Using Asynchronous Method Invocation in Session Beans

Discusses how to implement asynchronous business methods in session beans, and call them from enterprise bean clients.

The following topics are addressed here:

- Asynchronous Method Invocation
- The `async` Example Application

Asynchronous Method Invocation

Session beans can implement asynchronous methods, business methods where control is returned to the client by the enterprise bean container before the method is invoked on the session bean instance.

Clients may then use the Java SE concurrency API to retrieve the result, cancel the invocation, and check for exceptions.

Asynchronous methods are typically used for long-running operations, for processor-intensive tasks, for background tasks, to increase application throughput, or to improve application response time if the method invocation result isn't required immediately.

When a session bean client invokes a typical non-asynchronous business method, control is not returned to the client until the method has completed.

Clients calling asynchronous methods, however, immediately have control returned to them by the enterprise bean container.

This allows the client to perform other tasks while the method invocation completes.

If the method returns a result, the result is an implementation of the `java.util.concurrent.Future<V>` interface, where "V" is the result value type.

The `Future<V>` interface defines methods the client may use to check if the computation is completed, wait for the invocation to complete, retrieve the final result, and cancel the invocation.

Creating an Asynchronous Business Method

Annotate a business method with `javax.ejb.Asynchronous` to mark that method as an asynchronous method, or apply `@Asynchronous` at the class level to mark all the business methods of the session bean as asynchronous methods.

Session bean methods that expose web services can't be asynchronous.

Asynchronous methods must return either `void` or an implementation of the `Future<V>` interface.

Asynchronous methods that return `void` can't declare application exceptions, but if they return `Future<V>`, they may declare application exceptions.

For example:

```
@Asynchronous
public Future<String>
processPayment(Order order)
throws PaymentException { ... }
```

This method will attempt to process the payment of an order, and return the status as a `String`.

Even if the payment processor takes a long time, the client can continue working, and display the result when the processing finally completes.

The `javax.ejb.AsyncResult<V>` class is a concrete implementation of the `Future<V>` interface provided as a helper class for returning asynchronous results.

`AsyncResult` has a constructor with the result as a parameter, making it easy to create `Future<V>` implementations.

For example, the `processPayment` method would use `AsyncResult` to return the status as a `String`:

```
@Asynchronous
public Future<String>
processPayment(Order order)
throws PaymentException { ...
String status = ...;
```

```
return
new AsyncResult<String>(status);
}
```

The result is returned to the enterprise bean container, not directly to the client, and the enterprise bean container makes the result available to the client.

The session bean can check if the client requested that the invocation be cancelled by calling the `javax.ejb.SessionContext.wasCancelled` method.

For example:

```
@Asynchronous
```

```
public Future<String>
processPayment(Order order) throws
PaymentException { ...
if(SessionContext.wasCancelled()) {
// clean up
} else {
// process the payment
}
...
}
```

Calling Asynchronous Methods from Enterprise Bean Clients

Session bean clients call asynchronous methods just like non-asynchronous business methods.

If the asynchronous method returns a result, the client receives a `Future<V>` instance as soon as the method is invoked.

This instance can be used to retrieve the final result, cancel the invocation, check whether the invocation has completed, check if there were any exceptions thrown during processing, and check if the invocation was cancelled.

Retrieving the Final Result from an Asynchronous Method Invocation

The client may retrieve the result using one of the `Future<V>.get` methods.

If processing hasn't completed by the session bean handling the invocation, calling one of the `get` methods will result in the client halting execution until the invocation completes.

Use the `Future<V>.isDone` method to determine if processing has completed before calling one of the `get` methods.

The `get()` method returns the result as the type specified in the type value of the `Future<V>` instance.

For example, calling `Future<String>.get()` will return a `String` object.

If the method invocation was cancelled, calls to `get()` result in a `java.util.concurrent.CancellationException` being thrown.

If the invocation resulted in an exception during processing by the session bean, calls to `get()` result in a `java.util.concurrent.ExecutionException` being thrown.

The cause of the `ExecutionException` may be retrieved by calling the `ExecutionException.getCause` method.

The `get(long timeout, java.util.concurrent.TimeUnit unit)` method is similar to the `get()` method, but allows the client to set a timeout value.

If the timeout value is exceeded, a `java.util.concurrent.TimeoutException` is thrown.

See the Javadoc for the `TimeUnit` class for the available units of time to specify the timeout value.

Cancelling an Asynchronous Method Invocation

Call the `cancel(boolean mayInterruptIfRunning)` method on the `Future<V>` instance to attempt to cancel the method invocation.

The `cancel` method returns `true` if the cancellation was successful, and `false` if the method invocation cannot be cancelled.

When the invocation cannot be cancelled, the `mayInterruptIfRunning` parameter is used to alert the session bean instance on which the method invocation is running that the client attempted to cancel the invocation.

If `mayInterruptIfRunning` is set to `true`, calls to `SessionContext.wasCancelled` by the session bean instance will return `true`.

If `mayInterruptIfRunning` is to set `false`, calls to `SessionContext.wasCancelled` by the session bean instance will return `false`.

The `Future<V>.isCancelled` method is used to check if the method invocation was cancelled before the asynchronous method invocation completed by calling `Future<V>.cancel`.

The `isCancelled` method returns `true` if the invocation was cancelled.

Checking the Status of an Asynchronous Method Invocation

The `Future<V>.isDone` method returns `true` if the session bean instance completed processing the method invocation.

The `isDone` method returns `true` if the asynchronous method invocation completed normally, was cancelled, or resulted in an exception.

That is, `isDone` only indicates whether the session bean has completed processing the invocation.

The `async` Example Application

The `async` example demonstrates how to define an asynchronous business method on a session bean, and call it from a web client.

The `MailerBean` stateless session bean defines an asynchronous method, `sendMessage`, which uses the JavaMail API to send an email to a specified email address.

Note - This example needs to be configured for your environment before it runs correctly, and requires access to an SMTPS server.

Architecture of the `async` Example Application

The `async` application consists of a single stateless session bean, `MailerBean`, and a JavaServer Faces web application front-end that uses Facelets tags in XHTML files to display a form for users to enter the email address for the recipient of an email.

The status of the email is updated when the email is finally sent.

The `MailerBean` session bean injects a JavaMail resource used to send an email message to an address specified by the user.

The message is created, modified, and sent using the JavaMail API.

The injected JavaMail resource is configured through the GlassFish Server Administration Console, or through a resource configuration file packaged with the application.

The resource configuration can be modified at runtime by GlassFish Server administrator to use a different mail server or transport protocol.

```
@Asynchronous
public Future<String>
sendMessage(String email) {
String status;
try {
Message message =
new MimeMessage(session);
message.setFrom();
```

```
message.setRecipients
 (Message.RecipientType.TO,
InternetAddress.parse
 (email, false));
message.setSubject
 ("Test message from async example");
message.setHeader
 ("X-Mailer", "JavaMail");
```

```
DateFormat dateFormatter =
DateFormat.getDateTimeInstance
(DateFormat.LONG,DateFormat.SHORT);
Date timeStamp = new Date();
String messageBody =
"This is a test message from the async example "
+ "of the Java EE Tutorial.
It was sent on "
+ dateFormatter.format(timeStamp)
+ ".";
message.setText(messageBody);
```

```
message.setSentDate(timeStamp);
Transport.send(message);
status = "Sent";
logger.log
(Level.INFO, "Mail sent to {0}",
email);
}catch(MessagingException ex){
logger.severe
("Error in sending message.");
status = "Encountered an error";
```

```
logger.severe(ex.getMessage() +
ex.getNextException().getMessage());
logger.severe
(ex.getCause().getMessage());
}
return
new AsyncResult<String>(status);
}
```

The web client consists of a Facelets template, `template.xhtml`, two Facelets clients, `index.xhtml` and `response.xhtml`, and a JavaServer Faces managed bean, `MailerManagedBean`.

The `index.xhtml` file contains a form for the target email address.

When the user submits the form, the `MailerManagedBean.send` method is called.

This method uses an injected instance of the `MailerBean` session bean to call `MailerBean.sendMessage`.

The result is sent to the `response.xhtml` Facelets view.

Configuring the Keystore and Truststore in GlassFish Server

The GlassFish Server domain needs to be configured with the server's master password to access the keystore and truststore used to initiate secure communications using the SMTPS transport protocol.

1. Open the GlassFish Server Administration Console in a web browser at `http://localhost:4848`.

2. Expand Configurations, then expand server-config, then click JVM Settings.

3. Click JVM Options, then click Add JVM Option and enter `-Djavax.net.ssl.keyStorePassword=`*`master password`*, replacing *`master password`* with the keystore master password.

The default master password is `changeit`.

4. Click Add JVM Option and enter `-Djavax.net.ssl.trustStorePassword=`*`master password`*, replacing *`master password`* `with the truststore master password.`

The default master password is `changeit`.

5. Click Save, then restart GlassFish Server.

Running the `async` Example Application in NetBeans IDE

Follow these instructions for running the `async` example application in NetBeans IDE.

Before You Begin

Before running this example, you must configure your GlassFish Server instance to access the keystore and truststore used by GlassFish Server to create a secure connection to the target SMTPS server.

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:

   `tut-install/examples/ejb/`

3. Select the `async` folder and click Open Project.

4. Under `async` in the project pane, expand the Server Resources node and double-click `glassfish-resources.xml`.

5. Enter the configuration settings for your SMTPS server in `glassfish-resources.xml`.

The SMTPS server host name is set in the `host` attribute, email address from which you want the message sent is the `from` attribute, the SMTPS user name is the `user` attribute.

Set the `mail-smtps-password` property value to the password for the SMTPS server user.

The following code snippet shows an example resource configuration.

Lines in bold need to be modified.

```
<resources>
<mail-resource debug="false"
enabled="true"
from="user@example.com"
host="smtp.example.com"
jndi-name="mail/myExampleSession"
object-type="user"
store-protocol="imap"
store-protocol-class=
"com.sun.mail.imap.IMAPStore"
```

```
transport-protocol="smtps"
transport-protocol-class=
"com.sun.mail.smtp.SMTPSSLTransport"
user="user@example.com"
>
<description/>
<property name="mail-smtps-auth"
value="true"
/>
```

```
<property
name="mail-smtps-password"
value="mypassword"
/>
</mail-resource>
</resources>
```

6. Right-click `async` in the project pane and select Run.

This will compile, assemble, and deploy the application, and start a web browser at the following URL:

`http://localhost:8080/async`.

7. In the web browser window, enter the email to which you want the test message sent and click Send email.

If your configuration settings are correct, a test email will be sent, and the status message will read `Sent` in the web client.

The test message should appear momentarily in the inbox of the recipient.

If an error occurs, the status will read `Encountered an error.`

Check the `server.log` file for your domain to find the cause of the error.

Running the `async` Example Application Using Ant

Follow these instructions for running the `async` example application using Ant.

1. In a terminal window, navigate to *tut-install*`/examples/ejb/async/`.

2. In a text editor, open `setup/glassfish-resources.xml` and enter the configuration settings for your SMTPS server.

The SMTPS server host name is set in the `host` attribute, email address from which you want the message sent is the `from` attribute, the SMTPS user name is the `user` attribute.

Set the `mail-smtps-password` property value to the password for the SMTPS server user.

The following code snippet shows an example resource configuration.

Lines in bold need to be modified.

```
<resources>
<mail-resource debug="false"
enabled="true"
```

```
from="user@example.com"
host="smtp.example.com"
jndi-name="mail/myExampleSession"
object-type="user"
store-protocol="imap"
store-protocol-class=
"com.sun.mail.imap.IMAPStore"
transport-protocol="smtps"


transport-protocol-class=
```

```
"com.sun.mail.smtp.SMTPSSLTransport"
user="user@example.com">
<description/>
<property name="mail-smtps-auth"
value="true"/>
<property
name="mail-smtps-password"
value="mypassword"/>
</mail-resource>
</resources>
```

3. Enter the following command:

```
ant all
```

This will compile, assemble, and deploy the application, and start a web browser at the following URL:
`http://localhost:8080/async`.

Note - If your build system isn't configured to automatically open a web browser, open the above URL in a browser window.

4. In the web browser window, enter the email to which you want the test message sent and click Send email.

If your configuration settings are correct, a test email will be sent, and the status message will read `Sent` in the web client.

The test message should appear momentarily in the inbox of the recipient.

If an error occurs, the status will read `Encountered an error.`

Check the `server.log` file for your domain to find the cause of the error.