

Errai

Errai CDI Reference Guide

Preface	v
1. Document Conventions	v
2. Feedback	v
1. CDI Introduction	1
2. Errai CDI Features	3
2.1. Integration with the CDI event subsystem	3
2.1.1. Conversational events	4
2.2. RPC Style Invocations on CDI beans	5
2.3. Publish/Subscribe with CDI managed components	5
2.4. CDI Producers	5
3. Client-Server Event Example	7
4. Deploying Errai CDI	11
4.1. Deployment in Development Mode	11
4.2. Deployment to a Servlet Engine	12
4.3. Deployment to an Application Server	12
4.4. Configuration Options	13
5. License and EULA	15
6. CDI Introduction	17
7. Errai CDI Features	19
7.1. Integration with the CDI event subsystem	19
7.1.1. Conversational events	20
7.2. RPC Style Invocations on CDI beans	21
7.3. Publish/Subscribe with CDI managed components	21
7.4. CDI Producers	21
8. Client-Server Event Example	23
9. Deploying Errai CDI	27
9.1. Deployment in Development Mode	27
9.2. Deployment to a Servlet Engine	28
9.3. Deployment to an Application Server	28
9.4. Configuration Options	29
10. License	31
A. Revision History	33

Preface

1. Document Conventions

2. Feedback

CDI Introduction

CDI (Contexts and Dependency Injection) is the Java EE standard (JSR-299) for handling dependency injection. In addition to dependency injection, the standard encompasses component lifecycle, application configuration, call-interception and a decoupled, type-safe eventing specification.

The Errai CDI extension implements a subset of the specification for use inside of client-side applications within Errai, as well as additional capabilities such as distributed eventing.

Errai CDI does not currently implement all life cycles specified in JSR-299 or interceptors. These deficiencies may be addressed in future versions.



Important

The Errai CDI extension itself is implemented on top of the *Errai IOC* [<https://docs.jboss.org/author/pages/viewpage.action?pageId=5931397>] Framework, which itself implements the JSR-330 specification. Inclusion of the CDI module your GWT project will result in the extensions automatically being loaded and made available to your application.

Errai CDI Features

Beans that are deployed to a CDI container will automatically be registered with Errai and exposed to your GWT client application. So, you can use Errai to communicate between your GWT client components and your CDI backend beans. There are several very easy-to-use options:

- Wiring up your GWT application with the CDI event subsystem
- RPC style invocations on beans through a typed interface
- Access beans in a publish/subscribe manner

Further, Errai enables you to make use of CDI producer methods and fields in your GWT client!

2.1. Integration with the CDI event subsystem

Any CDI managed component may produce and consume [events](http://docs.jboss.org/weld/reference/latest/en-US/html/events.html) [http://docs.jboss.org/weld/reference/latest/en-US/html/events.html] . This allows beans to interact in a completely decoupled fashion. Beans consume events by registering for a particular event type and optional qualifiers. The Errai CDI extension simply extends this concept into the client tier. A GWT client application can simply register an `Observer` for a particular event type and thus receive events that are produced on the server-side. Likewise and using the same API, GWT clients can produce events that are consumed by a server-side observer.

Let's take a look at an example.

Example 2.1. FraudClient.java

```
public class FraudClient extends LayoutPanel {

    @Inject
    private Event<AccountActivity> event; (1)

    private HTML responsePanel;

    public FraudClient() {
        super(new BoxLayout(BoxLayout.Orientation.VERTICAL));
    }

    @PostConstruct
    public void buildUI() {
        Button button = new Button("Create activity", new ClickHandler() {
            public void onClick(ClickEvent clickEvent) {
                event.fire(new AccountActivity());
            }
        });
    }
}
```

```
responsePanel = new HTML();
add(button);
add(responsePanel);
}

public void processFraud(@Observes @Detected Fraud fraudEvent) { (2)
    responsePanel.setText("Fraud detected: " + fraudEvent.getTimestamp());
}
}
```

Two things are noteworthy in this example:

1. Injection of an `Event` dispatcher proxy
2. Creation of an `Observer` method for a particular event type

The event dispatcher is responsible for sending events created on the client-side to the server-side event subsystem (CDI container). This means any event that is fired through a dispatcher will eventually be consumed by a CDI managed bean, if there is an corresponding `Observer` registered for it on the server side.

In order to consume events that are created on the server-side you need to declare an client-side observer method for a particular event type. In case an event is fired on the server this method will be invoked with an event instance of type you declared.

To complete the example, let's look at the corresponding server-side CDI bean:

Example 2.2. AccountService.java

```
@ApplicationScoped
public class AccountService {

    @Inject @Detected
    private Event<Fraud> event;

    public void watchActivity(@Observes AccountActivity activity) {
        Fraud fraud = new Fraud(System.currentTimeMillis());
        event.fire(fraud);
    }
}
```

2.1.1. Conversational events

A server can address a single client in response to an event by using `@Conversational`. Consider a service that responds to a subscription event. Naturally, only the newly subscribed client should receive the response.

Example 2.3. SubscriptionService.java

```
@ApplicationScoped
public class SubscriptionService {

    @Inject
    private Event<Documents> welcomeEvent;

    @Conversational
    public void onSubscription(@Observes Subscription subscription) {
        Document docs = createWelcomePackage(subscription);
        welcomeEvent.fire(docs);
    }
}
```

2.2. RPC Style Invocations on CDI beans

When choosing RPC style invocations on beans, you basically rely on a typed java interface the CDI managed bean needs to expose. A GWT client component can then create an invocation proxy based on this interface. For more information see [chapter on RPC mechanism](https://docs.jboss.org/author/pages/viewpage.action?pageId=5931313) [https://docs.jboss.org/author/pages/viewpage.action?pageId=5931313] .

2.3. Publish/Subscribe with CDI managed components

If you choose publish/subscribe then your CDI bean needs to implement the `MessageCallback` interface, as described in chapter [Messaging](https://docs.jboss.org/author/pages/viewpage.action?pageId=5931263) [https://docs.jboss.org/author/pages/viewpage.action?pageId=5931263] . Any bean exposed in this way can be accessed through the [MessageBuilderAPI](https://docs.jboss.org/author/pages/viewpage.action?pageId=5931280) [https://docs.jboss.org/author/pages/viewpage.action?pageId=5931280] .

2.4. CDI Producers

Producer methods and fields act as sources of objects to be injected. They are useful when additional control over object creation is needed before injections can take place e.g. when you need to make a decision at runtime before an object can be created and injected.

Example 2.4. App.java

```
@EntryPoint
public class App {
    ...

    @Produces @Supported
    public MyBaseWidget createWidget() {
```

```
        return (Canvas.isSupported()) ? new MyHtml5Widget() : new MyDefaultWidget();
    }
}
```

Example 2.5. MyComposite.java

```
@ApplicationScoped
public class MyComposite extends Composite {

    @Inject @Supported
    public MyBaseWidget widget;

    ...
}
```

For more information on CDI producers, see the [CDI specification](http://docs.jboss.org/cdi/spec/1.0/html/) [http://docs.jboss.org/cdi/spec/1.0/html/] and the [WELD reference documentation](http://seamframework.org/Weld/WeldDocumentation) [http://seamframework.org/Weld/WeldDocumentation] .

Client-Server Event Example

A key feature of the Errai CDI framework is the ability to federate the CDI eventing bus between the client and the server. This permits the observation of server produced events on the client, and vice-versa.

Example server code:

Example 3.1. MyServerBean.java

```
@ApplicationScoped
public class MyServerBean {
    @Inject
    Event<MyResponseEvent> myResponseEvent;

    public void myClientObserver(@Observes MyRequestEvent event) {
        MyResponseEvent response;

        if (event.isThankYou()) {
            // aww, that's nice!
            response = new MyResponseEvent("Well, you're welcome!");
        }
        else {
            // how rude!
            response = new MyResponseEvent("What? Nobody says 'thank you' anymore?");
        }

        myResponseEvent.fire(response);
    }
}
```

Domain-model:

Example 3.2. MyRequestEvent.java

```
@ExposeEntity
public class MyRequestEvent {
    private boolean thankYou;

    public MyRequestEvent(boolean thankYou) {
        setThankYou(thankYou);
    }

    public void setThankYou(boolean thankYou) {
        this.thankYou = thankYou;
    }
}
```

```
}

public boolean isThankYou() {
    return thankYou;
}
}
```

Example 3.3. MyResponseEvent.java

```
@ExposeEntity
public class MyResponseEvent {
    private String message;

    public MyRequestEvent(String message) {
        setMessage(message);
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}
```

Client application logic:

Example 3.4. MyClientBean.java

```
@EntryPoint
public class MyClientBean {
    @Inject
    Event<MyRequestEvent> requestEvent;

    public void myResponseObserver(@Observes MyResponseEvent event) {
        Window.alert("Server replied: " + event.getMessage());
    }

    @PostConstruct
    public void init() {
        Button thankYou = new Button("Say Thank You!");
        thankYou.addClickListener(new ClickHandler() {
            public void onClick(ClickEvent event) {
                requestEvent.fire(new MyRequestEvent(true));
            }
        });
    }
}
```

```
    }  
    }  
  
    Button nothing = new Button("Say nothing!");  
    nothing.addClickHandler(new ClickHandler() {  
        public void onClick(ClickEvent event) {  
            requestEvent.fire(new MyRequestEvent(false));  
        }  
    })  
  
    VerticalPanel vPanel = new VerticalPanel();  
    vPanel.add(thankYou);  
    vPanel.add(nothing);  
  
    RootPanel.get().add(vPanel);  
}  
}
```


Deploying Errai CDI

If you do not care about the deployment details for now and just want to get started take a look at the [CDI Quickstart Guide](https://docs.jboss.org/author/pages/viewpage.action?pageId=5931395) [https://docs.jboss.org/author/pages/viewpage.action?pageId=5931395] .

The CDI integration is a plugin to the Errai core framework and represents a CDI portable extension. Which means it is discovered automatically by both Errai and the CDI container. In order to use it, you first need to understand the different runtime models involved when working GWT, Errai and CDI.

Typically a GWT application lifecycle begins in [Development Mode](http://code.google.com/webtoolkit/doc/latest/DevGuideCompilingAndDebugging.html) [http://code.google.com/webtoolkit/doc/latest/DevGuideCompilingAndDebugging.html] and finally a web application containing the GWT client code will be deployed to a target container (Servlet Engine, Application Server). This is no way different when working with CDI components to back your application.

What's different however is availability of the CDI container across the different runtimes. In GWT development mode and in a pure servlet environment you need to provide and bootstrap the CDI environment on your own. While any Java EE 6 Application Server already provides a preconfigured CDI container. To accomodate these differences, we need to do a little trickery when executing the GWT Development Mode and packaging our application for deployment.

4.1. Deployment in Development Mode

In development mode we need to bootstrap the CDI environment on our own and make both Errai and CDI available through JNDI (common denominator across all runtimes). Since GWT uses Jetty, that only supports read only JNDI, we need to replace the default Jetty launcher with a custom one that will setup the JNDI bindings:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>gwt-maven plugin</artifactId>
  <version>${gwt.maven}</version>

  <configuration>
    ...
    <server>org.jboss.errai.cdi.server.gwt.JettyLauncher</server>
  </configuration>
  <executions>
    ...
  </executions>
</plugin>
```



Starting Development Mode from within your IDE

Consequently, when starting Development Mode from within your IDE the following program argument has to be provided: `-server org.jboss.errai.cdi.server.gwt.JettyLauncher`

Once this is set up correctly, we can bootstrap the CDI container through a servlet listener:

```
<web-app>
...
<listener>
  <listener-class>org.jboss.errai.container.DevModeCDIBootstrap</listener-
class>
</listener>

<resource-env-ref>
  <description>Object factory for the CDI Bean Manager</description>
  <resource-env-ref-name>BeanManager</resource-env-ref-name>
  <resource-env-ref-type>javax.enterprise.inject.spi.BeanManager</resource-
env-ref-type>
</resource-env-ref>
...
</web-app>
```



Errai-CDI maven archetype

Sounds terribly complicated, no? Don't worry we provide a [maven archetype](https://docs.jboss.org/author/pages/viewpage.action?pageId=5931395) [https://docs.jboss.org/author/pages/viewpage.action?pageId=5931395] that takes care of all these setup steps and configuration details.

4.2. Deployment to a Servlet Engine

Deployment to servlet engine has basically the same requirements as running in development mode. You need to include the servlet listener that bootstraps the CDI container and make sure both Errai and CDI are accessible through JNDI. For Jetty you can re-use the artefacts we ship with the archetype. In case you want to run on tomcat, please consult the [Apache Tomcat Documentation](http://tomcat.apache.org/tomcat-5.5-doc/jndi-resources-howto.html) [http://tomcat.apache.org/tomcat-5.5-doc/jndi-resources-howto.html] .

4.3. Deployment to an Application Server

We provide integration with the [JBoss Application Server](http://jboss.org/jbossas) [http://jboss.org/jbossas] , but the requirements are basically the same for other vendors. When running a GWT client app that

leverages CDI beans on a Java EE 6 application server, CDI is already part of the container and accessible through JNDI (`java: /BeanManager`).

4.4. Configuration Options

Since the discovery of service implementations (beans) is delegated to the CDI container, we need to disable Errai's own service discovery mechanism. In order to do so, simply turn off the auto-discovery feature in `ErraiService.properties`

```
errai.auto_discover_services=false
```


License and EULA

Errai CDI is distributed under the terms of the Apache License, Version 2.0. See [the full Apache license text](http://www.apache.org/licenses/LICENSE-2.0) [http://www.apache.org/licenses/LICENSE-2.0] .

CDI Introduction

CDI (Contexts and Dependency Injection) is the Java EE standard (JSR-299) for handling dependency injection. In addition to dependency injection, the standard encompasses component lifecycle, application configuration, call-interception and a decoupled, type-safe eventing specification.

The Errai CDI extension implements a subset of the specification for use inside of client-side applications within Errai, as well as additional capabilities such as distributed eventing.

Errai CDI does not currently implement all life cycles specified in JSR-299 or interceptors. These deficiencies may be addressed in future versions.



Important

The Errai CDI extension itself is implemented on top of the *Errai IOC* [<https://docs.jboss.org/author/pages/viewpage.action?pageId=5931397>] Framework, which itself implements the JSR-330 specification. Inclusion of the CDI module your GWT project will result in the extensions automatically being loaded and made available to your application.

Errai CDI Features

Beans that are deployed to a CDI container will automatically be registered with Errai and exposed to your GWT client application. So, you can use Errai to communicate between your GWT client components and your CDI backend beans. There are several very easy-to-use options:

- Wiring up your GWT application with the CDI event subsystem
- RPC style invocations on beans through a typed interface
- Access beans in a publish/subscribe manner

Further, Errai enables you to make use of CDI producer methods and fields in your GWT client!

7.1. Integration with the CDI event subsystem

Any CDI managed component may produce and consume [events](http://docs.jboss.org/weld/reference/latest/en-US/html/events.html) [http://docs.jboss.org/weld/reference/latest/en-US/html/events.html] . This allows beans to interact in a completely decoupled fashion. Beans consume events by registering for a particular event type and optional qualifiers. The Errai CDI extension simply extends this concept into the client tier. A GWT client application can simply register an `Observer` for a particular event type and thus receive events that are produced on the server-side. Likewise and using the same API, GWT clients can produce events that are consumed by a server-side observer.

Let's take a look at an example.

Example 7.1. FraudClient.java

```
public class FraudClient extends LayoutPanel {

    @Inject
    private Event<AccountActivity> event; (1)

    private HTML responsePanel;

    public FraudClient() {
        super(new BoxLayout(BoxLayout.Orientation.VERTICAL));
    }

    @PostConstruct
    public void buildUI() {
        Button button = new Button("Create activity", new ClickHandler() {
            public void onClick(ClickEvent clickEvent) {
                event.fire(new AccountActivity());
            }
        });
    }
}
```

```
responsePanel = new HTML();
add(button);
add(responsePanel);
}

public void processFraud(@Observes @Detected Fraud fraudEvent) { (2)
    responsePanel.setText("Fraud detected: " + fraudEvent.getTimestamp());
}
}
```

Two things are noteworthy in this example:

1. Injection of an `Event` dispatcher proxy
2. Creation of an `Observer` method for a particular event type

The event dispatcher is responsible for sending events created on the client-side to the server-side event subsystem (CDI container). This means any event that is fired through a dispatcher will eventually be consumed by a CDI managed bean, if there is an corresponding `Observer` registered for it on the server side.

In order to consume events that are created on the server-side you need to declare an client-side observer method for a particular event type. In case an event is fired on the server this method will be invoked with an event instance of type you declared.

To complete the example, let's look at the corresponding server-side CDI bean:

Example 7.2. AccountService.java

```
@ApplicationScoped
public class AccountService {

    @Inject @Detected
    private Event<Fraud> event;

    public void watchActivity(@Observes AccountActivity activity) {
        Fraud fraud = new Fraud(System.currentTimeMillis());
        event.fire(fraud);
    }
}
```

7.1.1. Conversational events

A server can address a single client in response to an event by using `@Conversational`. Consider a service that responds to a subscription event. Naturally, only the newly subscribed client should receive the response.

Example 7.3. SubscriptionService.java

```
@ApplicationScoped
public class SubscriptionService {

    @Inject
    private Event<Documents> welcomeEvent;

    @Conversational
    public void onSubscription(@Observes Subscription subscription) {
        Document docs = createWelcomePackage(subscription);
        welcomeEvent.fire(docs);
    }
}
```

7.2. RPC Style Invocations on CDI beans

When choosing RPC style invocations on beans, you basically rely on a typed java interface the CDI managed bean needs to expose. A GWT client component can then create an invocation proxy based on this interface. For more information see [chapter on RPC mechanism](https://docs.jboss.org/author/pages/viewpage.action?pageId=5931313) [https://docs.jboss.org/author/pages/viewpage.action?pageId=5931313] .

7.3. Publish/Subscribe with CDI managed components

If you choose publish/subscribe then your CDI bean needs to implement the `MessageCallback` interface, as described in chapter [Messaging](https://docs.jboss.org/author/pages/viewpage.action?pageId=5931263) [https://docs.jboss.org/author/pages/viewpage.action?pageId=5931263] . Any bean exposed in this way can be accessed through the [MessageBuilderAPI](https://docs.jboss.org/author/pages/viewpage.action?pageId=5931280) [https://docs.jboss.org/author/pages/viewpage.action?pageId=5931280] .

7.4. CDI Producers

Producer methods and fields act as sources of objects to be injected. They are useful when additional control over object creation is needed before injections can take place e.g. when you need to make a decision at runtime before an object can be created and injected.

Example 7.4. App.java

```
@EntryPoint
public class App {
    ...

    @Produces @Supported
    public MyBaseWidget createWidget() {
```

```
        return (Canvas.isSupported()) ? new MyHtml5Widget() : new MyDefaultWidget();
    }
}
```

Example 7.5. MyComposite.java

```
@ApplicationScoped
public class MyComposite extends Composite {

    @Inject @Supported
    public MyBaseWidget widget;

    ...
}
```

For more information on CDI producers, see the [CDI specification](http://docs.jboss.org/cdi/spec/1.0/html/) [http://docs.jboss.org/cdi/spec/1.0/html/] and the [WELD reference documentation](http://seamframework.org/Weld/WeldDocumentation) [http://seamframework.org/Weld/WeldDocumentation] .

Client-Server Event Example

A key feature of the Errai CDI framework is the ability to federate the CDI eventing bus between the client and the server. This permits the observation of server produced events on the client, and vice-versa.

Example server code:

Example 8.1. MyServerBean.java

```
@ApplicationScoped
public class MyServerBean {
    @Inject
    Event<MyResponseEvent> myResponseEvent;

    public void myClientObserver(@Observes MyRequestEvent event) {
        MyResponseEvent response;

        if (event.isThankYou()) {
            // aww, that's nice!
            response = new MyResponseEvent("Well, you're welcome!");
        }
        else {
            // how rude!
            response = new MyResponseEvent("What? Nobody says 'thank you' anymore?");
        }

        myResponseEvent.fire(response);
    }
}
```

Domain-model:

Example 8.2. MyRequestEvent.java

```
@ExposeEntity
public class MyRequestEvent {
    private boolean thankYou;

    public MyRequestEvent(boolean thankYou) {
        setThankYou(thankYou);
    }

    public void setThankYou(boolean thankYou) {
        this.thankYou = thankYou;
    }
}
```

```
}

public boolean isThankYou() {
    return thankYou;
}
}
```

Example 8.3. MyResponseEvent.java

```
@ExposeEntity
public class MyResponseEvent {
    private String message;

    public MyRequestEvent(String message) {
        setMessage(message);
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}
```

Client application logic:

Example 8.4. MyClientBean.java

```
@EntryPoint
public class MyClientBean {
    @Inject
    Event<MyRequestEvent> requestEvent;

    public void myResponseObserver(@Observes MyResponseEvent event) {
        Window.alert("Server replied: " + event.getMessage());
    }

    @PostConstruct
    public void init() {
        Button thankYou = new Button("Say Thank You!");
        thankYou.addClickListener(new ClickHandler() {
            public void onClick(ClickEvent event) {
                requestEvent.fire(new MyRequestEvent(true));
            }
        });
    }
}
```

```
    }  
}  
  
Button nothing = new Button("Say nothing!");  
nothing.addClickHandler(new ClickHandler() {  
    public void onClick(ClickEvent event) {  
        requestEvent.fire(new MyRequestEvent(false));  
    }  
})  
  
VerticalPanel vPanel = new VerticalPanel();  
vPanel.add(thankYou);  
vPanel.add(nothing);  
  
RootPanel.get().add(vPanel);  
}  
}
```


Deploying Errai CDI

If you do not care about the deployment details for now and just want to get started take a look at the [CDI Quickstart Guide](https://docs.jboss.org/author/pages/viewpage.action?pageId=5931395) [https://docs.jboss.org/author/pages/viewpage.action?pageId=5931395] .

The CDI integration is a plugin to the Errai core framework and represents a CDI portable extension. Which means it is discovered automatically by both Errai and the CDI container. In order to use it, you first need to understand the different runtime models involved when working GWT, Errai and CDI.

Typically a GWT application lifecycle begins in [Development Mode](http://code.google.com/webtoolkit/doc/latest/DevGuideCompilingAndDebugging.html) [http://code.google.com/webtoolkit/doc/latest/DevGuideCompilingAndDebugging.html] and finally a web application containing the GWT client code will be deployed to a target container (Servlet Engine, Application Server). This is no way different when working with CDI components to back your application.

What's different however is availability of the CDI container across the different runtimes. In GWT development mode and in a pure servlet environment you need to provide and bootstrap the CDI environment on your own. While any Java EE 6 Application Server already provides a preconfigured CDI container. To accomodate these differences, we need to do a little trickery when executing the GWT Development Mode and packaging our application for deployment.

9.1. Deployment in Development Mode

In development mode we need to bootstrap the CDI environment on our own and make both Errai and CDI available through JNDI (common denominator across all runtimes). Since GWT uses Jetty, that only supports read only JNDI, we need to replace the default Jetty launcher with a custom one that will setup the JNDI bindings:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>gwt-maven plugin</artifactId>
  <version>${gwt.maven}</version>

  <configuration>
    ...
    <server>org.jboss.errai.cdi.server.gwt.JettyLauncher</server>
  </configuration>
  <executions>
    ...
  </executions>
</plugin>
```



Starting Development Mode from within your IDE

Consequently, when starting Development Mode from within your IDE the following program argument has to be provided: `-server org.jboss.errai.cdi.server.gwt.JettyLauncher`

Once this is set up correctly, we can bootstrap the CDI container through a servlet listener:

```
<web-app>
...
<listener>
  <listener-class>org.jboss.errai.container.DevModeCDIBootstrap</listener-
class>
</listener>

<resource-env-ref>
  <description>Object factory for the CDI Bean Manager</description>
  <resource-env-ref-name>BeanManager</resource-env-ref-name>
  <resource-env-ref-type>javax.enterprise.inject.spi.BeanManager</resource-
env-ref-type>
</resource-env-ref>
...
</web-app>
```



Errai-CDI maven archetype

Sounds terribly complicated, no? Don't worry we provide a [maven archetype](https://docs.jboss.org/author/pages/viewpage.action?pageId=5931395) [https://docs.jboss.org/author/pages/viewpage.action?pageId=5931395] that takes care of all these setup steps and configuration details.

9.2. Deployment to a Servlet Engine

Deployment to servlet engine has basically the same requirements as running in development mode. You need to include the servlet listener that bootstraps the CDI container and make sure both Errai and CDI are accessible through JNDI. For Jetty you can re-use the artefacts we ship with the archetype. In case you want to run on tomcat, please consult the [Apache Tomcat Documentation](http://tomcat.apache.org/tomcat-5.5-doc/jndi-resources-howto.html) [http://tomcat.apache.org/tomcat-5.5-doc/jndi-resources-howto.html] .

9.3. Deployment to an Application Server

We provide integration with the [JBoss Application Server](http://jboss.org/jbossas) [http://jboss.org/jbossas] , but the requirements are basically the same for other vendors. When running a GWT client app that

leverages CDI beans on a Java EE 6 application server, CDI is already part of the container and accessible through JNDI (`java: /BeanManager`).

9.4. Configuration Options

Since the discovery of service implementations (beans) is delegated to the CDI container, we need to disable Errai's own service discovery mechanism. In order to do so, simply turn off the auto-discovery feature in `ErraiService.properties`

```
errai.auto_discover_services=false
```


License

Errai CDI is distributed under the terms of the Apache License, Version 2.0. See [the full Apache license text](http://www.apache.org/licenses/LICENSE-2.0) [http://www.apache.org/licenses/LICENSE-2.0] .

Appendix A. Revision History

Revision History			
Revision ToDo 0-0	ToDo Wed Jan 19 2011	ToDo	DudeToDo
		McPants<ToDo	Dude.McPants@example.com>
ToDo Initial creation of book			

