EBook Gratuito

APPRENDIMENTO

cdi

Free unaffiliated eBook created from
Stack Overflow contributors.

#cdi

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: cdi

It is an unofficial and free cdi ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official cdi.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con cdi

Osservazioni

Questa sezione fornisce una panoramica di ciò che cdi è, e perché uno sviluppatore potrebbe voler usarlo.

Dovrebbe anche menzionare tutti i soggetti di grandi dimensioni all'interno di cdi e collegarsi agli argomenti correlati. Poiché la documentazione di cdi è nuova, potrebbe essere necessario creare versioni iniziali di tali argomenti correlati.

Examples

Installazione o configurazione

Istruzioni dettagliate su come installare o installare cdi.

Configurazione rapida in un ambiente Java SE

Se si sta lavorando con un server di applicazioni Java EE 6+, CDI fa parte del contenitore e non è necessario fare nulla per iniziare a usarlo. Ma CDI non è limitato ai server di applicazioni Java EE. Può essere utilizzato con facilità in applicazioni Java SE o semplici contenitori servlet. Diamo un'occhiata all'utilizzo di CDI in una semplice applicazione da riga di comando.

Passaggio 1. Aggiungi dipendenze al tuo POM.

```
<dependency>
    <groupId>org.jboss.weld.se</groupId>
    <artifactId>weld-se-core</artifactId>
    <version>3.0.0.Alpha15</version>
</dependency>
```

Passaggio 2. Aggiungi bean.xml

CDI richiede un file beans.xml vuoto in modo che possa eseguire la scansione del JAR per le classi. Quindi crea

```
src/main/resources/META-INF/beans.xml
```

con il seguente

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
```

```
       bean-discovery-mode="all">
    <scan>
        <exclude name="org.jboss.weld.**" />
    </scan>
</beans>
```

Passaggio 3. Inizializza CDI

In questo esempio, il metodo principale (String []) inizializza CDI e quindi CDI viene utilizzato per ottenere un'istanza della classe stessa per avviare l'esecuzione dell'applicazione SE.

```
import java.util.Arrays;
import java.util.List;
import javax.enterprise.inject.spi.CDI;
import javax.inject.Inject;

public class Main {
    public static void main(String[] args) {
        CDI<Object> cdi = CDI.getCDIProvider().initialize();
        Main main = cdi.select(Main.class).get();
        main.main(Arrays.asList(args));
    }

    @Inject
    protected MyService myService;

    protected void main(List<String> args) {
        System.out.println("Application starting");

        // MyService object injected by CDI
        myService.go();
    }
}
```

È tutto, davvero semplice.

implementazioni

CDI è una specifica Java EE . Specifica come devono essere fatte le cose e quali caratteristiche devono essere fornite, ma in realtà non è una libreria o un set di codice specifico. Per utilizzare CDI, è necessario utilizzare *un'implementazione* CDI.

L'implementazione di riferimento delle specifiche CDI è un set di librerie noto come Weld . Un'implementazione alternativa della specifica CDI esiste come Apache OpenWebBeans . Entrambe queste implementazioni saranno in grado di darti le caratteristiche del CDI. Se non si sta utilizzando un application server Java EE fornito con una di queste implementazioni, spetterà a voi selezionare e installare una di queste implementazioni nell'applicazione o nel runtime.

Leggi Iniziare con cdi online: https://riptutorial.com/it/cdi/topic/5624/iniziare-con-cdi

Capitolo 2: Iniezione di dipendenza

introduzione

La caratteristica principale di CDI è un'API dichiarativa per l'iniezione delle dipendenze. Le classi possono avere dipendenze contrassegnate con l'annotazione `@Inject` , che indica al gestore CDI che è necessario fornire tali dipendenze durante la costruzione di un'istanza della classe.

Examples

Costruttore di iniezione

Il caso comune per l'iniezione di dipendenze in una classe è con l'iniezione del costruttore. Ciò comporta l'annotazione di un costruttore sulla classe con @Inject. Il gestore CDI cercherà un costruttore con l'annotazione @Inject durante la creazione di un'istanza della classe. Quando trova un costruttore @ annotato con Inject, utilizzerà reflection per trovare quali parametri sono richiesti dal costruttore, costruire o ottenere istanze di tali dipendenze, quindi chiamare il costruttore con tali dipendenze.

```
public class Spaceship {

    private final PropulsionSystem propulsionSystem;
    private final NavigationSystem navigationSystem;

    @Inject
    public Spaceship(PropulsionSystem propulsionSystem, NavigationSystem navigationSystem) {
        this.propulsionSystem = propulsionSystem;
        this.navigationSystem = navigationSystem;
    }

    public void launch() throws FlightUnavailableException {
        if (propulsionSystem.hasFuel()) {
            propulsionSystem.engageThrust();
        } else {
            throw new FlightUnavailableException("Launch requirements not met. Ship needs
fuel.");
        }
    }

}
```

Ogni volta che un'istanza Spaceship definita in questo modo viene creata da CDI, riceverà un PropulsionSystem e un NavigationSystem come argomenti per il suo costruttore. Poiché queste dipendenze vengono aggiunte tramite il costruttore, abbiamo un modo semplice per fornire dipendenze alternative nel nostro cablaggio di test durante il test dell'unità:

```
public class SpaceshipTest {

    private Spaceship systemUnderTest;
    private TestPropulsionSystem testPropulsionSystem;
```

```
    private TestNavigationSystem testNavigationSystem;

    @Before
    public void setup() {
        setupCollaborators();
        systemUnderTest = new Spaceship(testPropulsionSystem, testNavigationSystem);
    }

    @Test
    public void launchSequenceEngagesThrustIfFueled() {
        //given
        testPropulsionSystem.simulateHavingFuel();

        //when
        systemUnderTest.launch();

        //then
        testPropulsionSystem.ensureThrustWasEngaged();
    }

}
```

Field Injection

Lo stesso esempio di cui sopra può anche essere fatto usando ciò che è noto come iniezione di campo. Invece di annotare il costruttore con `@Inject` , annotare i campi che desideriamo avere iniettato

```
public class Spaceship {

    @Inject
    private PropulsionSystem propulsionSystem;
    @Inject
    private NavigationSystem navigationSystem;

    public void launch() throws FlightUnavailableException {
        if (propulsionSystem.hasFuel()) {
            propulsionSystem.engageThrust();
        } else {
            throw new FlightUnavailableException("Launch requirements not met. Ship needs
fuel.");
        }
    }

}
```

Si noti che i metodi getter / setter non sono necessari per il funzionamento dell'iniezione sul campo. Si noti inoltre che il campo non ha bisogno di essere pubblico e può infatti essere privato, anche se non può essere definitivo. Rendere il campo privato, tuttavia, renderà più difficili i test di scrittura del codice, poiché i test dovranno usare la riflessione per modificare il campo se non ha un setter.

Si noti inoltre che l'iniezione del costruttore e l'iniezione sul campo possono essere usati in tandem, se lo si desidera, anche se dovrebbe essere valutato con molta attenzione se sia o meno opportuno farlo caso per caso. Forse è necessario quando si lavora con il codice legacy per

aggiungere una dipendenza alla classe senza modificare la firma del costruttore per qualche motivo particolare.

```
public class Spaceship {

    private PropulsionSystem propulsionSystem;
    @Inject
    private NavigationSystem navigationSystem;

    @Inject
    public Spaceship(PropulsionSystem propulsionSystem) {
        this.propulsionSystem = propulsionSystem;
    }

    public void launch() throws FlightUnavailableException {
        if (propulsionSystem.hasFuel()) {
            propulsionSystem.engageThrust();
        } else {
            throw new FlightUnavailableException("Launch requirements not met. Ship needs
fuel.");
        }
    }

}
```

Leggi Iniezione di dipendenza online: https://riptutorial.com/it/cdi/topic/9637/iniezione-di-dipendenza

Capitolo 3: Scopes

Osservazioni

Perché ho bisogno di questi costruttori no-arg?

Cosa succede se un bean con scope di sessione viene iniettato in un bean con scope dell'applicazione? In che modo il bean con ambito applicazione ottiene l'istanza del bean con scope sessione corretta per ogni richiesta? Il bean con scope della sessione non si estenderebbe in altre richieste? Come funziona? Per facilitare lo scoping, CDI utilizza ciò che è noto come proxy. Quando CDI inietta un bean con scope non dipendente in un altro oggetto, non inserisce direttamente il bean. Invece, sottoclasse quel bean per creare ciò che è noto come proxy. Ogni volta che viene chiamato un metodo sul proxy, richiede al runtime CDI di cercare il bean corretto per quell'ambito particolare (se è ambito ambito richiesta, ottiene il bean per quella richiesta.Se ha scope scope, ottieni il bean per quella sessione. .), quindi inoltra la chiamata all'oggetto reale, restituendo qualsiasi risultato per i metodi non void. Ciò significa che quanto segue è una buona cosa da fare:

```
@ApplicationScoped
public class ApplicationScopedClass {

    private final RequestScopedClass requestScopedClass;

    @Inject
    public ApplicationScopedClass(RequestScopedClass requestScopedClass) {
        this.requestScopedClass = requestScopedClass;
    }

    public ApplicationScopedClass() {

    }

    public doSomething() {
        requestScopedClass.doSomethingRequestSpecific(); //This works, because of the proxy
    }

}
```

Tuttavia, per fare in modo che una classe leghi di sottoclasse un'altra classe in Java, deve avere un costruttore valido. E un costruttore valido deve chiamare un altro costruttore sulla classe o il costruttore della classe genitore (ad esempio `super()`). Per semplificare questa sottoclasse, le specifiche CDI richiedono che qualsiasi bean con scope non dipendente fornisca un costruttore public no-args, in modo che il runtime non debba cercare di indovinare cosa fare con una classe che deve essere sottoposta a proxy .

Examples

L'ambito predefinito

```
public class DependentScopedClass {

    //This class has no scoping annotations, so a new instance gets created at every injection
point.

    @Inject
    public DependentScopedClass(SomeDependency someDependency) {
        doSomethingWith(someDependency);
    }

}
```

L'ambito predefinito per la maggior parte dei bean CDI viene definito come ambito dipendente. Una classe che non contiene alcuna annotazione dell'ambito verrà considerata come ambito dipendente, a **meno** che **non** si tratti di una risorsa o provider JAX-RS (risorse predefinite per la richiesta dell'ambito e provider predefiniti per ambito ambito). Un'istanza di una classe di ambito dipendente vive finché l'oggetto in cui viene iniettato non esiste. Ogni volta che una classe viene costruita e ha una classe di scope dipendente come dipendenza, la classe scoped dipendente viene creata e iniettata *direttamente* nell'oggetto che ne ha bisogno. Il significato di questo sarà evidente dopo gli esempi successivi.

@RequestScoped

```
@RequestScoped
public class RequestScopedClass {
    //This class gets constructed once per Servlet request, and is shared among all CDI-
managed classes within that request.

    @Inject
    public RequestScopedClass(SomeDependency someDependency) {
        doSomethingWith(someDependency);
    }

    public RequestScopedClass() {
        //Note that it is required that a request scoped class have a public no-args
constructor
    }

}
```

Se un bean è annotato con @RequestScoped, verrà creato una volta per ogni richiesta. Se due oggetti dipendono da una classe con scope richiesta, entrambi otterranno riferimenti allo stesso oggetto.

Nota: qualsiasi bean con scope richiesta deve avere un costruttore public no-args. Il motivo per questo verrà spiegato più avanti.

@ApplicationScoped

```
@ApplicationScoped
public class ApplicationScopedClass {
    //This class gets constructed once for the entire life of the application, and is shared
among all CDI-managed classes throughout the life of the application.
```

```
    @Inject
    public ApplicationScopedClass(SomeDependency someDependency) {
        doSomethingWith(someDependency);
    }

    public ApplicationScopedClass() {
        //Note that it is required that an application scoped class have a public no-args
constructor
    }

}
```

Le classi con @ApplicationScoped vengono create una sola volta e ogni oggetto che dipende dalla classe condivide la stessa istanza. Queste classi sono singleton "effettivi", tuttavia va notato che non c'è nulla che impedisca a un programmatore di creare manualmente ulteriori istanze della classe. Pertanto, l'utilizzo di @ApplicationScoped è utile per condividere il contesto in tutta l'applicazione o come ottimizzazione delle prestazioni (se la classe è costosa per costruire istanze di), ma non dovrebbe essere considerato come misura di integrità per garantire solo un'istanza di la classe esiste

Come per i bean con scope richiesta, i bean con scope delle applicazioni devono avere un costruttore public no-args.

@SessionScoped

```
@SessionScoped
public class SessionScopedClass implements Serializable {
    //This class gets constructed once per session, and is shared among all CDI-managed
classes within that session. Notice that it implements Serializable, since the instance will
get put on the session.

    @Inject
    public SessionScopedClass(SomeDependency someDependency) {
        doSomethingWith(someDependency);
    }

    public SessionScopedClass() {
        //Note that it is required that a session scoped class have a public no-args
constructor
    }

}
```

Le classi annotate con @SessionScoped verranno create una volta per sessione e due oggetti all'interno della stessa sessione condivideranno la stessa istanza della classe con ambito sessione.

È importante notare che la classe con scope della sessione deve implementare Serializable. Questo requisito esiste perché il bean verrà archiviato nella sessione del contenitore del servlet per quella specifica istanza dell'ambito della sessione. In generale, qualsiasi oggetto inserito nella sessione deve essere serializzabile per due motivi: in primo luogo, le sessioni possono essere mantenute su disco dopo una certa quantità di inattività per risparmiare memoria. Questo è noto

come passivazione della sessione e richiede la serializzazione. La seconda ragione è che nei cluster ad alta disponibilità, la replica di sessione viene spesso utilizzata per consentire a qualsiasi server nel cluster di soddisfare una richiesta per una determinata sessione. Questo generalmente richiede anche la serializzazione.

Proprio come la richiesta scope e l'ambito dell'applicazione, le classi con scope di sessione devono avere un costruttore public no-args.

Leggi Scopes online: https://riptutorial.com/it/cdi/topic/9641/scopes

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con cdi	Community, Dogs, Michael Remijan
2	Iniezione di dipendenza	Dogs
3	Scopes	Dogs