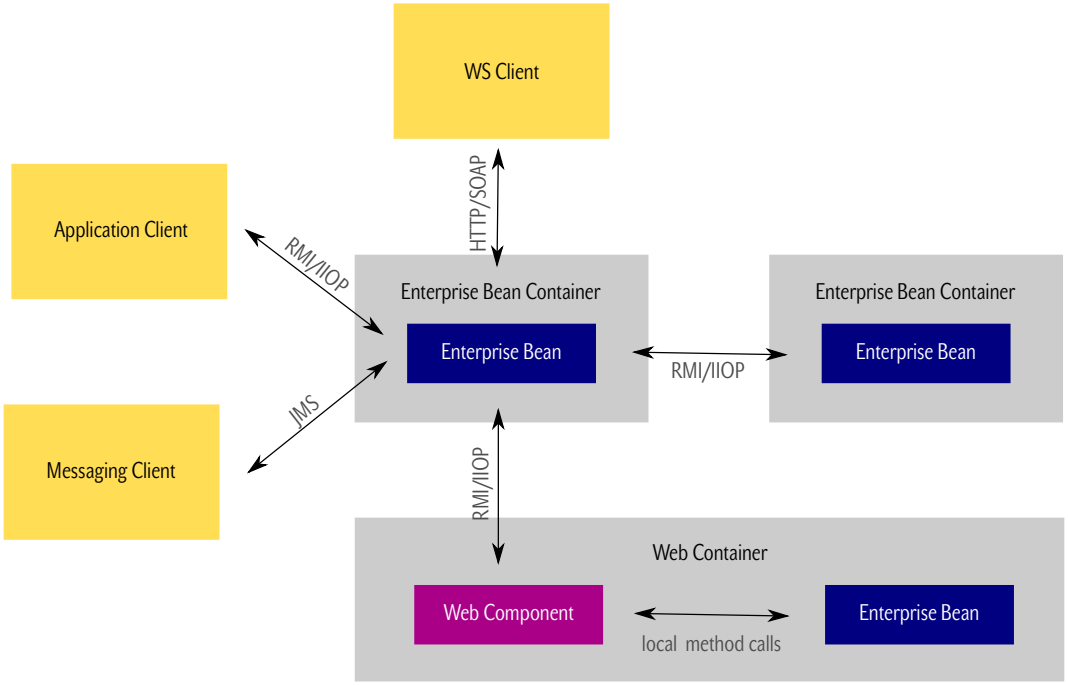
1. Session beans – komponenty vykonávající úlohy na základě požadavků klienta,
2. Message-driven beans – komponenty zpracovávající asynchronně přicházející zprávy, typicky přes Java Message Service API (JMS).

V tomto semináři se budeme věnovat hlavně session beans, message-driven beans budou podrobněji probrány v semináři následujícím.

Session beans se dále dělí na tři typy:

1. bezestavové (stateless) – komponenta nemá svůj vlastní stav a implementuje jednotlivé operace s daty,

[2] ve smyslu Java Virtual Machine



Obrázek 1: Architektura a použití EJB

2. stavové (stateful) – komponenta si uchovává stav po dobu aktivního sezení, které probíhá s klientem, musí implementovat rozhraní `Serializable`,

3. singleton – komponenta jež má nanejvýš jednu instanci v rámci aplikace a umožňuje souběžný přístup od různých klientů.

K jednotlivým Session beans se dá přistupovat (i) lokálně (v rámci jednoho virtuálního stroje) nebo (ii) vzdáleně (v rámci jednoho nebo více virtuálních strojů). Druhá varianta sice umožňuje aplikace lépe škálovat, ale je pomalejší vzhledem k použití vzdáleného volání metod a k nutnosti serializovat předáváné argumenty do binárního formátu. Různé metody přístupu ilustruje Obrázek 2.

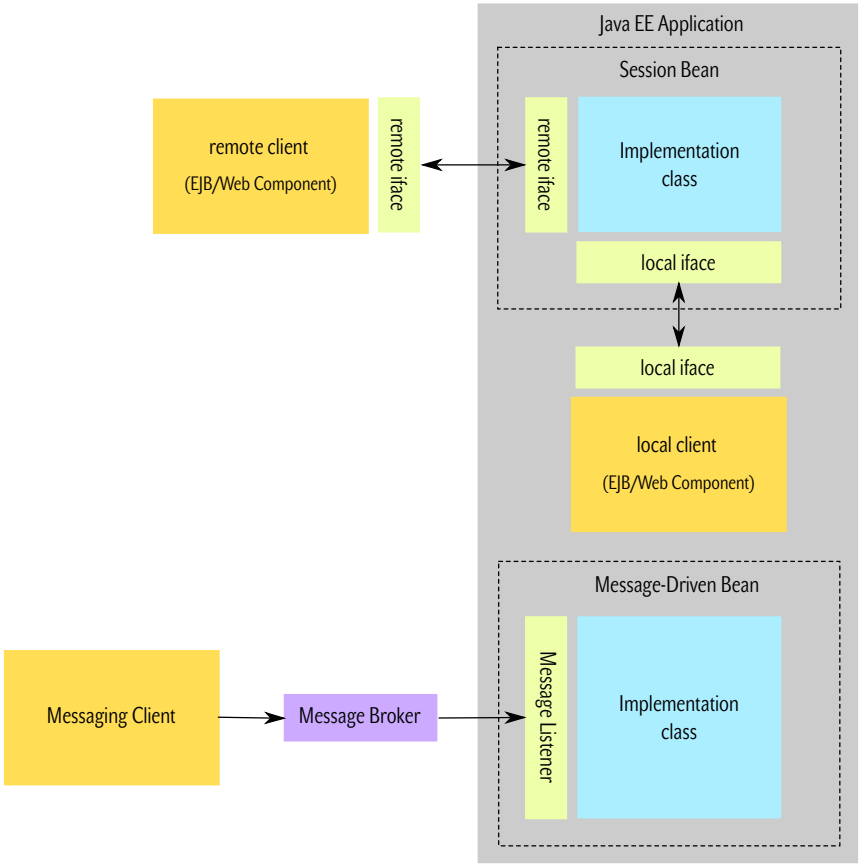
2.2 Použití EJB

EJB staví na osvědčené praxi programování proti rozhraní, tj. nejdříve je nadefinováno rozhraní, které používají klienti pro přístup k EJB a které, pochopitelně, EJB implementují.

V našem případě by použití EJB pro zkrášlování textu mohlo vypadat následovně. Nejdříve nadefinujeme rozhraní pro přístup k EJB.

```
@Local
public interface Prettifier {
    public String prettify(String str);
}
```

V tomto případě se jedná o lokální rozhraní, pokud bychom chtěli použít vzdálené rozhraní, použili bychom anotaci `@Remote`.



Obrázek 2: Různé pohledy (metody přístupu) na EJB

Implementace EJB je podobná tomu, co jsme tu již viděli, jen s tím rozdílem, že použijeme anotaci `@Stateless`, protože se jedná o bezstavovou komponentu.

```
@Stateless
public class BasicPrettifierBean implements Prettifier { }
```

A k získání reference použijeme anotaci `@EJB`.

```
@EJB private Prettifier prettifier;
```

2.3 Ukázková aplikace

Pro praktičtější pochopení, je k semináři přiložena ukázková aplikace, která umožňuje shromažďovat citáty různých osobností a pak si udělat test z jejich znalostí.

Business logika aplikace je popsána dvěma rozhraními (i) `QuoteManagement` pro správu citátů a (ii) `QuoteTest` představující rozhraní testu. Ke každému rozhraní je implementována právě jedna EJB (i) bezstavovová `QuoteManagementBean` a (ii) stavová `QuoteTestBean`.

Na implementaci těchto tříd je zajímavé to, že transakční zpracování dat přebírá AS, jelikož v nastavení `persistence.xml` je použit `transaction-type="JTA"` a k získání `EntityManageru` použita anotace `@PersistenceContext`.

```
@PersistenceContext(unitName="jpa-quote-db")
private EntityManager entityManager;
```

Toto je v zásadě standardní způsob, jak přistupovat k datům na platformě Java EE.

Další důležitou věcí hodnou pozoru v ukázkové aplikaci je její rozdělení na dva samostatné moduly[3], které jsou spojeny v jedné Java EE aplikaci. Jeden modul, označovaný jako *EJB modul* (`lecture07-ejb`), implementuje business logiku, druhý modul, označovaný jako *Web Application Module* (`lecture07-war`), implementuje uživatelské rozhraní. Všimněte si, že webová část opravdu obsahuje minimum kódu a všechen kód se vztahuje k prezentační logice a s business logikou pracuje přes definované rozhraní. Díky tomuto přístupu je pak velice snadné implementovat různé klienty pro jednu business logiku.

[3] V rámci platformy Java EE, nejedná se o rozdělení na moduly, jak je zná Java 11+.