



K9

TREINAMENTOS

Integração de Sistemas com Webservices, JMS e EJB

Integração de Sistemas com Webservices, JMS e EJB

22 de agosto de 2015

As apostilas atualizadas estão disponíveis em www.k19.com.br

Esta apostila contém:

- 158 exercícios de fixação.
- 4 exercícios complementares.
- 0 desafios.
- 0 questões de prova.

Sumário	i
Sobre a K19	1
Seguro Treinamento	2
Termo de Uso	3
Cursos	4
1 JMS	1
1.1 Middleware Orientado a Mensagens - MOM	1
1.2 Destinos: Filas e Tópicos	1
1.3 Exercícios de Fixação	2
1.4 Fábricas de Conexões	19
1.5 Visão Geral	19
1.6 Exercícios de Fixação	21
1.7 Modos de recebimento	38
1.8 Percorrendo uma fila	38
1.9 Exercícios de Fixação	39
1.10 Selecionando mensagens de um tópico	41
1.11 Exercícios de Fixação	41
1.12 Tópicos Duráveis	42
1.13 Exercícios de Fixação	43
1.14 JMS e EJB	46

1.15 Exercícios de Fixação	46
1.16 Projeto - Rede de Hotéis	49
1.17 Exercícios de Fixação	49
2 JAX-WS	51
2.1 Web Services	51
2.2 JAXB	52
2.3 Exercícios de Fixação	53
2.4 Criando um web service - Java SE	55
2.5 Consumindo um web service com JAX-WS	56
2.6 Exercícios de Fixação	57
2.7 Autenticação	60
2.8 Exercícios de Fixação	61
2.9 Exercícios Complementares	62
2.10 JAX-WS e EJB	65
2.11 Exercícios de Fixação	65
2.12 Projeto - Táxi no Aeroporto	67
2.13 Exercícios de Fixação	67
3 JAX-RS	69
3.1 REST vs Padrões W3C	69
3.2 Resources, URIs, Media Types e Operações	69
3.3 Web service com JAX-RS	70
3.4 Resources	70
3.5 Subresource	71
3.6 Exercícios de Fixação	71
3.7 Parâmetros	74
3.8 Exercícios de Fixação	76
3.9 URI Matching	78
3.10 Exercícios de Fixação	79
3.11 HTTP Headers com @Context	79
3.12 Exercícios de Fixação	80
3.13 Download de arquivos	80
3.14 Exercícios de Fixação	81
3.15 Produzindo XML ou JSON	82
3.16 Consumindo XML ou JSON	83
3.17 Exercícios de Fixação	84
3.18 Implementando um Cliente	87
3.19 Exercícios de Fixação	87
3.20 Exercícios Complementares	89



Sobre a K19

A K19 é uma empresa especializada na capacitação de desenvolvedores de software. Sua equipe é composta por profissionais formados em Ciência da Computação pela Universidade de São Paulo (USP) e que possuem vasta experiência em treinamento de profissionais para área de TI.

O principal objetivo da K19 é oferecer treinamentos de máxima qualidade e relacionados às principais tecnologias utilizadas pelas empresas. Através desses treinamentos, seus alunos tornam-se capacitados para atuar no mercado de trabalho.

Visando a máxima qualidade, a K19 mantém as suas apostilas em constante renovação e melhoria, oferece instalações físicas apropriadas para o ensino e seus instrutores estão sempre atualizados didática e tecnicamente.



Seguro Treinamento

Na K19 o aluno faz o curso quantas vezes quiser!

Comprometida com o aprendizado e com a satisfação dos seus alunos, a K19 é a única que possui o Seguro Treinamento. Ao contratar um curso, o aluno poderá refazê-lo quantas vezes desejar mediante a disponibilidade de vagas e pagamento da franquia do Seguro Treinamento.

As vagas não preenchidas até um dia antes do início de uma turma da K19 serão destinadas ao alunos que desejam utilizar o Seguro Treinamento. O valor da franquia para utilizar o Seguro Treinamento é 10% do valor total do curso.



Termo de Uso

Termo de Uso

Todo o conteúdo desta apostila é propriedade da K19 Treinamentos. A apostila pode ser utilizada livremente para estudo pessoal . Além disso, este material didático pode ser utilizado como material de apoio em cursos de ensino superior desde que a instituição correspondente seja reconhecida pelo MEC (Ministério da Educação) e que a K19 seja citada explicitamente como proprietária do material.

É proibida qualquer utilização desse material que não se enquadre nas condições acima sem o prévio consentimento formal, por escrito, da K19 Treinamentos. O uso indevido está sujeito às medidas legais cabíveis.



Conheça os nossos cursos

-  K01 - Lógica de Programação
-  K02 - Desenvolvimento Web com HTML, CSS e JavaScript
-  K03 - SQL e Modelo Relacional
-  K11 - Orientação a Objetos em Java
-  K12 - Desenvolvimento Web com JSF2 e JPA2
-  K21 - Persistência com JPA2 e Hibernate
-  K22 - Desenvolvimento Web Avançado com JFS2, EJB3.1 e CDI
-  K23 - Integração de Sistemas com Webservices, JMS e EJB
-  K41 - Desenvolvimento Mobile com Android
-  K51 - Design Patterns em Java
-  K52 - Desenvolvimento Web com Struts
-  K31 - C# e Orientação a Objetos
-  K32 - Desenvolvimento Web com ASP.NET MVC

www.k19.com.br/cursos



Middleware Orientado a Mensagens - MOM

Geralmente, em ambientes corporativos, existem diversos sistemas para implementar as inúmeras regras de negócio da empresa. É comum dividir esses sistemas por departamentos ou por regiões geográficas.

Muito provavelmente, em algum momento, os diversos sistemas de uma empresa devem trocar informações ou requisitar procedimentos entre si. Essa integração pode ser realizada através de intervenção humana. Contudo, quando o volume de comunicação entre os sistemas é muito grande, essa abordagem se torna inviável.

Daí surge a necessidade de automatizar a integração entre sistemas. A abordagem mais simples para implementar essa automatização é utilizar arquivos de texto contendo os dados que devem ser transmitidos de um sistema para outro. Normalmente, um sistema compartilhado de arquivos é utilizado para essa transmissão. Essa estratégia possui certas limitações principalmente em relação à integridade das informações armazenadas nos arquivos.

Uma abordagem mais robusta para implementar essa integração é utilizar um Middleware Orientado a Mensagens (MOM). Um Middleware Orientado a Mensagens permite que um sistema receba ou envie mensagens para outros sistemas de forma **assíncrona**. Além disso, o sistema que envia uma mensagem não precisa conhecer os sistemas que a receberão. Da mesma forma, que os sistemas que recebem uma mensagem não precisam conhecer o sistema que a enviou. Essas características permitem que os sistemas sejam integrados com baixo acoplamento.

A plataforma Java define o funcionamento de um Middleware Orientado a Mensagens através da especificação **Java Message Service - JMS**. Todo servidor de aplicação que segue a especificação **Java EE** deve oferecer uma implementação do MOM definido pela JMS. Especificações importantes da plataforma Java como Enterprise Java Beans (EJB), Java Transaction API (JTA) e Java Transaction Service (JTS) possuem um relacionamento forte com a especificação JMS. A seguir veremos a arquitetura do MOM definido pela JMS.



Destinos: Filas e Tópicos

Na arquitetura JMS, os sistemas que devem ser integrados podem enviar mensagens para **filas(queues)** ou **tópicos(topics)** cadastrados anteriormente no MOM. Na terminologia JMS, as filas e os tópicos são chamados de **destinos(destinations)**.

Uma mensagem enviada para uma fila pode ser recebida por apenas **um** sistema (point-to-point). Uma mensagem enviada para um tópico pode ser recebida por **diversos** sistemas (publish-and-subscribe).

As filas e os tópicos são objetos que podem ser criados pelos administradores do MOM. Os provedores JMS podem oferecer ferramentas distintas para a criação desses objetos.

No Glassfish, as filas e os tópicos podem ser criados através da interface web de administração do servidor. No Wildfly, esses objetos podem ser criados através de código XML.



Exercícios de Fixação

- 1 Copie o arquivo **wildfly-8.2.0.Final.zip** da pasta **K19-Arquivos** para a sua Área de Trabalho. Depois, descompacte esse arquivo.



Importante

Você também pode obter o arquivo **wildfly-8.2.0.Final.zip** através do site da K19: www.k19.com.br/arquivos.

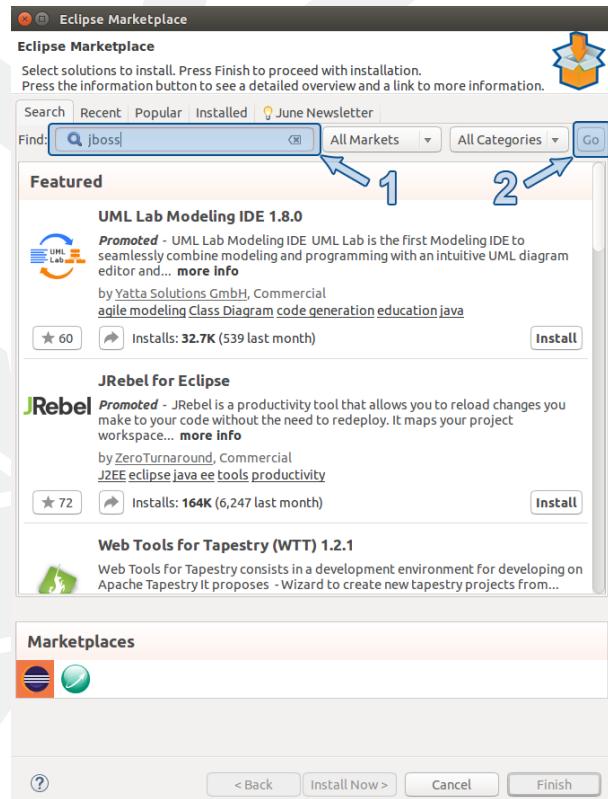
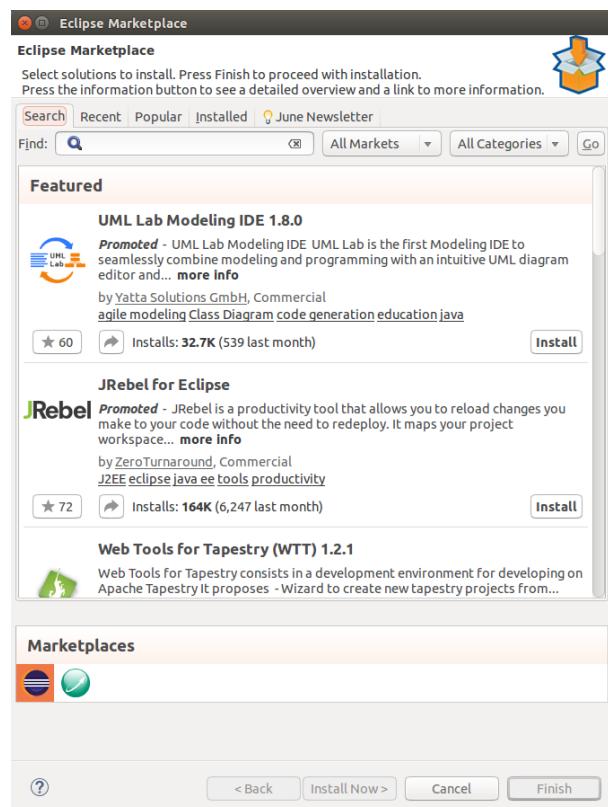
- 2 Copie o arquivo **glassfish-4.1.zip** da pasta **K19-Arquivos** para a sua Área de Trabalho. Depois, descompacte esse arquivo.

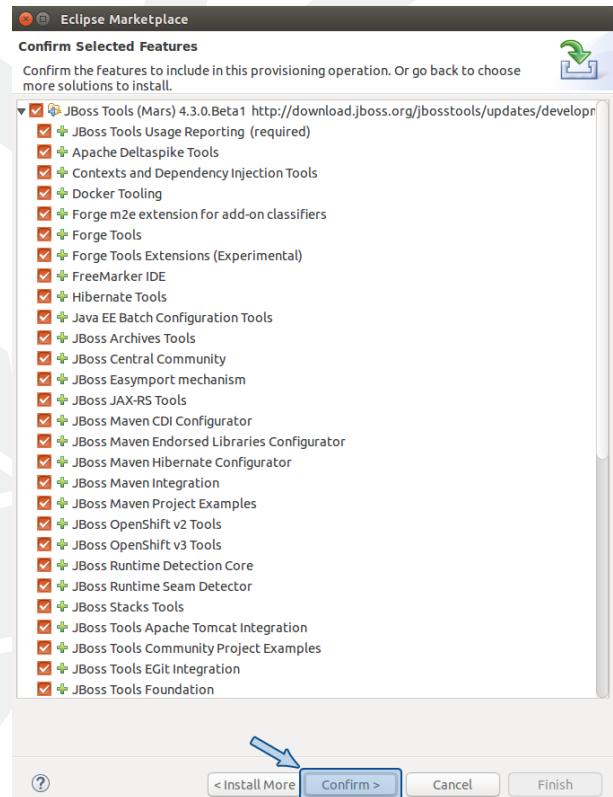
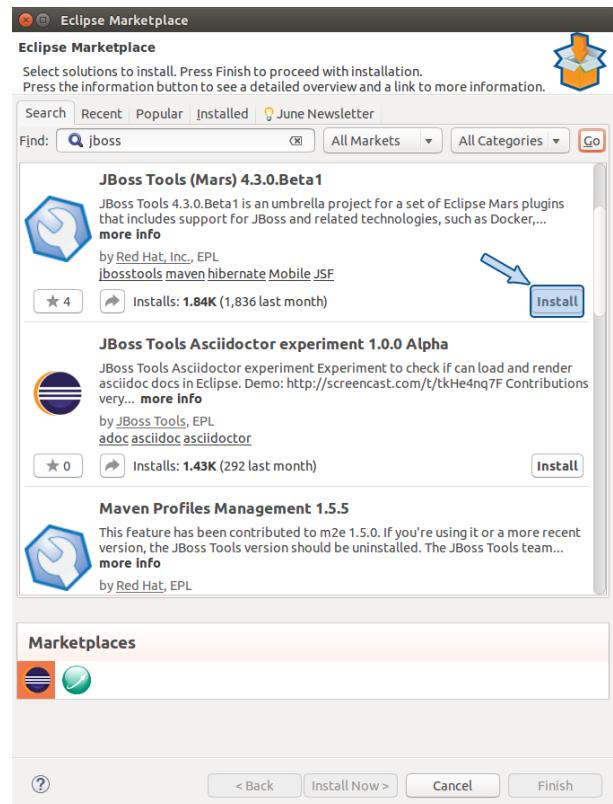


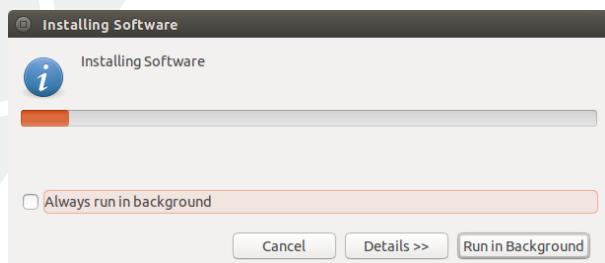
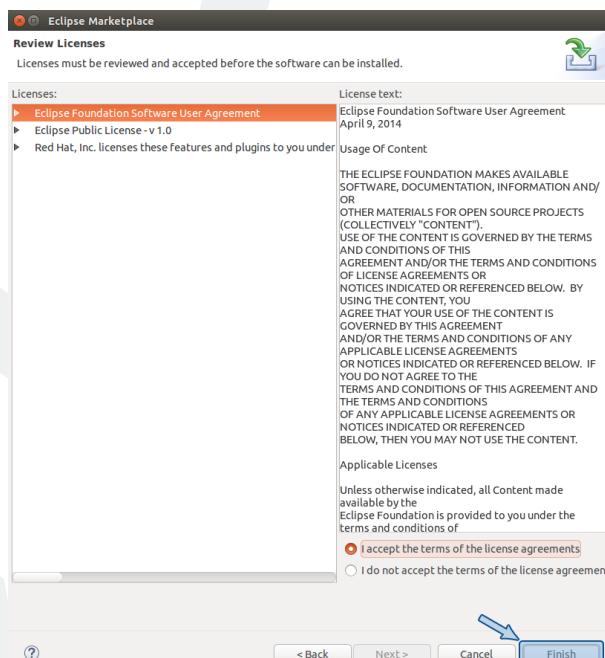
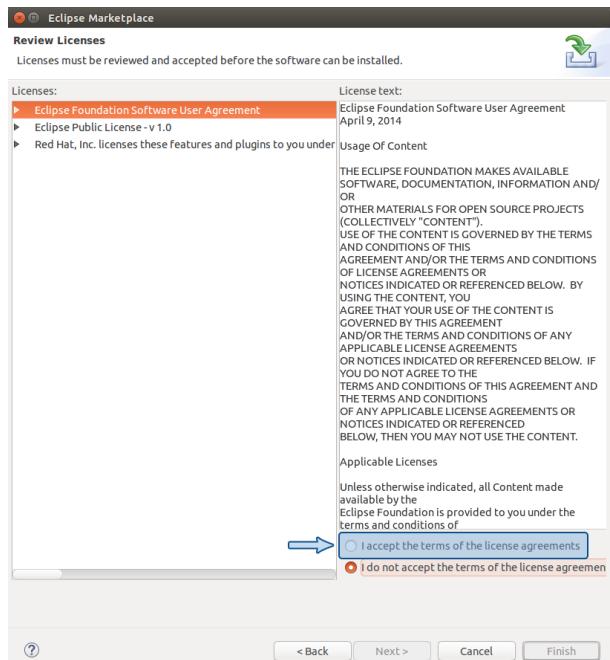
Importante

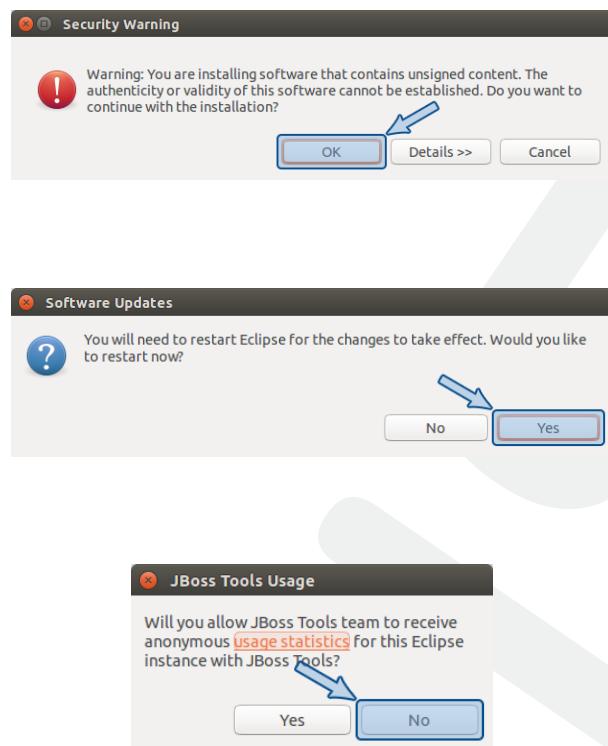
Você também pode obter o arquivo **glassfish-4.1.zip** através do site da K19: www.k19.com.br/arquivos.

- 3 Utilizando o Eclipse Marketplace, adicione no Eclipse Mars o suporte ao Wildfly. Digite “CTRL + 3” para abrir o Quick Access. Em seguida, pesquise por “Eclipse Marketplace”.

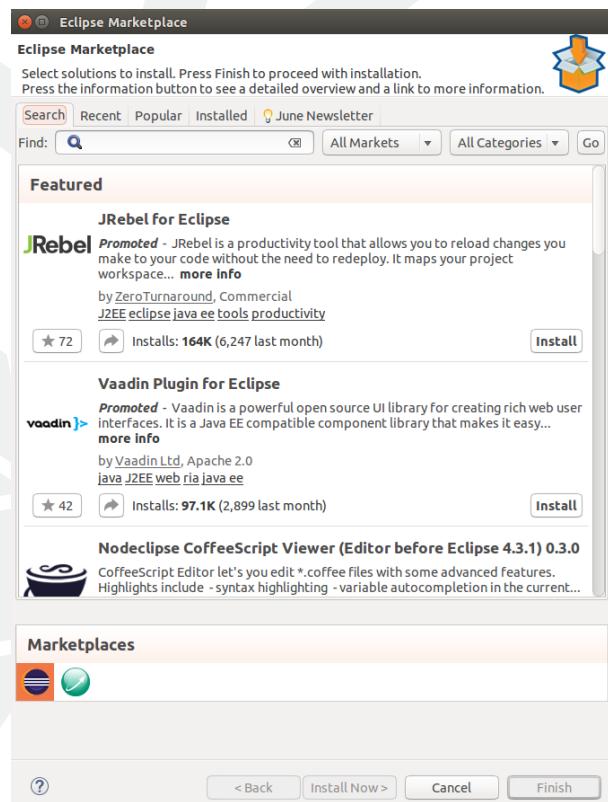


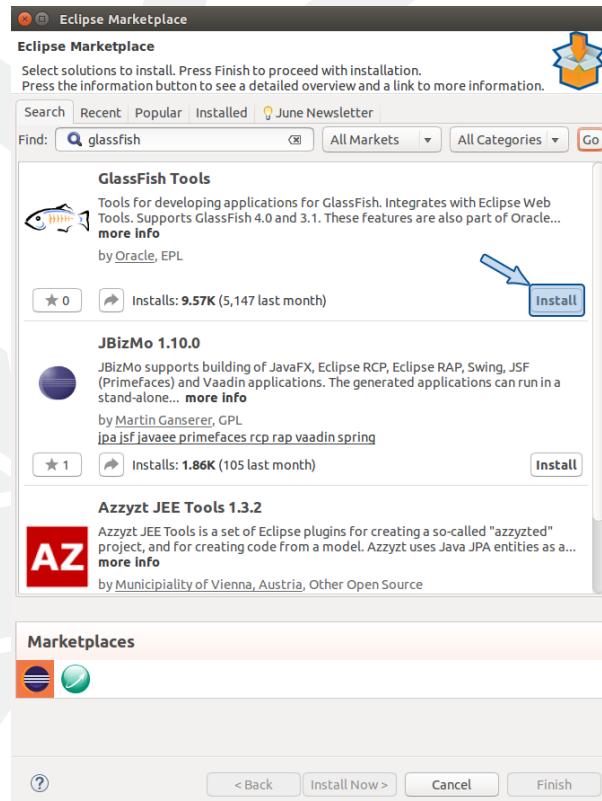
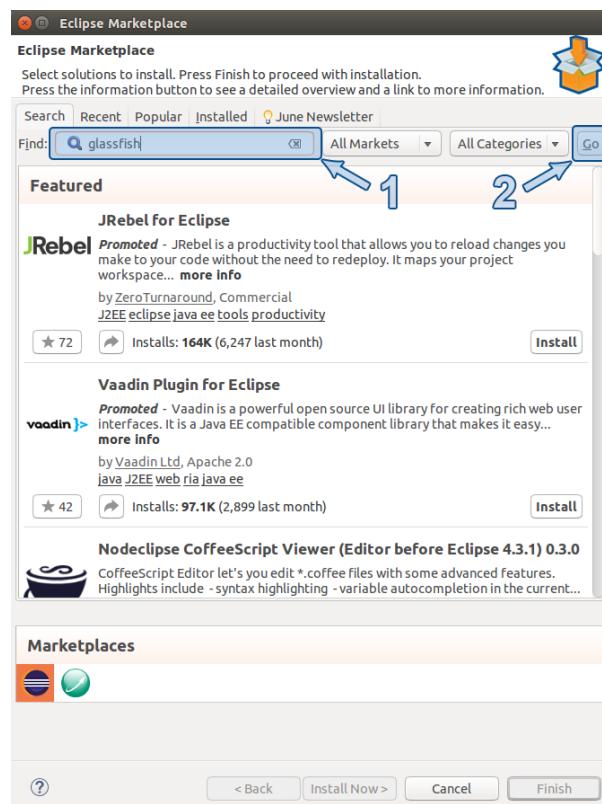


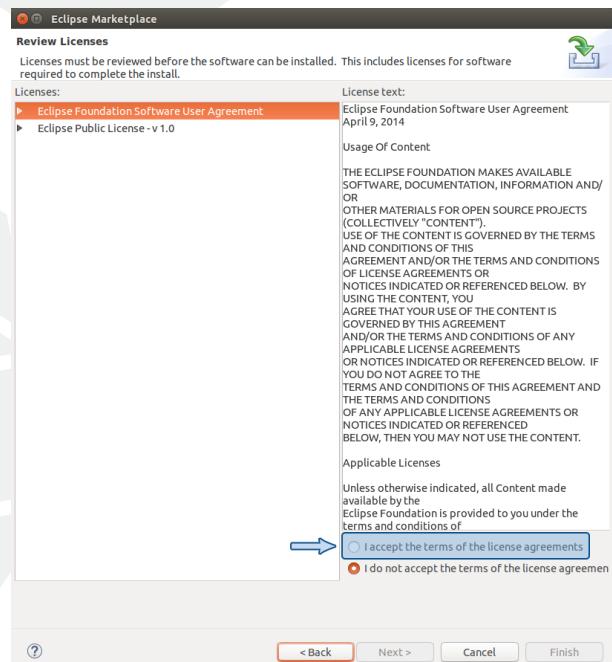
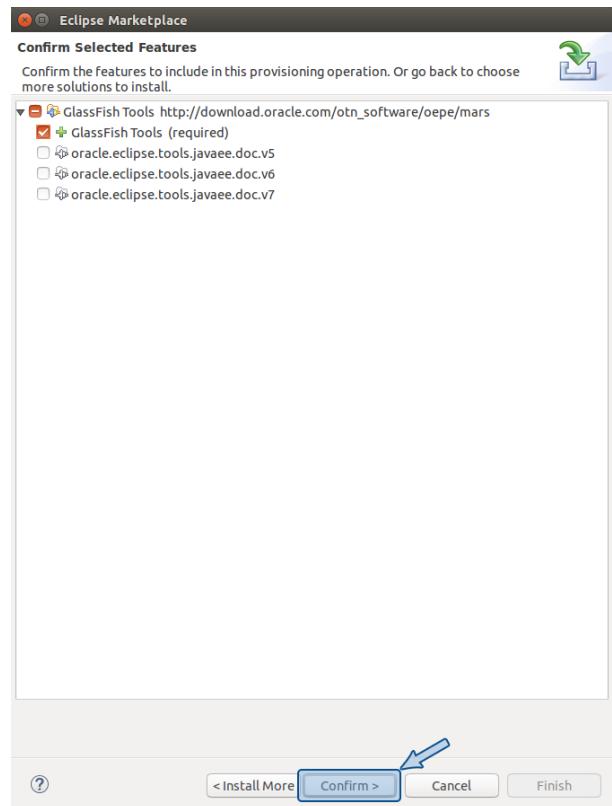


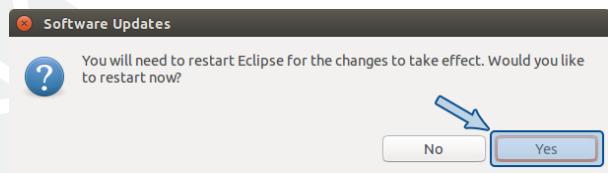
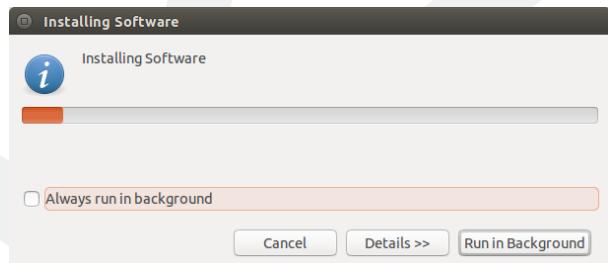
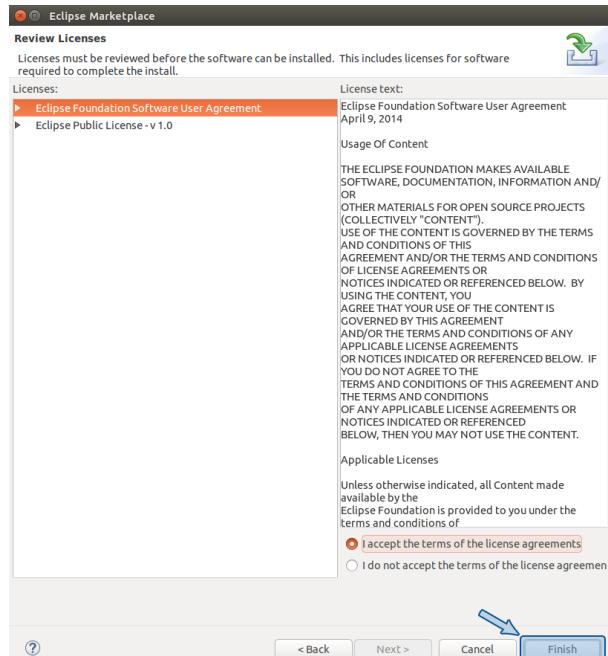


- 4 Utilizando o Eclipse Marketplace, adicione no Eclipse Mars o suporte ao Glassfish. Digite “CTRL + 3” para abrir o Quick Access. Em seguida, pesquise por “Eclipse Marketplace”.



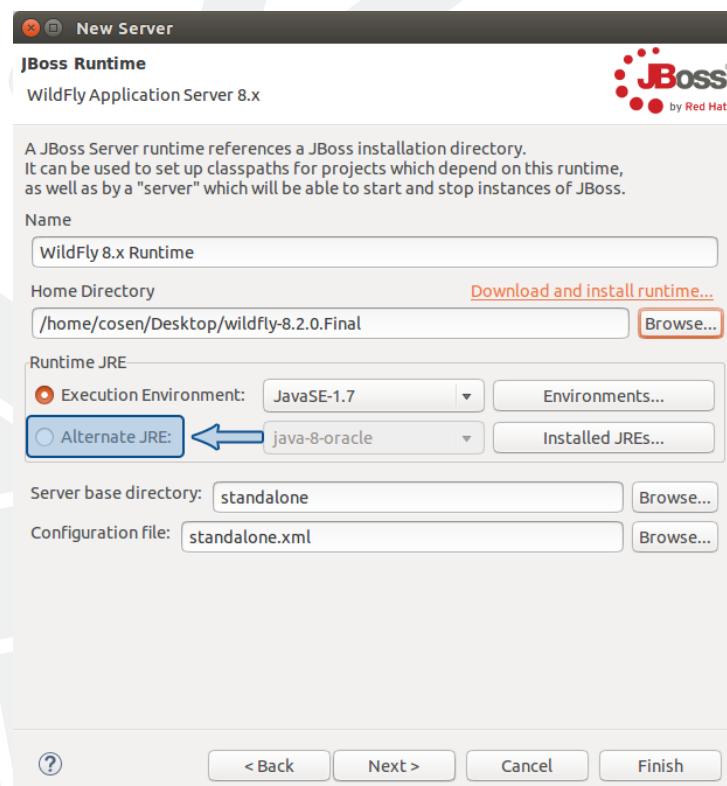
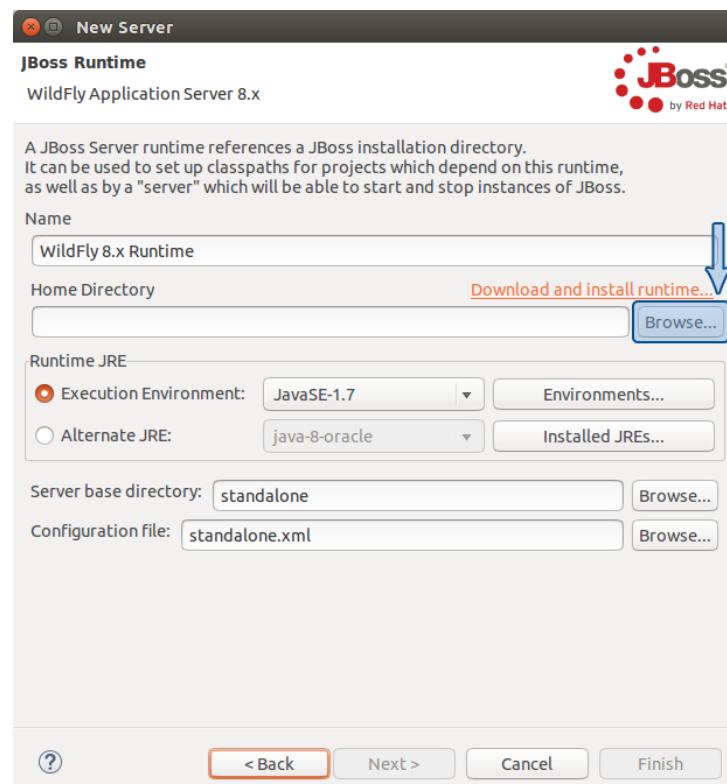


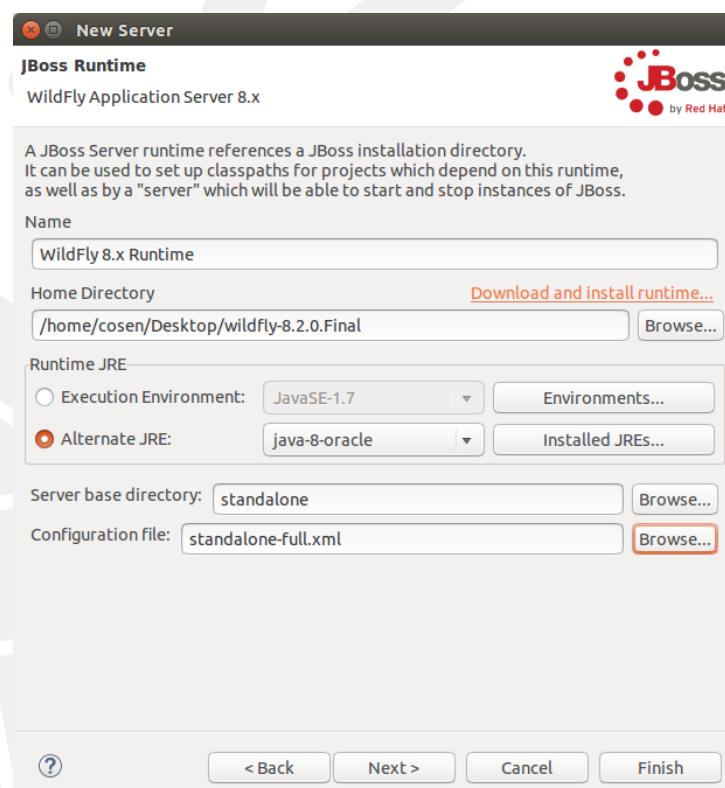
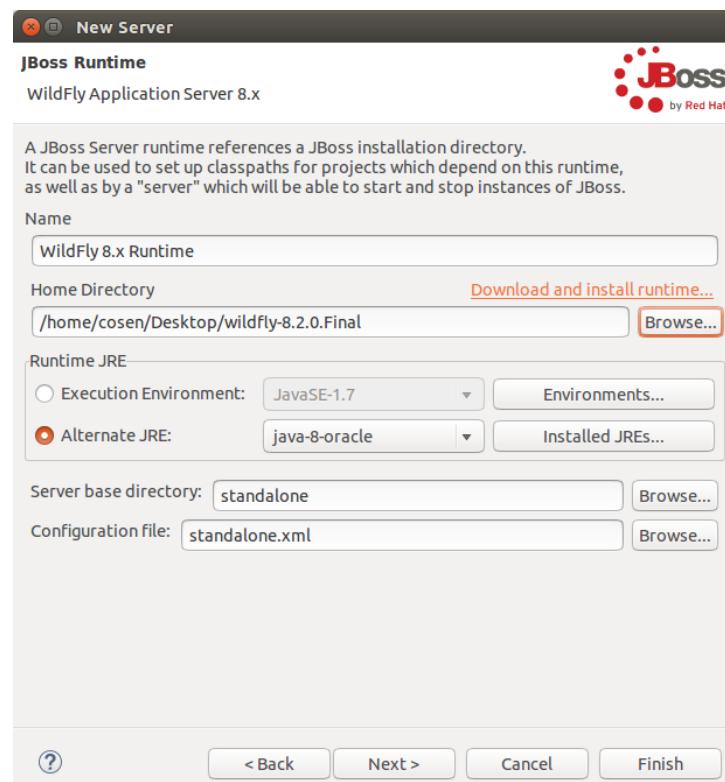


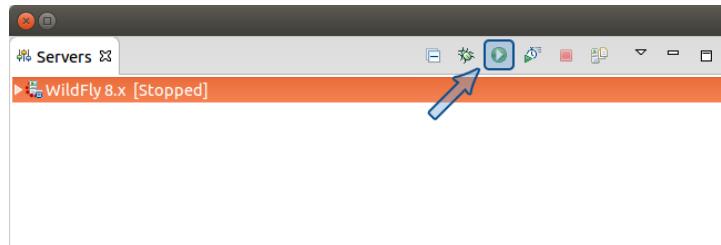


- 5 Configure o Wildfly no Eclipse Mars. Digite “CTRL + 3” para abrir o Quick Access. Em seguida, pesquise por “Define a new server”.









```

WildFly 8.x [JBoss Application Server Startup Configuration] /usr/lib/jvm/java-8-oracle/bin/java [Jul 1, 2015, 6:48:41 PM]
18:48:43,931 INFO  [org.jboss.as.connector.subsystems.datasources] (ServerService Thread Pool -- 27) JBAS010403: Deploying JDBC-compliant driver class org.postgresql.Driver (version 9.1)
18:48:43,946 INFO  [org.jboss.as.jmx] (ServerService Thread Pool -- 30) JBAS011302: Started the following JMX Implementation
18:48:43,952 INFO  [org.jboss.as.connector.deployers.jdbc] (MSC service thread 1-5) JBAS010417: Started Driver service with driver-name=postgresql
18:48:43,962 INFO  [org.jboss.as.naming] (MSC service thread 1-5) JBAS011802: Starting Naming Service
18:48:43,974 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-14) JBAS017502: Undertow 1.1.0.Final starting
18:48:43,965 INFO  [org.wildfly.extension.undertow] (ServerService Thread Pool -- 47) JBAS017502: Undertow 1.1.0.Final startin
18:48:43,977 INFO  [org.jboss.as.mail.extension] (MSC service thread 1-12) JBAS015400: Bound mail session [java:jboss/mail/Def
18:48:44,006 INFO  [org.jboss.remoting] (MSC service thread 1-8) JBoss Remoting version 4.8.6.Final
18:48:44,012 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-1) JBAS017527: Creating file handler for pa
18:48:44,351 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-4) JBAS017521: Starting server default-server.
18:48:44,386 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-4) JBAS017531: Host default-host starting
18:48:44,481 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-14) JBAS017519: Undertow HTTP listener default liste
18:48:44,568 INFO  [org.jboss.as.server.deployment.scanner] (MSC service thread 1-9) JBAS015012: Started FilesystemDeployments
18:48:44,617 INFO  [org.jboss.as.connector.subsystems.datasources] (MSC service thread 1-6) JBAS010400: Bound data source [jav
18:48:45,276 INFO  [org.jboss.ws.common.management] (MSC service thread 1-11) JBAS022852: Starting JBoss Web Services - Stack
18:48:45,302 INFO  [org.jboss.as] (Controller Boot Thread) JBAS010501: Admin console listening on http://127.0.0.1:9999
18:48:45,502 INFO  [org.jboss.as] (Controller Boot Thread) JBAS010501: Admin console listening on http://127.0.0.1:9999
18:48:45,503 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015074: WildFly 8.0.0.Final "Tweek" started in 3079ms - Started

```



Welcome to WildFly 8

Your WildFly 8 is running.

[Documentation](#) | [Quickstarts](#) | [Administration Console](#)

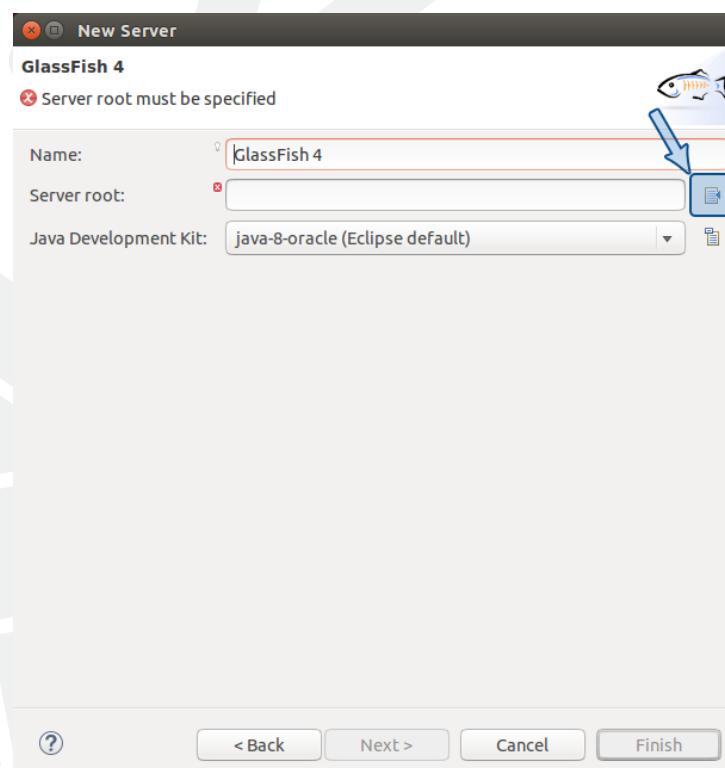
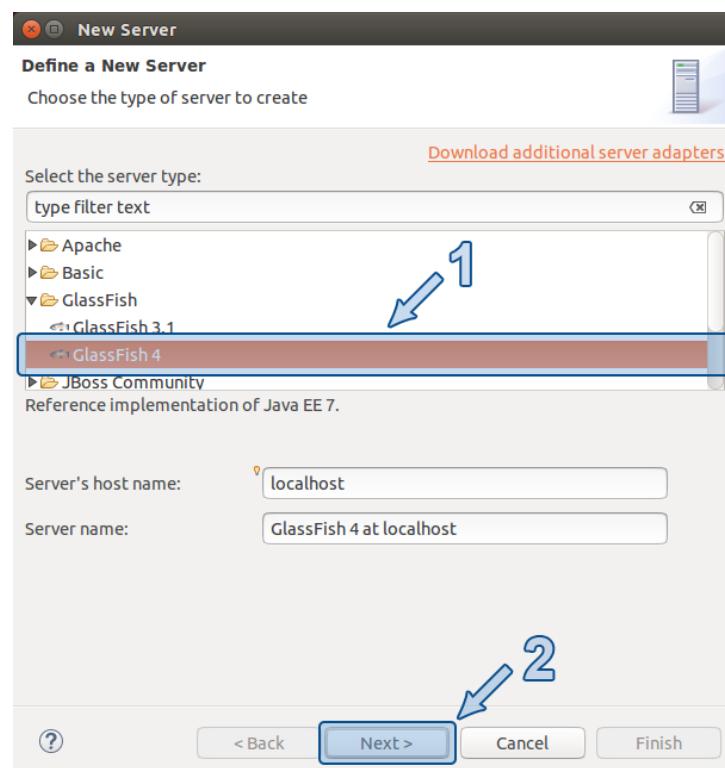
[WildFly Project](#) | [User Forum](#) | [Report an issue](#)

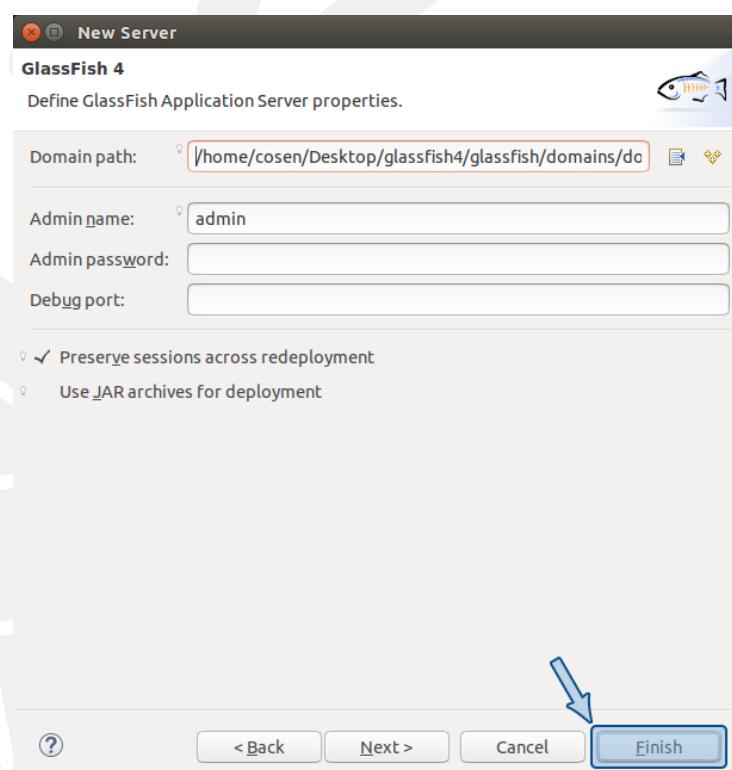
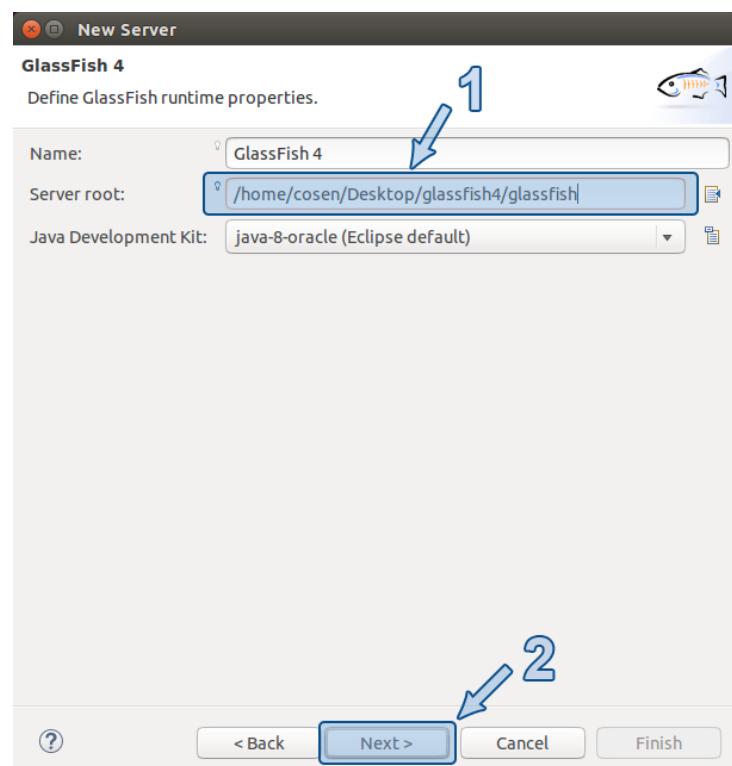
 | [JBoss Community](#)

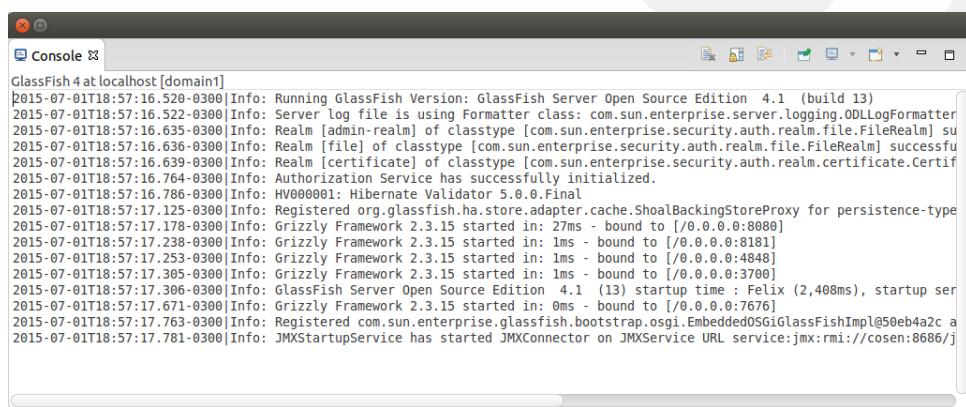
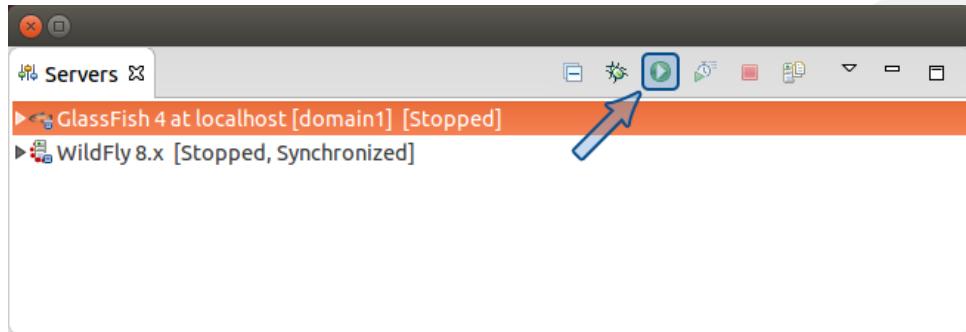
To replace this page simply deploy your own war with / as its context path.

To disable it, remove "welcome-content" header for location / in undertow subsystem.

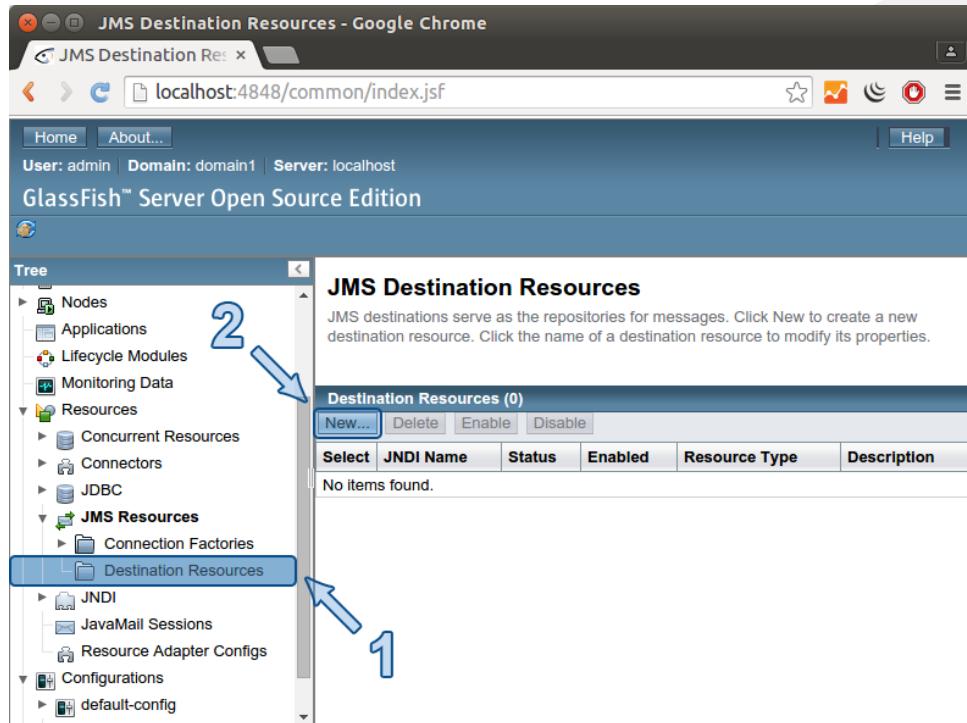
- 6 Pare o Wildfly e configure o Glassfish no Eclipse Mars. Digite “CTRL + 3” para abrir o Quick Access. Em seguida, pesquise por “Define a new server”.







- 7 Crie uma fila JMS através da interface de administração do **Glassfish** seguindo exatamente os passos das imagens abaixo:



- 8 Crie um tópico JMS através da interface de administração do Glassfish seguindo exatamente os passos das imagens abaixo:

- 9 Pare o Glassfish. Crie uma fila e um tópico JMS no Wildfly alterando o arquivo de configuração **standalone-full.xml**. Você encontra esse arquivo na pasta **wildfly-8.2.0.Final/standalone/configuration**.

```

1 ...
2 <jms-destinations>

```

```

3 <jms-queue name="pedidos">
4   <entry name="queue/pedidos"/>
5   <entry name="java:jboss/exported/jms/queue/pedidos"/>
6 </jms-queue>
7 <jms-topic name="noticias">
8   <entry name="topic/noticias"/>
9   <entry name="java:jboss/exported/jms/topic/noticias"/>
10 </jms-topic>
11 . .
12 </jms-destinations>
13 . .

```

Código XML 1.1: standalone-full.xml

- 10 Inicie o **Wildfly** e verifique no Console se as mensagens abaixo foram exibidas.

```

trying to deploy queue jms.topic.noticias
Bound messaging object to jndi name topic/noticias
Bound messaging object to jndi name java:jboss/exported/jms/topic/noticias

trying to deploy queue jms.queue.pedidos
Bound messaging object to jndi name java:jboss/exported/jms/queue/pedidos
Bound messaging object to jndi name queue/pedidos

```



Fábricas de Conexões

Os sistemas que desejam trocar mensagens através de filas ou tópicos devem obter conexões JMS através das fábricas cadastradas no MOM. A partir da versão 2.0 da especificação JMS, os provedores JMS em ambientes Java EE devem oferecer uma fábrica de conexão padrão. Essa fábrica pode ser obtida através do serviço de nomes JNDI executando um lookup usando o nome **java:comp:DefaultJMSSConnectionFactory**.



Visão Geral

Inicialmente, sem se preocupar com detalhes, vamos mostrar os passos necessários para enviar ou receber mensagens através da arquitetura JMS.

Fábricas de conexão, filas e tópicos

As fábricas de conexão, as filas e os tópicos são objetos administrados pelos provedores JMS. Quando uma aplicação deseja utilizar esses objetos, ela deve obtê-los através de pesquisas ao serviço de nomes. O serviço de nomes é definido pela especificação JNDI.

```

1 InitialContext ic = new InitialContext();
2 ConnectionFactory factory;
3 factory = (ConnectionFactory)ic.lookup("NomeDaFactory");
4
5 Queue queue
6 queue = (Queue)ic.lookup("NomeDaQueue");
7

```

```
8  
9 Topic topic;  
10 topic = (Topic)ic.lookup("NomeDoTopic");
```

As aplicações Java EE podem obter mais facilmente as fábricas de conexão, as filas e os tópicos através de injeção de dependências.

```
1 @Resource(lookup = "jms/nome-da-factory")  
2 private ConnectionFactory connectionFactory;
```

```
1 @Resource(lookup = "jms/queue/nome-da-queue")  
2 private Queue queue;
```

```
1 @Resource(lookup = "jms/queue/nome-do-topic")  
2 private Topic topic;
```

Conexões e Sessões

Na API JMS clássica, as sessões JMS são responsáveis pela criação das mensagens JMS, dos produtores e dos consumidores de mensagens. As sessões JMS são criadas pelas conexões JMS que por sua vez são obtidas através de uma fábrica de conexões.

```
1 Connection connection = factory.createConnection();
```

```
1 Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

Contextos

Na API JMS simplificada, os contextos JMS são responsáveis pela criação dos produtores e dos consumidores de mensagens. Esses contextos são obtidos através de uma fábrica de conexões.

```
1 JMSContext context = factory.createContext();
```

As aplicações Java EE podem obter mais facilmente os contextos através de injeção de dependências.

```
1 @Inject  
2 private JMSContext context;
```

Produtores e Consumidores

Na API JMS clássica, através de uma sessão JMS, podemos criar produtores e consumidores de mensagens. Esses objetos são criados já ligados a um destino específico, ou seja, uma fila ou um tópico.

```
1 MessageProducer producer = session.createProducer(queue);
```

```
1 MessageConsumer consumer = session.createConsumer(queue);
```

Na API JMS simplificada, através de um contexto JMS, podemos criar produtores e consumidores de mensagens. Os consumidores são criados já ligados a um destino específico, ou seja, uma fila ou um tópico.

```
1 JMSProducer producer = context.createProducer();
```

```
1 JMSConsumer consumer = context.createConsumer(topic);
```

Mensagens

Na API JMS clássica, as mensagens são criadas pelas sessões JMS. O tipo mais simples de mensagem é o **TextMessage**.

```
1 TextMessage message = session.createTextMessage();
2 message.setText("Olá");
```

Na API JMS simplificada, não é necessário criar uma **TextMessage**, basta utilizar objetos do tipo **String**.

```
1 String message = "Olá";
```

Produzindo Mensagens

Na API JMS clássica, as mensagens JMS são enviadas através do método **send()** de um produtor (**MessageProducer**).

```
1 sender.send(message);
```

Na API JMS simplificada, as mensagens JMS são enviadas através do método **send()** de um produtor (**JMSProducer**).

```
1 sender.send(destination, message);
```

Consumindo Mensagens

Na API JMS clássica, para receber mensagens JMS, é necessário inicializar a conexão e depois utilizar o método **receive()** de um consumidor (**MessageConsumer**).

```
1 connection.start();
2 TextMessage message = (TextMessage) receiver.receive();
```

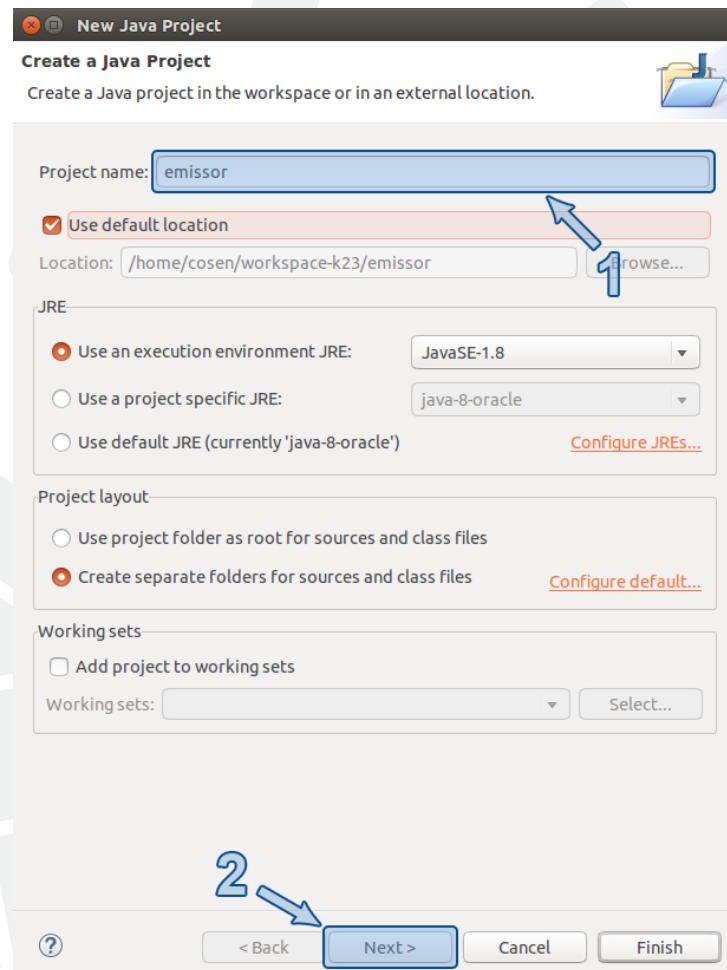
Na API JMS simplificada, para receber mensagens JMS, podemos utilizar o método **receiveBody()** de um consumidor (**JMSConsumer**).

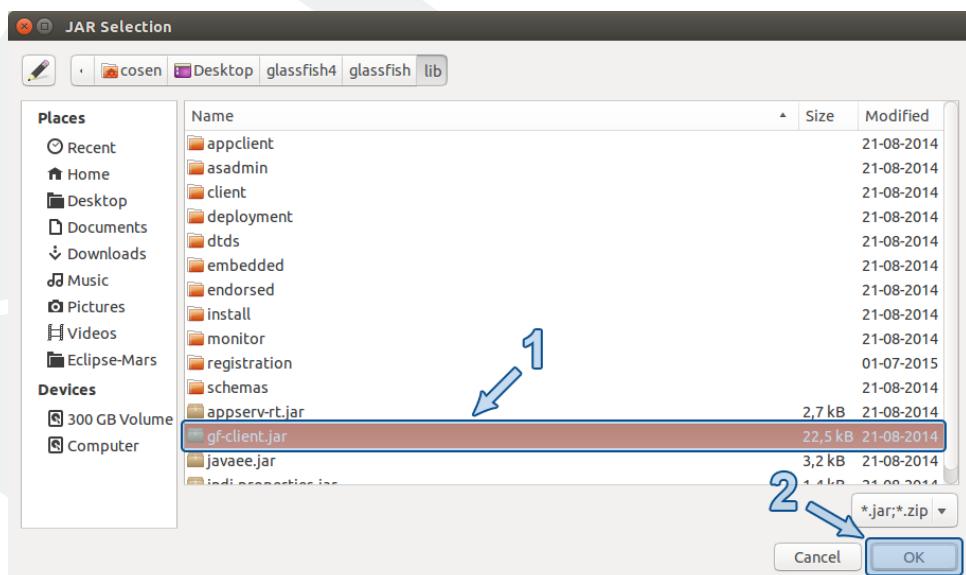
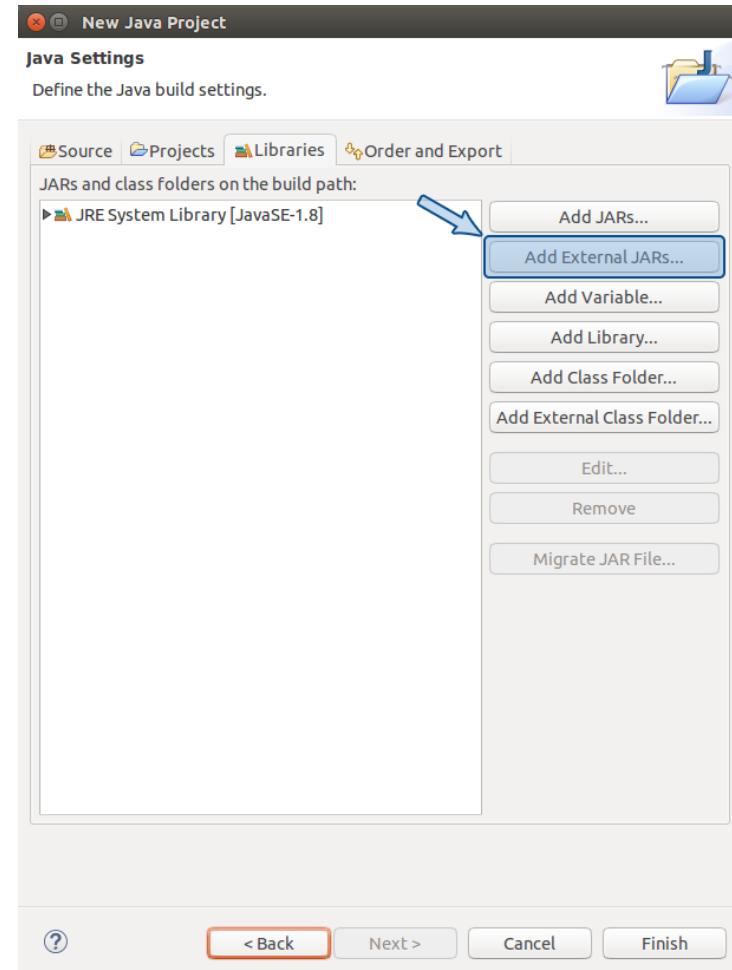
```
1 String mensagem = consumer.receiveBody(String.class);
```

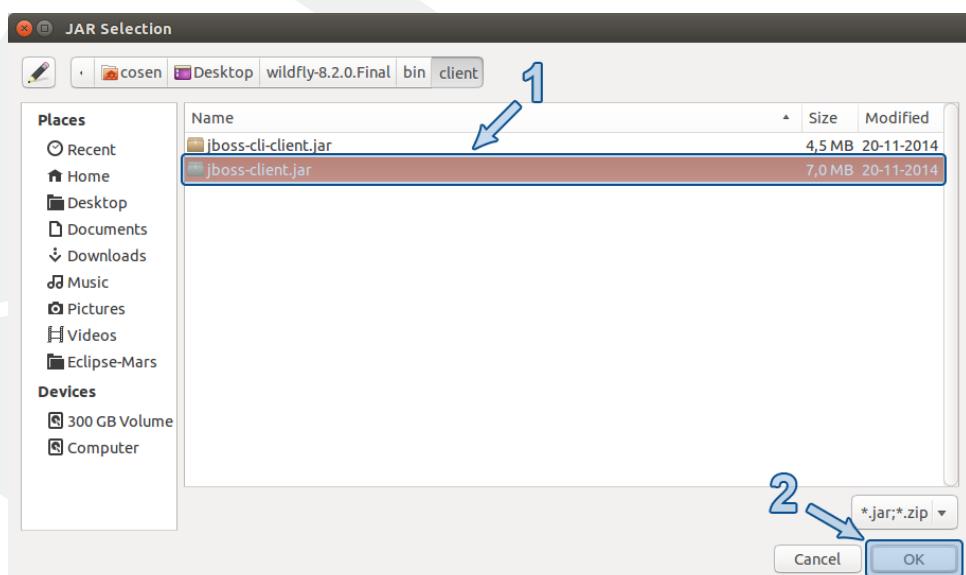
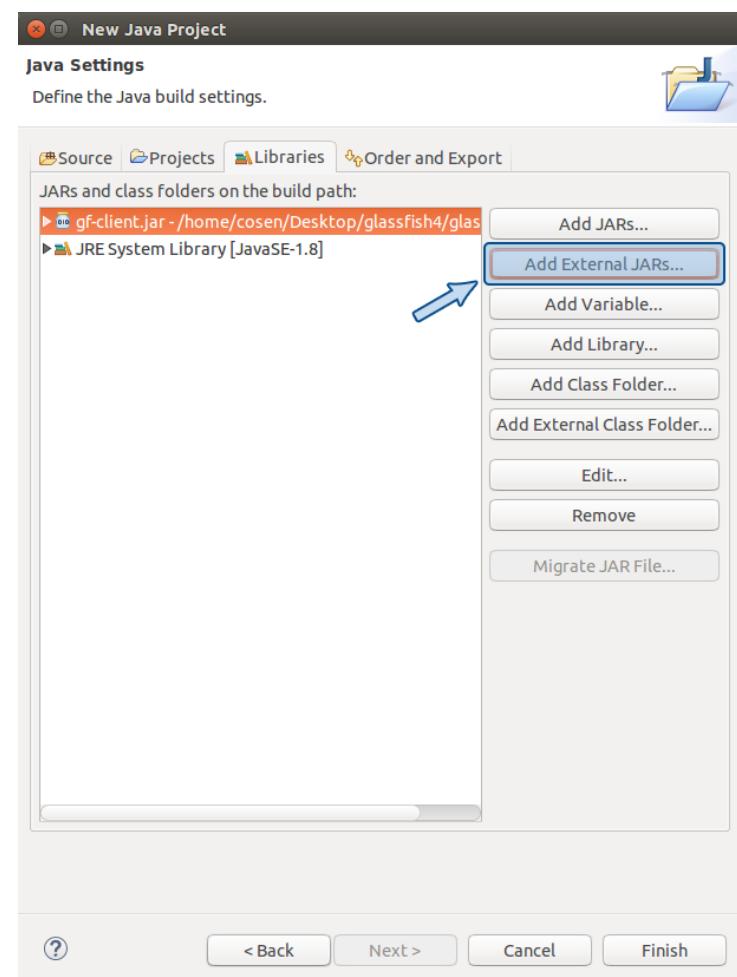


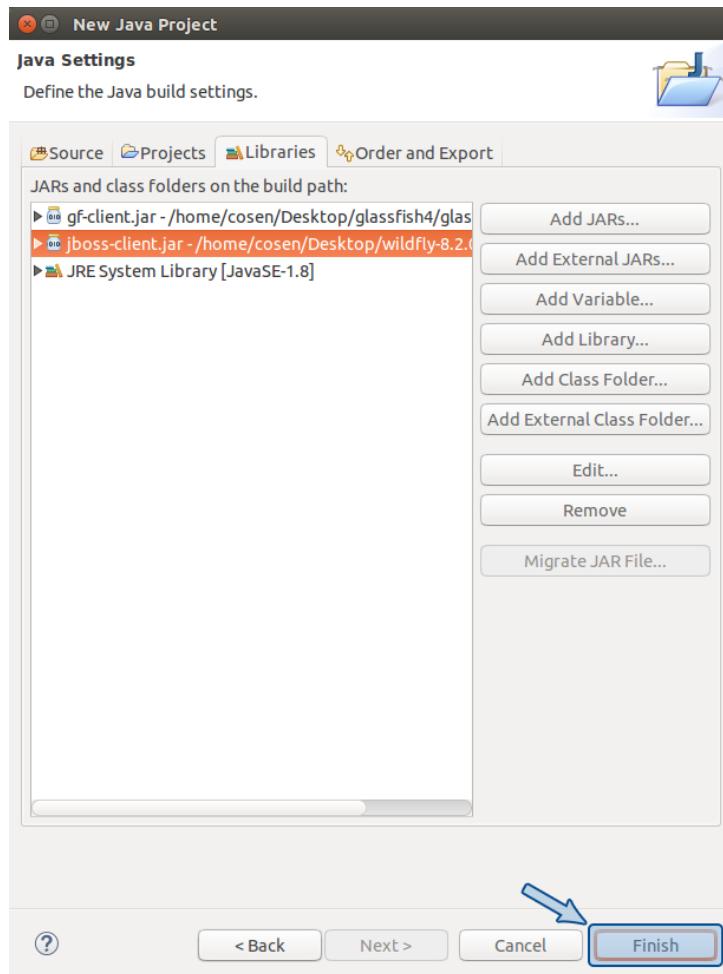
Exercícios de Fixação

- 11 Crie um Java Project no eclipse para implementar uma aplicação que possa enviar mensagens para a fila e o tópico criados anteriormente. Você pode digitar “CTRL + 3” em seguida “new Java Project” e “ENTER”. Depois, siga exatamente as imagens abaixo.









- 12 Crie um pacote chamado **br.com.k19.emissores** no projeto **emissor**.
- 13 Adicione no pacote **br.com.k19.emissores** uma classe com main para enviar uma mensagem JMS para a fila **pedidos** do Glassfish.

```

1 package br.com.k19.emissores;
2
3 import java.util.Properties;
4
5 import javax.jms.ConnectionFactory;
6 import javax.jms.JMSCContext;
7 import javax.jms.JMSProducer;
8 import javax.jms.Queue;
9 import javax.naming.InitialContext;
10
11 public class EnviaNovoPedidoGlassfish {
12     public static void main(String[] args) throws Exception {
13         // serviço de nomes - JNDI
14         Properties props = new Properties();
15
16         props.setProperty("java.naming.factory.initial",
17             "com.sun.enterprise.naming.SerialInitContextFactory");
18
19         props.setProperty("java.naming.factory.url.pkgs",
20             "com.sun.enterprise.naming");

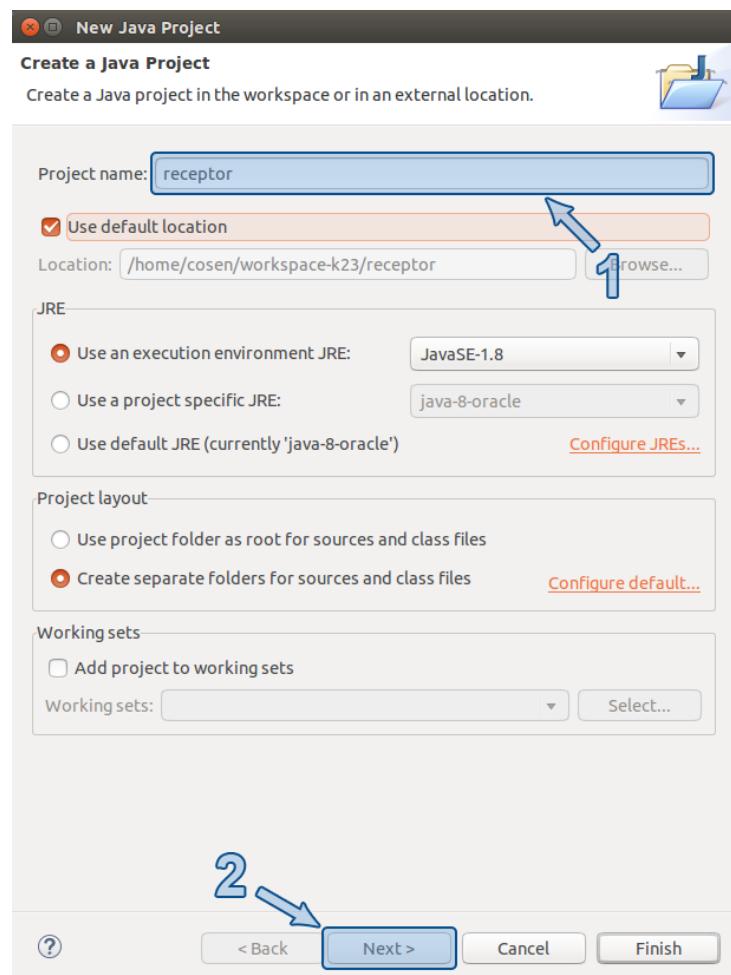
```

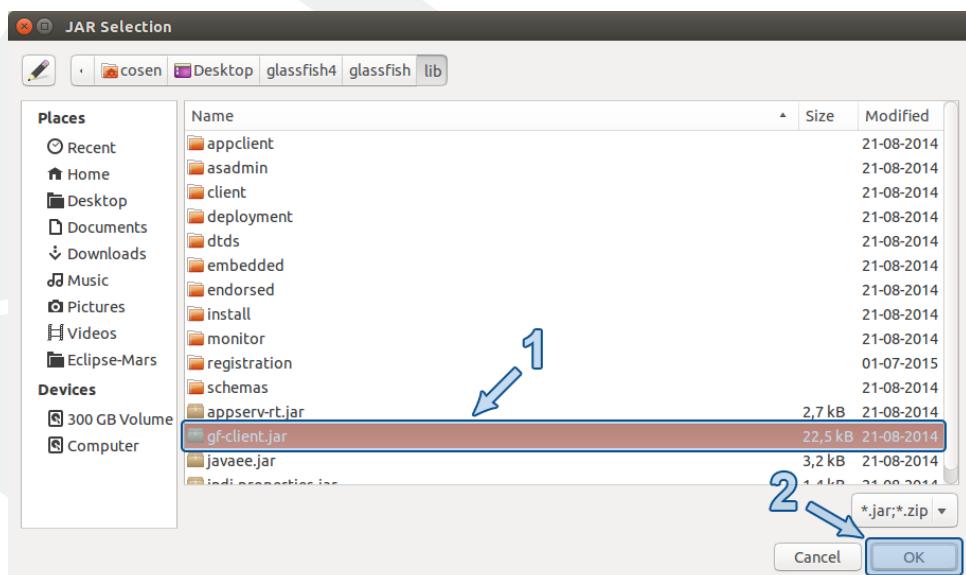
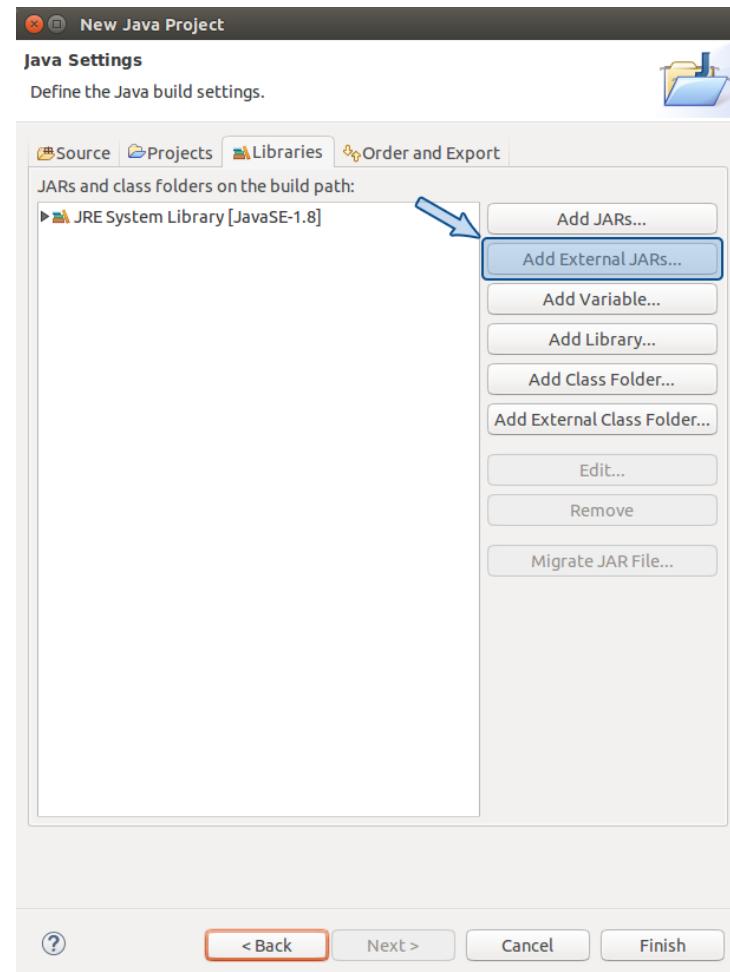
```
21 props.setProperty("java.naming.factory.state",
22     "com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl");
23
24 InitialContext ic = new InitialContext(props);
25
26 // fábrica de conexões JMS
27 ConnectionFactory factory =
28     (ConnectionFactory) ic.lookup("jms/_defaultConnectionFactory");
29
30 // fila
31 Queue queue = (Queue) ic.lookup("jms/pedidos");
32
33 // contexto JMS
34 JMSContext context = factory.createContext();
35
36 // produtor de mensagens
37 JMSProducer producer = context.createProducer();
38
39 // mensagem
40 String mensagem = "Pedido " + System.currentTimeMillis();
41
42 // enviando
43 producer.send(queue, mensagem);
44
45 System.out.println("MENSAGEM ENVIADA: " + mensagem);
46
47 // fechando
48 context.close();
49
50 }
51 }
```

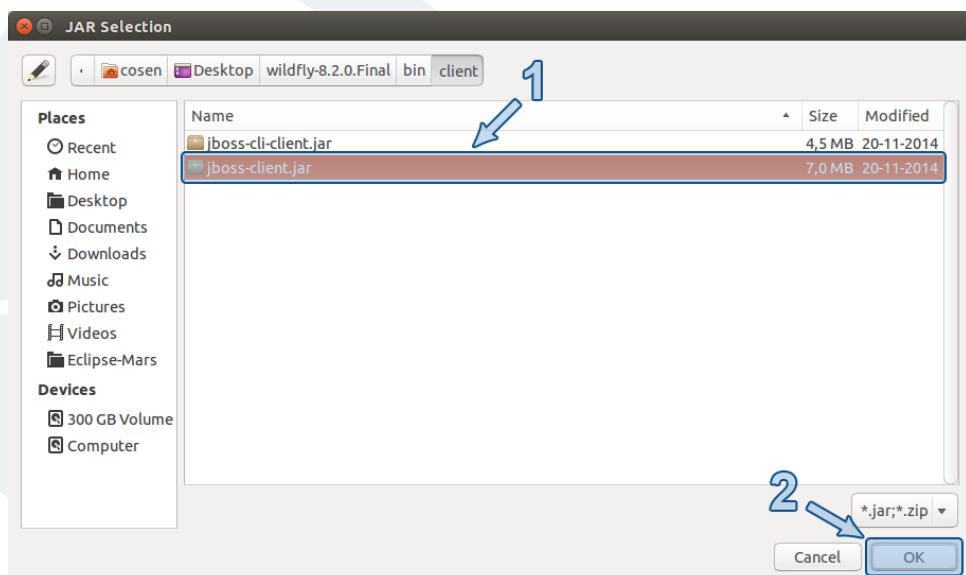
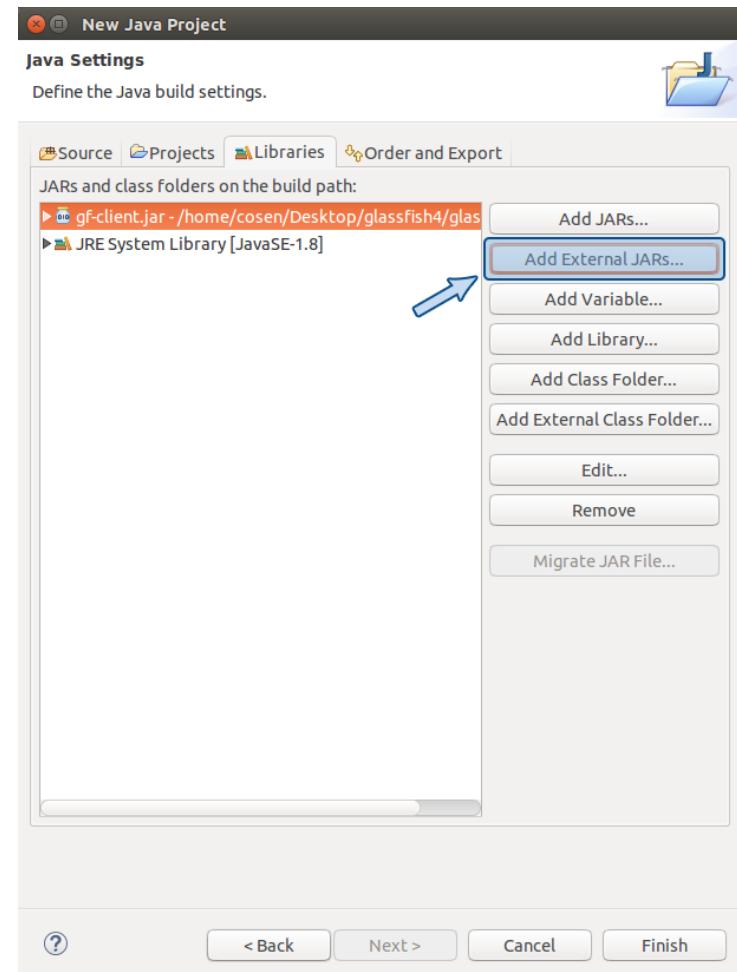
Código Java 1.19: EnviaNovoPedidoGlassfish.java

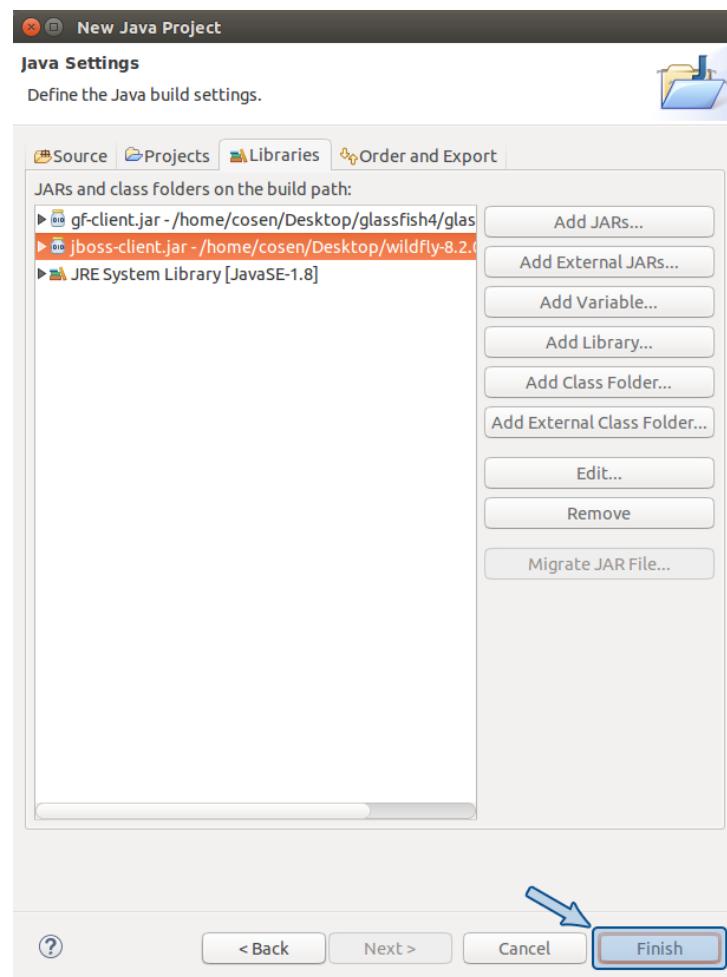
14 Pare o Wildfly e inicie o Glassfish. Em seguida, execute **uma** vez a classe **EnviaNovoPedidoGlassfish**.

15 Crie um Java Project no eclipse para implementar uma aplicação que possa receber as mensagens da fila e do tópico criados anteriormente. Você pode digitar “CTRL + 3” em seguida “new Java Project” e “ENTER”. Depois, siga exatamente as imagens abaixo.









- 16 Crie um pacote chamado **br.com.k19.receptores** no projeto **receptorJMS**.
- 17 Adicione no pacote **br.com.k19.receptores** uma classe com main para receber uma mensagem JMS da fila **pedidos** do Glassfish.

```
1 package br.com.k19.receptores;
2
3 import java.util.Properties;
4
5 import javax.jms.ConnectionFactory;
6 import javax.jms.JMSConsumer;
7 import javax.jms.JMSContext;
8 import javax.jms.Queue;
9 import javax.naming.InitialContext;
10
11 public class RecebePedidoGlassfish {
12     public static void main(String[] args) throws Exception {
13         // serviço de nomes - JNDI
14         Properties props = new Properties();
15
16         props.setProperty("java.naming.factory.initial",
17             "com.sun.enterprise.naming.SerialInitContextFactory");
18
19         props.setProperty("java.naming.factory.url.pkgs",
20             "com.sun.enterprise.naming");
```

```

21 props.setProperty("java.naming.factory.state",
22   "com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl");
23
24 InitialContext ic = new InitialContext(props);
25
26 // fábrica de conexões JMS
27 ConnectionFactory factory =
28   (ConnectionFactory) ic.lookup("jms/_defaultConnectionFactory");
29
30 // fila
31 Queue queue = (Queue) ic.lookup("jms/pedidos");
32
33 // contexto
34 JMSContext context = factory.createContext();
35
36 // consumidor de mensagens
37 JMSConsumer consumer = context.createConsumer(queue);
38
39 // consumindo a mensagem
40 String mensagem = consumer.receiveBody(String.class);
41
42 System.out.println("MENSAGEM RECEBIDA: " + mensagem);
43
44 // fechando
45 consumer.close();
46 context.close();
47
48 }
49 }
```

Código Java 1.20: RecebePedidoGlassfish.java

18 Execute **uma** vez a classe **RecebePedidoGlassfish**.

19 Adicione no pacote **br.com.k19.receptores** uma classe com main para receber uma mensagem JMS do tópico **notícias** do **Glassfish**.

```

1 package br.com.k19.receptores;
2
3 import java.util.Properties;
4
5 import javax.jms.ConnectionFactory;
6 import javax.jms.JMSConsumer;
7 import javax.jms.JMSContext;
8 import javax.jms.Topic;
9 import javax.naming.InitialContext;
10
11 public class AssinanteDeNoticiasGlassfish {
12   public static void main(String[] args) throws Exception {
13     // serviço de nomes - JNDI
14     Properties props = new Properties();
15
16     props.setProperty("java.naming.factory.initial",
17       "com.sun.enterprise.naming.SerialInitContextFactory");
18
19     props.setProperty("java.naming.factory.url.pkgs",
20       "com.sun.enterprise.naming");
21
22     props.setProperty("java.naming.factory.state",
23       "com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl");
24
25     InitialContext ic = new InitialContext(props);
26
27     // fábrica de conexões JMS
28     ConnectionFactory factory =
```

```

29     (ConnectionFactory) ic.lookup("jms/_defaultConnectionFactory");
30
31     // tópico
32     Topic topic = (Topic) ic.lookup("jms/noticias");
33
34     // contexto
35     JMSContext context = factory.createContext();
36
37     // consumidor de mensagens
38     JMSConsumer consumer = context.createConsumer(topic);
39
40     // consumindo a mensagem
41     String mensagem = consumer.receiveBody(String.class);
42
43     System.out.println("MENSAGEM RECEBIDA: " + mensagem);
44
45     // fechando
46     context.close();
47 }
48 }
```

Código Java 1.21: AssinanteDeNoticiasGlassfish.java

- 20** Execute **duas** vez a classe **AssinanteDeNoticiasGlassfish**. Observe que os assinantes aguardarão uma mensagem chegar.

- 21** Adicione no pacote **br.com.k19.emissores** uma classe com main para enviar uma mensagem JMS para o tópico **noticias** do Glassfish.

```

1 package br.com.k19.emissores;
2
3 import java.util.Properties;
4
5 import javax.jms.ConnectionFactory;
6 import javax.jms.JMSContext;
7 import javax.jms.JMSProducer;
8 import javax.jms.Topic;
9 import javax.naming.InitialContext;
10
11 public class EnviaNoticiaGlassfish {
12     public static void main(String[] args) throws Exception {
13         // serviço de nomes - JNDI
14         Properties props = new Properties();
15
16         props.setProperty("java.naming.factory.initial",
17             "com.sun.enterprise.naming.SerialInitContextFactory");
18
19         props.setProperty("java.naming.factory.url.pkgs",
20             "com.sun.enterprise.naming");
21
22         props.setProperty("java.naming.factory.state",
23             "com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl");
24
25         InitialContext ic = new InitialContext(props);
26
27         // fábrica de conexões JMS
28         ConnectionFactory factory = (ConnectionFactory) ic
29             .lookup("jms/_defaultConnectionFactory");
30
31         // tópico
32         Topic topic = (Topic) ic.lookup("jms/noticias");
33
34         // contexto
35         JMSContext context = factory.createContext();
```

```

36     // produtor de mensagens
37     JMSProducer producer = context.createProducer();
38
39     // mensagem
40     String mensagem = "Grazi Massafera foi ao shopping de chinelo";
41
42     // enviando
43     producer.send(topic, mensagem);
44
45     System.out.println("MENSAGEM ENVIADA: " + mensagem);
46
47     // fechando
48     context.close();
49 }
50 }
```

Código Java 1.22: EnviaNoticiaGlassfish.java

- 22 Execute **uma** vez a classe **EnviaNoticiaGlassfish** e observe os consoles dos assinantes.

- 23 Adicione um usuário no Wildfly com as seguintes características:

Tipo: Application User

Username: k19

Password: 1234

Roles: guest

Utilize o script **add-user.sh** para adicionar o usuário **k19** no Wildfly. Siga os passos abaixo.

```

co森en@co森en:~/Desktop/wildfly-8.2.0.Final/bin$ ./add-user.sh

What type of user do you wish to add?
  a) Management User (mgmt-users.properties)
  b) Application User (application-users.properties)
(a): b

Enter the details of the new user to add.
Using realm 'ApplicationRealm' as discovered from the existing property files.
Username : k19
Password recommendations are listed below. To modify these restrictions edit
the add-user.properties configuration file.
- The password should not be one of the following restricted values {root,
admin, administrator}
- The password should contain at least 8 characters, 1 alphabetic character(s),
1 digit(s), 1 non-alphanumeric symbol(s)
- The password should be different from the username
Password :
JBAS015269: Password must have at least 8 characters!
Are you sure you want to use the password entered yes/no? yes
Re-enter Password :
What groups do you want this user to belong to? (Please enter a comma separated
list, or leave blank for none)[ ]: guest
About to add user 'k19' for realm 'ApplicationRealm'
Is this correct yes/no? yes
Added user 'k19' to file '/home/co森en/Desktop/wildfly-8.2.0.Final/standalone/
configuration/application-users.properties'
Added user 'k19' to file '/home/co森en/Desktop/wildfly-8.2.0.Final/domain/
configuration/application-users.properties'
Added user 'k19' with groups guest to file '/home/co森en/Desktop/wildfly-8.2.0.Final/
standalone/configuration/application-roles.properties'
Added user 'k19' with groups guest to file '/home/co森en/Desktop/wildfly-8.2.0.Final/
domain/configuration/application-roles.properties'
```

```

Is this new user going to be used for one AS process to connect to another AS process?
e.g. for a slave host controller connecting to the master or for a Remoting connection
for server to server EJB calls.
yes/no? yes
To represent the user add the following to the server-identities definition <secret
value="MTIzNA==" />

```

Terminal 1.1: Criando um usuário no Wildfly

24 Adicione o arquivo **jboss-ejb-client.properties** na pasta **src** do projeto **emissor**.

```

1 endpoint.name=client-endpoint
2 remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
3
4 remote.connections=default
5
6 remote.connection.default.host=localhost
7 remote.connection.default.port = 8080
8 remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=<-->
9     false
10 remote.connection.default.username=k19
11 remote.connection.default.password=1234

```

Arquivo de Propriedades 1.1: jboss-ejb-client.properties

25 Adicione no pacote **br.com.k19.emissores** uma classe com main para enviar uma mensagem JMS para a fila **pedidos** do **Wildfly**.

```

1 package br.com.k19.emissores;
2
3 import java.util.Properties;
4
5 import javax.jms.ConnectionFactory;
6 import javax.jms.JMSCContext;
7 import javax.jms.JMSProducer;
8 import javax.jms.Queue;
9 import javax.naming.Context;
10 import javax.naming.InitialContext;
11
12 public class EnviaNovoPedidoWildfly {
13     public static void main(String[] args) throws Exception {
14         // serviço de nomes - JNDI
15         Properties props = new Properties();
16         props.put(Context.INITIAL_CONTEXT_FACTORY,
17             "org.jboss.naming.remote.client.InitialContextFactory");
18         props.put(Context.PROVIDER_URL, "http-remoting://localhost:8080");
19         props.put(Context.SECURITY_PRINCIPAL, "k19");
20         props.put(Context.SECURITY_CREDENTIALS, "1234");
21
22         InitialContext ic = new InitialContext(props);
23
24         // fábrica de conexões JMS
25         ConnectionFactory factory = (ConnectionFactory) ic
26             .lookup("jms/RemoteConnectionFactory");
27
28         // fila
29         Queue queue = (Queue) ic.lookup("jms/queue/pedidos");
30
31         // contexto JMS
32         JMSCContext context = factory.createContext("k19", "1234");
33
34         // produtor de mensagens
35         JMSProducer producer = context.createProducer();
36

```

```

37 // mensagem
38 String mensagem = "Pedido " + System.currentTimeMillis();
39
40 // enviando
41 producer.send(queue, mensagem);
42
43 System.out.println("MENSAGEM ENVIADA: " + mensagem);
44
45 // fechando
46 context.close();
47 }
48 }
```

Código Java 1.23: EnviaNovoPedidoWildfly.java

- 26** Pare o **Glasfish**. Inicie o **Wildfly**. Execute **uma** vez a classe **EnviaNovoPedidoWildfly**.
- 27** Adicione no pacote **br.com.k19.receptores** uma classe com main para receber uma mensagem JMS da fila **pedidos** do **Wildfly**.

```

1 package br.com.k19.receptores;
2
3 import java.util.Properties;
4
5 import javax.jms.ConnectionFactory;
6 import javax.jms.JMSConsumer;
7 import javax.jms.JMSContext;
8 import javax.jms.Queue;
9 import javax.naming.Context;
10 import javax.naming.InitialContext;
11
12 public class RecebePedidoWildfly {
13     public static void main(String[] args) throws Exception {
14         // serviço de nomes - JNDI
15         Properties props = new Properties();
16         props.put(Context.INITIAL_CONTEXT_FACTORY,
17                 "org.jboss.naming.remote.client.InitialContextFactory");
18         props.put(Context.PROVIDER_URL, "http-remoting://localhost:8080");
19         props.put(Context.SECURITY_PRINCIPAL, "K19");
20         props.put(Context.SECURITY_CREDENTIALS, "1234");
21
22         InitialContext ic = new InitialContext(props);
23
24         // fábrica de conexões JMS
25         ConnectionFactory factory = (ConnectionFactory) ic
26             .lookup("jms/RemoteConnectionFactory");
27
28         // fila
29         Queue queue = (Queue) ic.lookup("jms/queue/pedidos");
30
31         // contexto JMS
32         JMSContext context = factory.createContext("K19", "1234");
33
34         // consumidor de mensagens
35         JMSConsumer consumer = context.createConsumer(queue);
36
37         // consumindo a mensagem
38         String mensagem = consumer.receiveBody(String.class);
39
40         System.out.println("MENSAGEM RECEBIDA: " + mensagem);
41
42         // fechando
43         consumer.close();
44         context.close();
45     }
}
```

46 }

Código Java 1.24: RecebePedidoWildfly.java

- 28** Adicione o arquivo **jboss-ejb-client.properties** na pasta **src** do projeto **receptor**.

```

1 endpoint.name=client-endpoint
2 remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
3
4 remote.connections=default
5
6 remote.connection.default.host=localhost
7 remote.connection.default.port = 8080
8 remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=<-- 
    false
9
10 remote.connection.default.username=k19
11 remote.connection.default.password=1234

```

Arquivo de Propriedades 1.2: jboss-ejb-client.properties

- 29** Execute **uma** vez a classe **RecebePedidoWildfly**.

- 30** Adicione no pacote **br.com.k19.receptores** uma classe com main para receber uma mensagem JMS do tópico **notícias** do **Wildfly**.

```

1 package br.com.k19.receptores;
2
3 import java.util.Properties;
4
5 import javax.jms.ConnectionFactory;
6 import javax.jms.JMSConsumer;
7 import javax.jms.JMSText;
8 import javax.jms.Topic;
9 import javax.naming.Context;
10 import javax.naming.InitialContext;
11
12 public class AssinanteDeNoticiasWildfly {
13     public static void main(String[] args) throws Exception {
14         // serviço de nomes - JNDI
15         Properties props = new Properties();
16         props.put(Context.INITIAL_CONTEXT_FACTORY,
17             "org.jboss.naming.remote.client.InitialContextFactory");
18         props.put(Context.PROVIDER_URL, "http-remoting://localhost:8080");
19         props.put(Context.SECURITY_PRINCIPAL, "k19");
20         props.put(Context.SECURITY_CREDENTIALS, "1234");
21
22         InitialContext ic = new InitialContext(props);
23
24         // fábrica de conexões JMS
25         ConnectionFactory factory = (ConnectionFactory) ic
26             .lookup("jms/RemoteConnectionFactory");
27
28         // tópico
29         Topic topic = (Topic) ic.lookup("jms/topic/noticias");
30
31         // contexto JMS
32         JMSText context = factory.createContext("k19", "1234");
33
34         // consumidor de mensagens
35         JMSConsumer consumer = context.createConsumer(topic);
36

```

```

37 // consumindo a mensagem
38 String mensagem = consumer.receiveBody(String.class);
39
40 System.out.println("MENSAGEM RECEBIDA: " + mensagem);
41
42 // fechando
43 consumer.close();
44 context.close();
45 }
46 }
```

Código Java 1.25: AssinanteDeNoticiasWildfly.java

- 31 Execute **duas** vez a classe **AssinanteDeNoticiasWildfly**.
- 32 Adicione no pacote **br.com.k19.emissores** uma classe com main para enviar uma mensagem JMS para o tópico **notícias** do **Wildfly**.

```

1 package br.com.k19.emissores;
2
3 import java.util.Properties;
4
5 import javax.jms.ConnectionFactory;
6 import javax.jms.JMSCContext;
7 import javax.jms.JMSProducer;
8 import javax.jms.Topic;
9 import javax.naming.Context;
10 import javax.naming.InitialContext;
11
12 public class EnviaNoticiaWildfly {
13     public static void main(String[] args) throws Exception {
14         // serviço de nomes - JNDI
15         Properties props = new Properties();
16         props.put(Context.INITIAL_CONTEXT_FACTORY,
17                 "org.jboss.naming.remote.client.InitialContextFactory");
18         props.put(Context.PROVIDER_URL, "http-remoting://localhost:8080");
19         props.put(Context.SECURITY_PRINCIPAL, "k19");
20         props.put(Context.SECURITY_CREDENTIALS, "1234");
21
22         InitialContext ic = new InitialContext(props);
23
24         // fábrica de conexões JMS
25         ConnectionFactory factory = (ConnectionFactory) ic
26             .lookup("jms/RemoteConnectionFactory");
27
28         // tópico
29         Topic topic = (Topic) ic.lookup("jms/topic/noticias");
30
31         // contexto JMS
32         JMSCContext context = factory.createContext("k19", "1234");
33
34         // produtor de mensagens
35         JMSProducer producer = context.createProducer();
36
37         // mensagem
38         String mensagem = "Grazi Massafera foi ao shopping de chinelo";
39
40         // enviando
41         producer.send(topic, mensagem);
42
43         System.out.println("MENSAGEM ENVIADA: " + mensagem);
44
45         // fechando
46         context.close();
47     }
}
```

48 }

Código Java 1.26: EnviaNoticiaWildfly.java

- 33 Execute **uma** vez a classe **EnviaNoticiaWildfly** e observe os consoles dos assinantes.



Modos de recebimento

Na API JMS clássica, as mensagens JMS podem ser recebidas através de um **MessageConsumer** de três maneiras diferentes:

Bloqueante: A execução não continua até o recebimento da mensagem.

```
1 consumer.receive();
```

Semi-Bloqueante: A execução não continua até o recebimento da mensagem ou até o término de um período estipulado.

```
1 // espera no máximo 5 segundos
2 consumer.receive(5000);
```

Não-Bloqueante: A execução não é interrompida se a mensagem não for recebida imediatamente.

```
1 consumer.receiveNoWait();
```

Analogamente, na API JMS simplificada, as mensagens JMS podem ser recebidas através de um **JMSConsumer** de três maneiras diferentes:

Bloqueante: A execução não continua até o recebimento da mensagem.

```
1 consumer.receiveBody(String.class);
```

Semi-Bloqueante: A execução não continua até o recebimento da mensagem ou até o término de um período estipulado.

```
1 // espera no máximo 5 segundos
2 consumer.receiveBody(String.class, 5000);
```

Não-Bloqueante: A execução não é interrompida se a mensagem não for recebida imediatamente.

```
1 consumer.receiveBodyNoWait(String.class);
```



Percorrendo uma fila

Podemos percorrer as mensagens de uma fila sem retirá-las de lá. Para isso, devemos utilizar um **QueueBrowser**.

```
1 QueueBrowser queueBrowser = session.createBrowser(queue);
2
3 Enumeration<TextMessage> messages = queueBrowser.getEnumeration();
4 while (messages.hasMoreElements()) {
5     TextMessage message = messages.nextElement();
6 }
```

Um QueueBrowser pode ser obtido na API JMS simplificada através de um contexto JMS.

```
1 QueueBrowser queueBrowser = context.createBrowser(queue);
```



Exercícios de Fixação

- 34 Adicione no pacote **br.com.k19.receptores** uma classe com main para percorrer as mensagens da fila **pedidos** do Glassfish.

```
1 package br.com.k19.receptores;
2
3 import java.util.Enumeration;
4 import java.util.Properties;
5
6 import javax.jms.ConnectionFactory;
7 import javax.jms.JMSContext;
8 import javax.jms.Queue;
9 import javax.jms.QueueBrowser;
10 import javax.jms.TextMessage;
11 import javax.naming.InitialContext;
12
13 public class PercorrendoFilaGlassfish {
14     public static void main(String[] args) throws Exception {
15         // serviço de nomes - JNDI
16         Properties props = new Properties();
17
18         props.setProperty("java.naming.factory.initial",
19             "com.sun.enterprise.naming.SerialInitContextFactory");
20
21         props.setProperty("java.naming.factory.url.pkgs",
22             "com.sun.enterprise.naming");
23
24         props.setProperty("java.naming.factory.state",
25             "com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl");
26
27         InitialContext ic = new InitialContext(props);
28
29         // fábrica de conexões JMS
30         ConnectionFactory factory =
31             (ConnectionFactory) ic.lookup("jms/_defaultConnectionFactory");
32
33         // fila
34         Queue queue = (Queue) ic.lookup("jms/pedidos");
35
36         // contexto
37         JMSContext context = factory.createContext();
38
39         // queue browser
40         QueueBrowser queueBrowser = context.createBrowser(queue);
41
42         Enumeration<TextMessage> messages = queueBrowser.getEnumeration();
```

```

43     int count = 1;
44
45     while (messages.hasMoreElements()) {
46         TextMessage message = messages.nextElement();
47         System.out.println(count + " : " + message.getText());
48         count++;
49     }
50
51     // fechando
52     queueBrowser.close();
53     context.close();
54 }
55 }
```

Código Java 1.35: PercorrendoFilaGlassfish.java

35 Pare o **Wildfly**. Inicie o **Glassfish**. Execute algumas vezes a classe **EnviaNovoPedidoGlassfish** para popular a fila **pedidos** do **Glassfish** e depois execute a classe **PercorrendoFilaGlassfish**.

36 Adicione no pacote **br.com.k19.receptores** uma classe com main para percorrer as mensagens da fila **pedidos** do **Wildfly**.

```

1 package br.com.k19.receptores;
2
3 import java.util.Enumeration;
4 import java.util.Properties;
5
6 import javax.jms.ConnectionFactory;
7 import javax.jms.JMSContext;
8 import javax.jms.Queue;
9 import javax.jms.QueueBrowser;
10 import javax.jms.TextMessage;
11 import javax.naming.Context;
12 import javax.naming.InitialContext;
13
14 public class PercorrendoFilaWildfly {
15     public static void main(String[] args) throws Exception {
16         // serviço de nomes - JNDI
17         Properties props = new Properties();
18         props.put(Context.INITIAL_CONTEXT_FACTORY,
19                 "org.jboss.naming.remote.client.InitialContextFactory");
20         props.put(Context.PROVIDER_URL, "http-remoting://localhost:8080");
21         props.put(Context.SECURITY_PRINCIPAL, "k19");
22         props.put(Context.SECURITY_CREDENTIALS, "1234");
23
24         InitialContext ic = new InitialContext(props);
25
26         // fábrica de conexões JMS
27         ConnectionFactory factory = (ConnectionFactory) ic
28             .lookup("jms/RemoteConnectionFactory");
29
30         // fila
31         Queue queue = (Queue) ic.lookup("jms/queue/pedidos");
32
33         // contexto JMS
34         JMSContext context = factory.createContext("k19", "1234");
35
36         // queue browser
37         QueueBrowser queueBrowser = context.createBrowser(queue);
38
39         Enumeration<TextMessage> messages = queueBrowser.getEnumeration();
40         int count = 1;
41
42         while (messages.hasMoreElements()) {
```

```

43     TextMessage message = messages.nextElement();
44     System.out.println(count + " : " + message.getText());
45     count++;
46 }
47
48 // fechando
49 queueBrowser.close();
50 context.close();
51 }
52 }
```

Código Java 1.36: PercorrendoFilaWildfly.java

- 37 Pare o **Glassfish**. Inicie o **Wildfly**. Execute algumas vezes a classe **EnviaNovoPedidoWildfly** para popular a fila **pedidos** do **Wildfly** e depois execute a classe **PercorrendoFilaWildfly**.



Selecionando mensagens de um tópico

Podemos anexar propriedades às mensagens enviadas a um tópico JMS. As propriedades podem servir como filtro para os assinantes do tópico selecionarem as mensagens que eles desejam receber.

O código abaixo acrescenta uma propriedade a uma mensagem JMS.

```
1 message.setStringProperty("categoria", "esporte");
```

Quando um consumidor é criado associado a um tópico, podemos aplicar o critério de seleção das mensagens desejadas.

```
1 MessageConsumer consumer = session.createConsumer(topic, "(categoria = 'esporte')");
```



Exercícios de Fixação

- 38 Altere a classe **AssinanteDeNoticiasGlassfish** para selecionar somente as mensagens de esporte. Observe o trecho de código que você deve alterar:

```
1 // consumidor de mensagens
2 JMSConsumer consumer = context.createConsumer(topic, "(categoria = 'inutilidades')");
```

- 39 Altere a classe **EnviaNoticiaGlassfish** acrescentando uma propriedade à mensagem enviada. Observe o trecho de código que você deve alterar:

```

1 // mensagem
2 String mensagem = "Grazi Massafera foi ao shopping de chinelo";
3 TextMessage textMessage = context.createTextMessage();
4 textMessage.setStringProperty("categoria", "inutilidades");
5 textMessage.setText(mensagem);
6
7 // enviando
8 producer.send(topic, textMessage);
```

- 40 Pare o **Wildfly**. Inicie o **Glassfish**. Execute uma vez a classe **AssinanteDeNoticiasGlassfish** e depois a classe **EnviaNoticiaGlassfish**. Observe que a mensagem é recebida pelo assinante.

- 41 Altere o valor da propriedade da mensagem enviada.

```
1 textMessage.setStringProperty("categoria", "bobagens");
```

- 42 Execute uma vez a classe **AssinanteDeNoticiasGlassfish** e depois a classe **EnviaNoticiaGlassfish**. Observe que agora a mensagem não é recebida pelo assinante.

- 43 Altere a classe **AssinanteDeNoticiasWildfly** para selecionar somente as mensagens de esporte. Observe o trecho de código que você deve alterar:

```
1 // consumidor de mensagens
2 JMSConsumer consumer = context.createConsumer(topic, "(categoria = 'inutilidades')");
```

- 44 Altere a classe **EnviaNoticiaWildfly** acrescentando uma propriedade à mensagem enviada. Observe o trecho de código que você deve alterar:

```
1 // mensagem
2 String mensagem = "Grazi Massafera foi ao shopping de chinelo";
3 TextMessage textMessage = context.createTextMessage();
4 textMessage.setStringProperty("categoria", "inutilidades");
5 textMessage.setText(mensagem);
6
7 // enviando
8 producer.send(topic, textMessage);
```

- 45 Pare o **Glassfish**. Inicie o **Wildfly**. Execute uma vez a classe **AssinanteDeNoticiasWildfly** e depois a classe **EnviaNoticiaWildfly**. Observe que a mensagem é recebida pelo assinante.

- 46 Altere o valor da propriedade da mensagem enviada.

```
1 textMessage.setStringProperty("categoria", "bobagens");
```

- 47 Execute uma vez a classe **EnviaNoticiaWildfly**. Observe que agora a mensagem não é recebida pelo assinante.



Tópicos Duráveis

As mensagens enviadas para tópicos JMS normais só podem ser recebidas pelos assinantes que estiverem conectados no momento do envio. Dessa forma, se um assinante estiver desconectado ele perderá as mensagens enviadas durante o período off-line.

A especificação JMS define um tipo especial de tópico que armazena as mensagens dos assinantes desconectados e as envia assim que eles se conectarem. Esses são os **tópicos duráveis**.



Exercícios de Fixação

- 48** Pare o Wildfly. Inicie o Glassfish. Adicione uma propriedade na fábrica padrão de conexões JMS.

Additional Properties (1)			
Add Property	ClientId	Value	Id

- 49** Adicione no pacote **br.com.k19.receptores** uma classe com main para receber uma mensagem JMS do tópico **noticias** do Glassfish inclusive as enviadas enquanto a aplicação estiver off-line.

```

1 package br.com.k19.receptores;
2
3 import java.util.Properties;
4
5 import javax.jms.ConnectionFactory;
6 import javax.jms.JMSConsumer;
7 import javax.jms.JMSContext;
8 import javax.jms.Topic;
9 import javax.naming.InitialContext;
10
11 public class AssinanteDuravelGlassfish {

```

```

12 public static void main(String[] args) throws Exception {
13     // serviço de nomes - JNDI
14     Properties props = new Properties();
15
16     props.setProperty("java.naming.factory.initial",
17         "com.sun.enterprise.naming.SerialInitContextFactory");
18
19     props.setProperty("java.naming.factory.url.pkgs",
20         "com.sun.enterprise.naming");
21
22     props.setProperty("java.naming.factory.state",
23         "com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl");
24
25     InitialContext ic = new InitialContext(props);
26
27     // fábrica de conexões JMS
28     ConnectionFactory factory =
29         (ConnectionFactory) ic.lookup("jms/_defaultConnectionFactory");
30
31     // tópico
32     Topic topic = (Topic) ic.lookup("jms/noticias");
33
34     // contexto
35     JMSContext context = factory.createContext();
36
37     // consumidor de mensagens
38     JMSConsumer consumer = context.createDurableConsumer(topic, "Assinante1");
39
40     // consumindo a mensagem
41     String mensagem = consumer.receiveBody(String.class, 2000);
42
43     System.out.println(mensagem);
44
45     System.out.println("MENSAGEM RECEBIDA: " + mensagem);
46
47     // fechando
48     context.close();
49 }
50 }
```

Código Java 1.45: AssinanteDuravelGlassfish.java

- 50 Execute **uma** vez a classe **AssinanteDuravelGlassfish** para cadastrar um assinante.
- 51 Execute algumas vezes a classe **EnviaNoticiaGlassfish**. Perceba que o assinante está offline.
- 52 Execute algumas vezes a classe **AssinanteDuravelGlassfish** para receber as mensagens enviadas enquanto o assinante estava offline.
- 53 Pare o **Glassfish**. Inicie o **Wildfly**. Altere as configurações do para criar assinantes duráveis. Para isso modifique o arquivo de configuração **standalone-full.xml**. Você encontra esse arquivo na pasta **wildfly-8.2.0.Final/standalone/configuration**.

```

1 ...
2 <security-settings>
3     <security-setting match="#">
4         <permission type="createDurableQueue" roles="guest"/>
5         <permission type="deleteDurableQueue" roles="guest"/>
6         ...
7     </security-setting>
```

```
8 </security-settings>
9 . . .
```

Código XML 1.2: standalone-full.xml

- 54 Adicione no pacote **br.com.k19.receptores** uma classe com main para receber uma mensagem JMS do tópico **notícias** do **Wildfly** inclusive as enviadas enquanto a aplicação estiver off-line.

```
1 package br.com.k19.receptores;
2
3 import java.util.Properties;
4
5 import javax.jms.ConnectionFactory;
6 import javax.jms.JMSConsumer;
7 import javax.jms.JMSContext;
8 import javax.jms.Topic;
9 import javax.naming.Context;
10 import javax.naming.InitialContext;
11
12 public class AssinanteDuravelWildfly {
13     public static void main(String[] args) throws Exception {
14         // serviço de nomes - JNDI
15         Properties props = new Properties();
16         props.put(Context.INITIAL_CONTEXT_FACTORY,
17                 "org.jboss.naming.remote.client.InitialContextFactory");
18         props.put(Context.PROVIDER_URL, "http-remoting://localhost:8080");
19         props.put(Context.SECURITY_PRINCIPAL, "k19");
20         props.put(Context.SECURITY_CREDENTIALS, "1234");
21
22         InitialContext ic = new InitialContext(props);
23
24         // fábrica de conexões JMS
25         ConnectionFactory factory = (ConnectionFactory) ic
26             .lookup("jms/RemoteConnectionFactory");
27
28         // tópico
29         Topic topic = (Topic) ic.lookup("jms/topic/noticias");
30
31         // contexto JMS
32         JMSContext context = factory.createContext("k19", "1234");
33         context.setClientID("k19");
34
35         // consumidor de mensagens
36         JMSConsumer consumer = context.createDurableConsumer(topic, "k19");
37
38         // consumindo a mensagem
39         String mensagem = consumer.receiveBody(String.class, 2000);
40
41         System.out.println(mensagem);
42
43         System.out.println("MENSAGEM RECEBIDA: " + mensagem);
44
45         // fechando
46         context.close();
47     }
48 }
```

Código Java 1.46: AssinanteDuravelWildfly.java

- 55 Execute **uma** vez a classe **AssinanteDuravelWildfly** para cadastrar um assinante.

- 56 Execute algumas vezes a classe **EnviaNoticiaWildfly**. Perceba que o assinante está offline.

- 57 Execute algumas vezes a classe **AssinanteDuravelWildfly** para receber as mensagens enviadas enquanto o assinante estava offline.



JMS e EJB

A arquitetura JMS é intrinsecamente relacionada com a arquitetura EJB. Uma aplicação EJB pode receber e processar mensagens JMS de uma forma simples e eficiente.

A especificação EJB define um tipo de objeto especializado no recebimento de mensagens JMS. Esse objetos são os **Message Driven Beans (MDBs)**. Para implementar um MDB, devemos aplicar a anotação **@MessageDriven** e implementar a interface **MessageListener**.

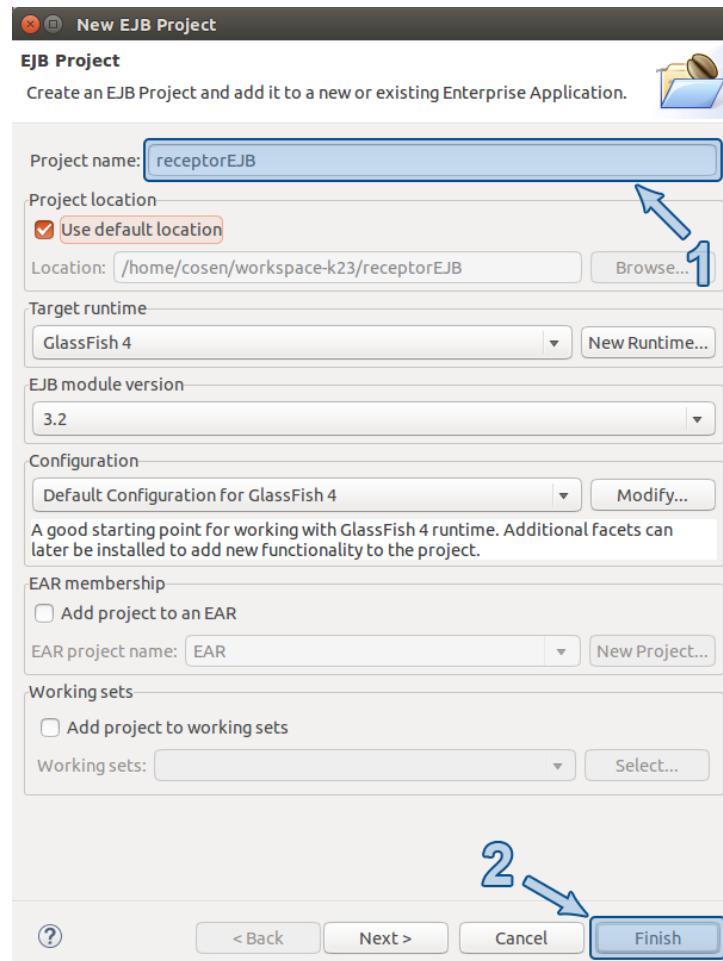
```
1 @MessageDriven(
2     activationConfig = {
3         @ActivationConfigProperty(
4             propertyName = "destinationLookup",
5             propertyValue = "jms/noticias"),
6         @ActivationConfigProperty(
7             propertyName = "destinationType",
8             propertyValue = "javax.jms.Topic")})
9 public class TratadorDeMensagensMDB implements MessageListener {
10
11     @Override
12     public void onMessage(Message message) {
13
14         try {
15             TextMessage msg = (TextMessage) message;
16             System.out.println(msg.getText());
17         } catch (JMSEException e) {
18             System.out.println("erro");
19         }
20     }
21 }
```

Código Java 1.47: *TratadorDeMensagensMDB.java*



Exercícios de Fixação

- 58 Crie um EJB Project no eclipse para implementar uma aplicação que possa receber as mensagens do tópico **notícias** do **Glassfish** criado anteriormente. Você pode digitar “CTRL + 3” em seguida “new EJB Project” e “ENTER”. Depois, siga exatamente a imagem abaixo.



59 Crie um pacote chamado **br.com.k19.mdbc** no projeto **receptorEJB**.

60 Adicione no pacote **br.com.k19.mdbc** um Message Driven Bean para receber e processar as mensagens do tópico **noticias** do **Glassfish**.

```

1 package br.com.k19.mdbc;
2
3 import javax.ejb.ActivationConfigProperty;
4 import javax.ejb.MessageDriven;
5 import javax.jms.JMSException;
6 import javax.jms.Message;
7 import javax.jms.MessageListener;
8 import javax.jms.TextMessage;
9
10 @MessageDriven(
11     activationConfig = {
12         @ActivationConfigProperty(
13             propertyName = "destinationLookup",
14             propertyValue = "jms/noticias"),
15         @ActivationConfigProperty(
16             propertyName = "destinationType",
17             propertyValue = "javax.jms.Topic")})
18 public class TratadorDeMensagensMDB implements MessageListener{
19     @Override
20     public void onMessage(Message message) {
21

```

```

22     try {
23         TextMessage msg = (TextMessage) message;
24         System.out.println(msg.getText());
25     } catch (JMSEException e) {
26         System.out.println("erro");
27     }
28 }
29 }
```

Código Java 1.48: TratadorDeMensagensMDB.java

- 61** Implante o projeto **receptorEJB** no **Glassfish** através da view **Servers**.
- 62** Pare o **Wildfly**. Inicie o **Glassfish**. Envie uma mensagem para o tópico **EnviaNoticiaGlassfish** e observe o console do eclipse.
- 63** Altere a classe **TratadorDeMensagensMDB** para receber e processar as mensagens do tópico **noticias** do **Wildfly**.

```

1 package br.com.k19.mdb;
2
3 import javax.ejb.ActivationConfigProperty;
4 import javax.ejb.MessageDriven;
5 import javax.jms.JMSEException;
6 import javax.jms.Message;
7 import javax.jms.MessageListener;
8 import javax.jms.TextMessage;
9
10 @MessageDriven(
11     activationConfig = {
12         @ActivationConfigProperty(
13             propertyName = "destinationLookup",
14             propertyValue = "jms/topic/noticias"),
15         @ActivationConfigProperty(
16             propertyName = "destinationType",
17             propertyValue = "javax.jms.Topic")})
18 public class TratadorDeMensagensMDB implements MessageListener{
19
20     @Override
21     public void onMessage(Message message) {
22
23         try {
24             TextMessage msg = (TextMessage) message;
25             System.out.println(msg.getText());
26         } catch (JMSEException e) {
27             System.out.println("erro");
28         }
29     }
30 }
```

Código Java 1.49: TratadorDeMensagensMDB.java

- 64** Implante o projeto **receptorEJB** no **Wildfly** através da view **Servers**.
- 65** Pare o **Glassfish**. Inicie o **Wildfly**. Envie uma mensagem para o tópico **EnviaNoticiaWildfly** e observe o console do eclipse.



Projeto - Rede de Hotéis

Para praticar os conceitos da arquitetura JMS, vamos implementar aplicações que devem se comunicar de forma automatizada. Suponha que uma rede de hotéis possua uma aplicação central para administrar os pagamentos realizados pelos clientes em qualquer unidade da rede. Cada hotel é controlado por uma aplicação local que deve enviar os dados necessários de cada pagamento para aplicação central.



Exercícios de Fixação

- 66 Adicione uma fila com o seguinte JNDI Name **jms/pagamentos** no Glassfish.
- 67 Implemente a aplicação local que deve executar em cada unidade da rede de hotéis através de uma aplicação Java SE. Essa aplicação deve obter os dados dos pagamentos através do Console do Eclipse e enviar uma mensagem JMS para a fila **jms/pagamentos** com esses dados. Utilize a classe **Scanner** para obter os dados digitados do Console.

```
1 // DICAS
2 Scanner entrada = new Scanner(System.in);
3 String linha = entrada.nextLine();
```

- 68 Implemente a aplicação central da rede de hotéis que deve tratar todas as mensagens enviadas à fila **jms/pagamentos** através de uma aplicação EJB. Essa aplicação deve utilizar Message Driven Beans. Simplesmente imprima no Console do Eclipse os dados extraídos das mensagens recebidas.
- 69 (Opcional) Faça a aplicação central enviar uma mensagem de confirmação para as aplicações locais a cada mensagem de pagamento processada. Dica: Utilize um tópico JMS e faça cada aplicação local filtrar as mensagens desse tópico que correspondem a pagamentos que ela enviou à aplicação central.



JAX-WS



Web Services

Muitas vezes, é necessário que os serviços oferecidos por uma organização sejam acessados diretamente pelos sistemas de outras organizações sem intervenção humana. Por exemplo, o Ministério da Fazenda do Brasil introduziu recentemente uma nova sistemática de emissão de notas fiscais. Hoje, as empresas podem descartar o método tradicional de emissão em papel e utilizar a emissão eletrônica. Inclusive, o serviço de emissão de nota fiscal eletrônica pode ser acessado diretamente pelos sistemas de cada empresa, automatizando o processo e consequentemente diminuindo gastos.

Diversos outros serviços possuem características semelhantes:

- Cotação de Moedas
- Previsão do Tempo
- Verificação de CPF ou CPNJ
- Cotação de Frete

Além da necessidade de serem acessados diretamente por sistemas e não por pessoas, geralmente, esses serviços devem ser disponibilizados através da internet para atingir um grande número de sistemas usuários. Nesse cenário, outra condição importante é que não exista nenhuma restrição quanto a plataforma utilizada em cada sistema.

Daí surge o conceito de **web service**. Resumidamente, um web service é um serviço oferecido por um sistema que pode ser acessado diretamente por outro sistema desenvolvido em qualquer tecnologia através de uma rede como a internet.

Cada plataforma oferece os recursos necessários para que os desenvolvedores possam disponibilizar ou acessar web services. A organização **W3C** define alguns padrões para definir o funcionamento de um web service. Em geral, as plataformas de maior uso comercial implementam a arquitetura definida pelos padrões da W3C.

Na plataforma Java, há especificações que definem a implementação Java dos padrões estabelecidos pelo W3C. A especificação java diretamente relacionada a Web Services que seguem os padrões da W3C é a **Java API for XML-Based Web Services - JAX-WS**. A JAX-WS depende fortemente de outra especificação Java, a **JAXB Java Architecture for XML Binding**.

A seguir mostraremos o funcionamento básico dos recursos definidos pela especificação JAXB e a arquitetura definida pela JAX-WS.



JAXB

A ideia principal da JAXB é definir o mapeamento e transformação dos dados de uma aplicação Java para XML e vice versa. Com os recursos do JAXB podemos transformar uma árvore de objetos Java em texto XML ou vice versa.

Suponha uma simples classe Java para modelar contas bancárias.

```
1 class Conta {  
2     private double saldo;  
3  
4     private double limite;  
5  
6     //GETTERS AND SETTERS  
7 }
```

Código Java 2.1: Conta.java

Para poder transformar objetos da classe conta em texto XML, devemos aplicar a anotação **@XMLElement**.

```
1 @XMLElement  
2 class Conta {  
3     ...  
4 }
```

Código Java 2.2: Conta.java

O contexto do JAXB deve ser criado para que as anotações de mapeamento possam ser processadas. O seguinte código cria o contexto do JAXB.

```
1 JAXBContext context = JAXBContext.newInstance(Conta.class);
```

O processo de transformar um objeto Java em texto XML é chamado de **marshal** ou **serialização**. Para serializar um objeto Java em XML, devemos obter através do contexto do JAXB um objeto da interface **Marshaller**.

```
1 Marshaller marshaller = context.createMarshaller();
```

Por fim, podemos criar um objeto da classe Conta e utilizar um Marshaller para serializá-lo em XML e guardar o conteúdo em um arquivo.

```
1 Conta conta = new Conta();  
2 conta.setLimite(2000);  
3 conta.setSaldo(2000);  
4  
5 marshaller.marshal(conta, new File("conta.xml"));
```

O processo inverso, ou seja, transformar um texto XML em um objeto da classe conta é denominado **unmarshal** ou **deserialização**. Para deserializar um XML, devemos obter através do contexto do JAXB um objeto da interface **Unmarshaller**.

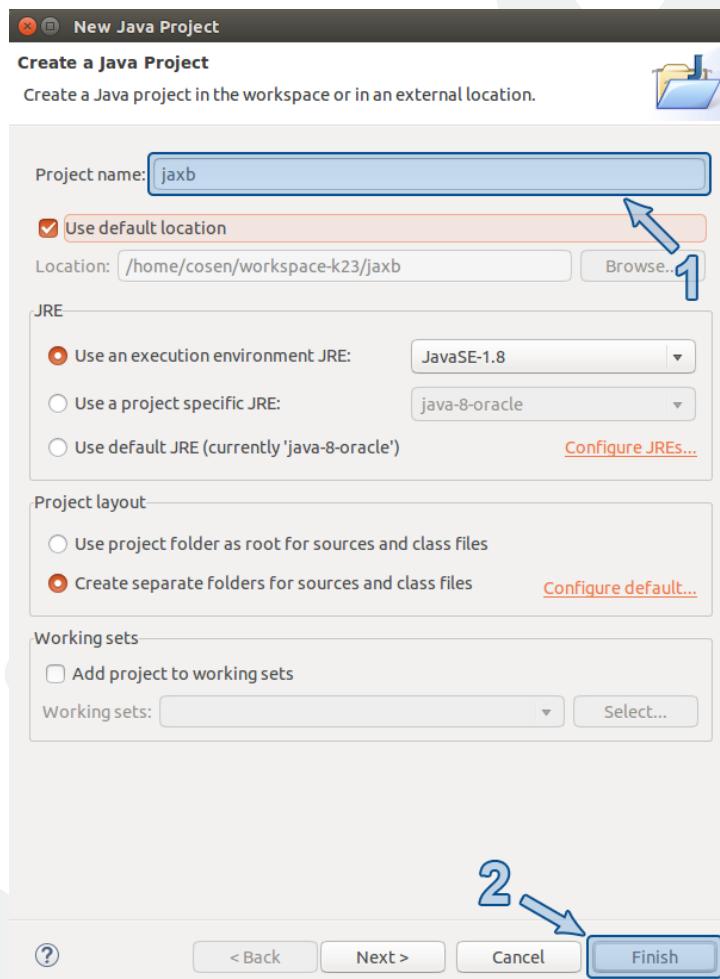
```
1 Unmarshaller unmarshaller = context.createUnmarshaller();  
2
```

```
3 Conta conta = (Conta) unmarshaller.unmarshal(new File("conta.xml"));
```



Exercícios de Fixação

- 1 Crie um Java Project no eclipse para testar o funcionamento básico dos recursos definidos pela especificação JAXB. Você pode digitar “CTRL + 3” em seguida “new Java Project” e “ENTER”. Depois, siga exatamente a imagem abaixo.



- 2 Crie um pacote chamado **jaxb** no projeto **jaxb**.

- 3 Adicione no pacote **jaxb** uma classe para modelar contas bancárias. Aplique a anotação **@XmlElement**.

```
1 @XmlRootElement
2 public class Conta {
3     private double saldo;
4 }
```

```

5  private double limite;
6
7  // GETTERS AND SETTERS
8 }
```

Código Java 2.7: Conta.java

- 4 Adicione no pacote **jaxb** uma classe com main para serializar um objeto da classe Conta.

```

1 public class Serializador {
2     public static void main(String[] args) throws JAXBException {
3
4         JAXBContext context = JAXBContext.newInstance(Conta.class);
5         Marshaller marshaller = context.createMarshaller();
6
7         Conta conta = new Conta();
8         conta.setLimite(2000);
9         conta.setSaldo(2000);
10        marshaller.marshal(conta, new File("conta.xml"));
11    }
12 }
```

Código Java 2.8: Serializador.java

- 5 Execute a classe **Serializador** e atualize o projeto para o arquivo **conta.xml** aparecer. Observe o conteúdo XML gerado.

- 6 Adicione no pacote **jaxb** uma classe com main para deserializar o XML criado anteriormente.

```

1 public class Deserializador {
2     public static void main(String[] args) throws JAXBException {
3
4         JAXBContext context = JAXBContext.newInstance(Conta.class);
5         Unmarshaller unmarshaller = context.createUnmarshaller();
6
7         Conta conta = (Conta) unmarshaller.unmarshal(new File("conta.xml"));
8
9         System.out.println(conta.getLimite());
10        System.out.println(conta.getSaldo());
11    }
12 }
```

Código Java 2.9: Deserializador.java

- 7 Execute a classe **Deserializador**.

- 8 Adicione no pacote **jaxb** uma classe para modelar os clientes donos das contas bancárias.

```

1 public class Cliente {
2     private String nome;
3
4     // GETTERS AND SETTERS
5 }
```

Código Java 2.10: Cliente.java

- 9 Altere a classe **Conta** para estabelecer um vínculo com a classe **Cliente**.

```

1 @XmlRootElement
2 public class Conta {
3     private double saldo;
4
5     private double limite;
6
7     private Cliente cliente;
8
9     // GETTERS AND SETTERS
10 }
```

Código Java 2.11: Conta.java

- 10 Altere a classe **Serializacao** e a teste novamente.

```

1 public class Serializador {
2     public static void main(String[] args) throws JAXBException {
3
4         JAXBContext context = JAXBContext.newInstance(Conta.class);
5         Marshaller marshaller = context.createMarshaller();
6
7         Cliente cliente = new Cliente();
8         cliente.setNome("Rafael Cosentino");
9
10        Conta conta = new Conta();
11        conta.setLimite(2000);
12        conta.setSaldo(2000);
13        conta.setCliente(cliente);
14
15        marshaller.marshal(conta, new File("conta.xml"));
16    }
17 }
```

Código Java 2.12: Serializador.java

- 11 Altere a classe **Deserializacao** e a teste novamente.

```

1 public class Deserializador {
2     public static void main(String[] args) throws JAXBException {
3
4         JAXBContext context = JAXBContext.newInstance(Conta.class);
5         Unmarshaller unmarshaller = context.createUnmarshaller();
6
7         Conta conta = (Conta) unmarshaller.unmarshal(new File("conta.xml"));
8
9         System.out.println(conta.getLimite());
10        System.out.println(conta.getSaldo());
11        System.out.println(conta.getCliente().getNome());
12    }
13 }
```

Código Java 2.13: Deserializador.java



Criando um web service - Java SE

Para começar, implementaremos um serviço e o disponibilizaremos como Web Service através dos recursos definido pela JAX-WS. Lembrando que a especificação JAX-WS é compatível com os

padrões do W3C. Essa implementação será realizada em ambiente Java SE.

Para exemplificar, suponha um serviço para gerar números aleatórios. A lógica desse serviço pode ser definida através de um método Java.

```
1 class Gerador {
2     public double geraNumero(){
3         return Math.random() * 200;
4     }
5 }
```

Código Java 2.14: Gerador.java

Para que essa classe seja interpretada como um web service devemos aplicar a anotação **@WebService**.

```
1 @WebService
2 class Gerador {
3     ...
4 }
```

Código Java 2.15: Gerador.java

Podemos publicar o serviço implementado pela classe **Gerador** através da classe **Endpoint**.

```
1 public class Publicador {
2     public static void main(String[] args) {
3         System.out.println("web service - Gerador Inicializado");
4         Gerador gerador = new Gerador();
5         Endpoint.publish("http://localhost:8080/gerador", gerador);
6     }
7 }
```

Código Java 2.16: Publicador.java

A definição do web service em **WSDL** pode ser consultada através da url <http://localhost:8080/gerador?wsdl>.



Consumindo um web service com JAX-WS

Agora, implementaremos um cliente de um web service que seja compatível com os padrões W3C utilizando os recursos do JAX-WS. O primeiro passo é utilizar a ferramenta **wsimport** para gerar as classes necessárias para acessar o web service a partir da definição do mesmo em WSDL. As classes geradas pelo wsimport não devem ser alteradas.

```
1 wsimport -keep http://localhost:8080/gerador?wsdl
```

Com o apoio das classes geradas pelo wsimport podemos consumir o web service.

```
1 public class Consumidor {
2     public static void main(String[] args) {
3         // Serviço
4         GeradorDeNumerosService service = new GeradorDeNumerosService();
5
6         // Proxy
7         GeradorDeNumeros proxy = service.getGeradorDeNumerosPort();
```

```

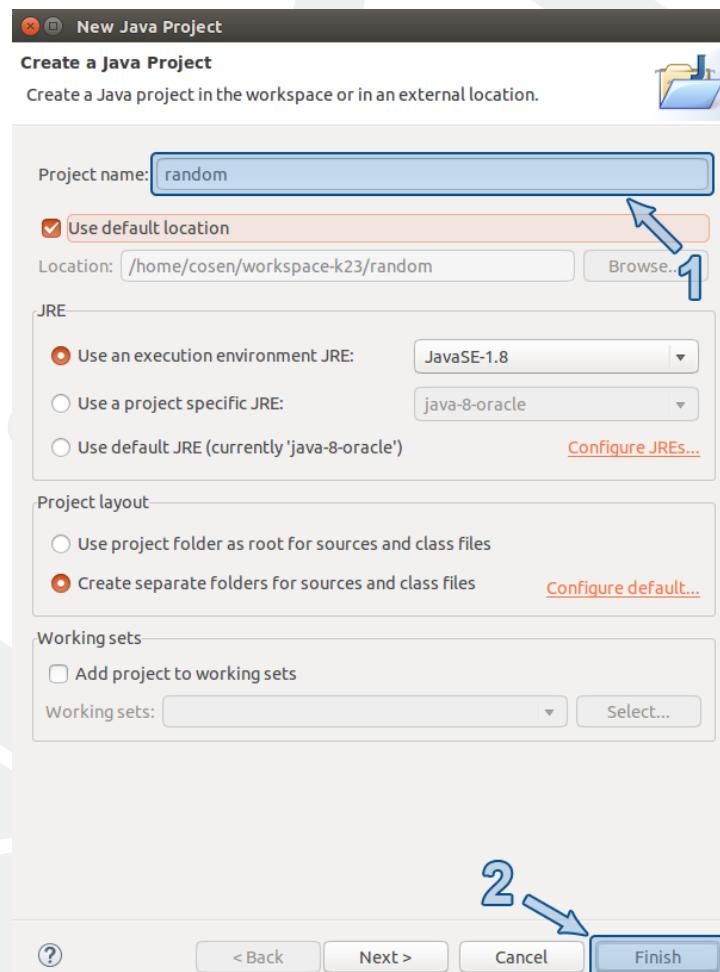
8     // Consumindo
9     double numero = proxy.geraNumero();
10    System.out.println(numero);
11 }
12 }
13 }
14 }
```

Código Java 2.18: Consumidor.java



Exercícios de Fixação

- 12 Crie um Java Project no eclipse para implementar um web service que gere números aleatórios. Você pode digitar “CTRL+3” em seguida “new Java Project” e “ENTER”. Depois, siga exatamente a imagem abaixo.



- 13 Crie um pacote chamado **webservices** no projeto **random**.

- 14 Adicione no pacote **webservices** uma classe para implementar o serviço de gerar números aleatórios.

```
1 @WebService  
2 public class Random {  
3     public double next(double max){  
4         return Math.random() * max;  
5     }  
6 }
```

Código Java 2.19: Random.java

- 15 Adicione no pacote **webservices** uma classe com main para publicar o web service.

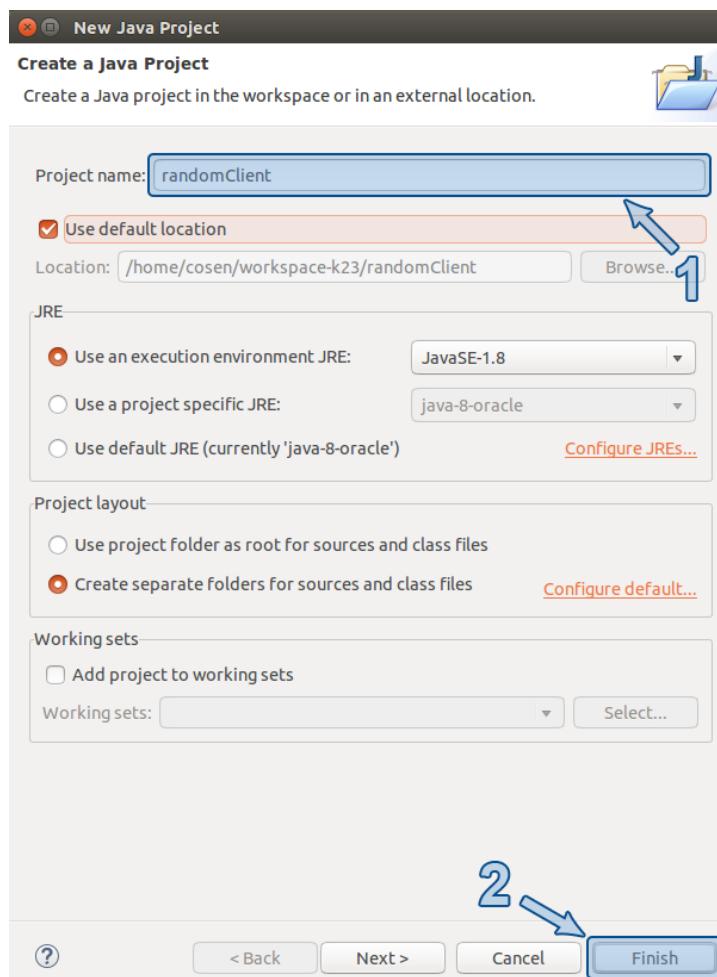
```
1 public class RandomPublisher {  
2     public static void main(String[] args) {  
3         System.out.println("Random web service start...");  
4         Random random = new Random();  
5         Endpoint.publish("http://localhost:8080/random", random);  
6     }  
7 }
```

Código Java 2.20: RandomPublisher.java

- 16 Execute a classe **RandomPublisher**.

- 17 Consulte através de um navegador a url <http://localhost:8080/random?wsdl> para conferir a definição do web service em WSDL.

- 18 Crie um Java Project no eclipse para consumir o web service que gera números aleatórios implementado anteriormente. Você pode digitar “CTRL+3” em seguida “new Java Project” e “ENTER”. Depois, siga exatamente a imagem abaixo.



- 19** Gere as classes necessárias para consumir o web service através da ferramenta **wsimport**. Abra um terminal e siga os passos abaixo.

```
cosen@k19:~/workspace-k23/randomClient/src$ wsimport -keep http://localhost:8080/random?wsdl
parsing WSDL...
generating code...
compiling code...
cosen@k19:~/workspace-k23/randomClient/src$
```

Terminal 2.1: wsimport

- 20** Atualize o projeto **randomClient**. Clique com o botão direito em cima desse projeto e depois clique com o esquerdo em **refresh**.

- 21** Adicione no pacote **webservices** do projeto **randomClient** uma classe para consumir o serviço de gerar números aleatórios.

```
1 public class Consumer {
2     public static void main(String[] args) {
3         // service
```

```

4 RandomService randomService = new RandomService();
5
6 // proxy
7 Random proxy = randomService.getRandomPort();
8
9 // operation
10 double next = proxy.next(50);
11 System.out.println(next);
12 }
13 }
```

Código Java 2.21: Consumer.java

- 22** Implemente outro consumidor só que agora em Ruby. Abra um terminal e siga os passos abaixo.

```

osen@k19:$ irb1.8
irb(main):001:0> require 'soap/wsdlDriver'
=> true
irb(main):002:0> wsdl = 'http://localhost:8080/random?wsdl'
=> "http://localhost:8080/random?wsdl"
irb(main):003:0> proxy = SOAP::WSDLDriverFactory.new(wsdl).create_rpc_driver
ignored attr: {}version
=> #<SOAP::RPC::Driver:#<SOAP::RPC::Proxy:http://localhost:8080/random>>
irb(main):004:0> proxy.next(:arg0 => 50).return
=> "22.9980548624178"
```

Terminal 2.2: irb

Autenticação

As especificações WSDL e SOAP não definem como o processo de autenticação deve ser realizado nos “Big Webservices”. Consequentemente, há diversas abordagens diferentes para tal tarefa. Mostraremos aqui uma solução que é adotada em muitos casos.

O cliente do serviço deve adicionar as informações necessárias para autenticação nos headers da requisição HTTP.

```

1 public class Consumer {
2     public static void main(String[] args) {
3         // service
4         RandomService randomService = new RandomService();
5
6         // proxy
7         Random proxy = randomService.getRandomPort();
8
9         // username e password
10        BindingProvider bp = (BindingProvider)proxy;
11        Map<String, List<String>> headers = new HashMap<String, List<String>>();
12        headers.put("Username", Collections.singletonList("k19"));
13        headers.put("Password", Collections.singletonList("k23"));
14        bp.getRequestContext().put(MessageContext.HTTP_REQUEST_HEADERS, headers);
15
16        // operation
17        double next = proxy.next(50);
18        System.out.println(next);
19    }
20 }
```

Código Java 2.22: Consumer.java

Depois, essas informações podem ser obtidas no serviço.

```

1  @WebService
2  public class Random {
3
4      @Resource
5      private WebServiceContext wsc;
6
7      public double next(double max){
8          MessageContext mc = wsc.getMessageContext();
9
10         //get detail from request headers
11         Map headers = (Map) mc.get(MessageContext.HTTP_REQUEST_HEADERS);
12         List usernameList = (List) headers.get("Username");
13         List passwordList = (List) headers.get("Password");
14
15         String username = usernameList != null ? usernameList.get(0).toString() : null;
16         String password = passwordList != null ? passwordList.get(0).toString() : null;
17
18         System.out.println(username);
19         System.out.println(password);
20
21         if("k19".equals(username) && "k23".equals(password)){
22             return Math.random() * max;
23         } else {
24             throw new RuntimeException("usuário ou senha incorretos");
25         }
26     }
27 }
```

Código Java 2.23: Random.java



Exercícios de Fixação

- 23 Altere a classe **Consumer** do projeto **randomClient**.

```

1  public class Consumer {
2      public static void main(String[] args) {
3          // service
4          RandomService randomService = new RandomService();
5
6          // proxy
7          Random proxy = randomService.getRandomPort();
8
9          // username e password
10         BindingProvider bp = (BindingProvider)proxy;
11         Map<String, List<String>> headers = new HashMap<String, List<String>>();
12         headers.put("Username", Collections.singletonList("k19"));
13         headers.put("Password", Collections.singletonList("k23"));
14         bp.getRequestContext().put(MessageContext.HTTP_REQUEST_HEADERS, headers);
15
16         // operation
17         double next = proxy.next(50);
18         System.out.println(next);
19     }
20 }
```

Código Java 2.24: Consumer.java

- 24 Agora, altere a classe **Random** do projeto **random**.

```
1  @WebService
```

```

2 public class Random {
3
4     @Resource
5     WebServiceContext wsc;
6
7     public double next(double max){
8         MessageContext mc = wsc.getMessageContext();
9
10        //get detail from request headers
11        Map headers = (Map) mc.get(MessageContext.HTTP_REQUEST_HEADERS);
12        List usernameList = (List) headers.get("Username");
13        List passwordList = (List) headers.get("Password");
14
15        String username = usernameList != null ? usernameList.get(0).toString() : null;
16        String password = passwordList != null ? passwordList.get(0).toString() : null;
17
18        System.out.println(username);
19        System.out.println(password);
20
21        if("k19".equals(username) && "k23".equals(password)){
22            return Math.random() * max;
23        } else {
24            throw new RuntimeException("usuário ou senha incorretos");
25        }
26    }
27 }

```

Código Java 2.25: Random.java

25 Execute a classe **RandomPublisher**. Depois, execute a classe **Consumer** e observe o console do eclipse.

26 Altere o usuário e senha definidos na classe **Consumer**.

```

1 headers.put("Username", Collections.singletonList("usuario"));
2 headers.put("Password", Collections.singletonList("senha"));

```

Código Java 2.26: Consumer.java

27 Novamente, execute a classe **RandomPublisher**. Depois, execute a classe **Consumer** e observe o console do eclipse.



Exercícios Complementares

1 Vamos monitorar as mensagens recebidas e enviadas pelo webservice do exercício através dos recursos de monitoramento do Eclipse. Siga os passos abaixo.

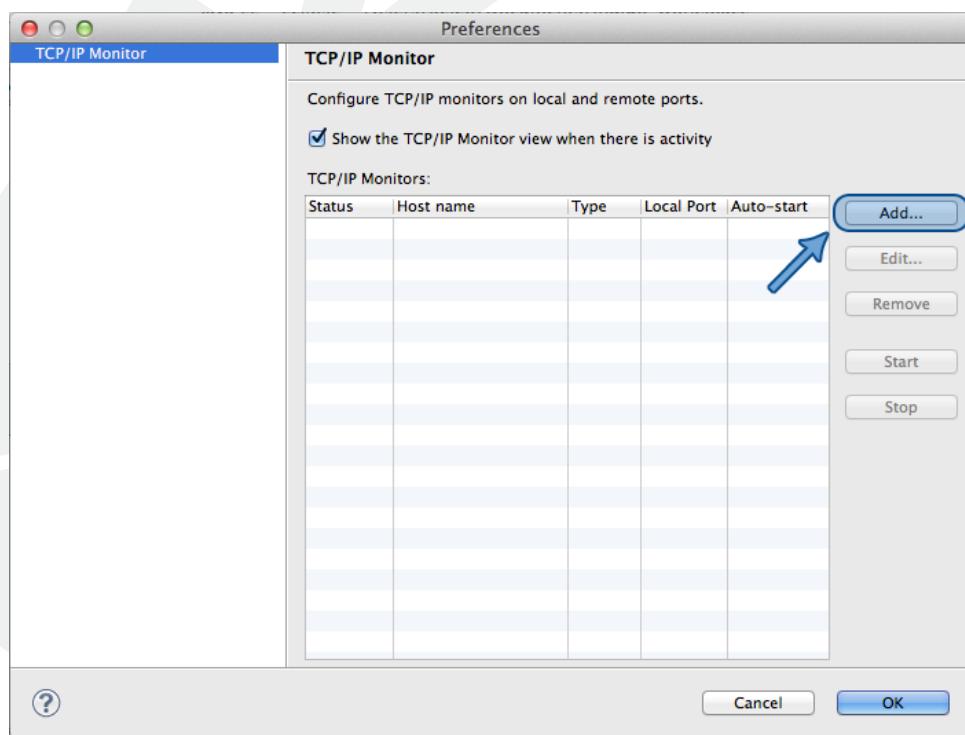
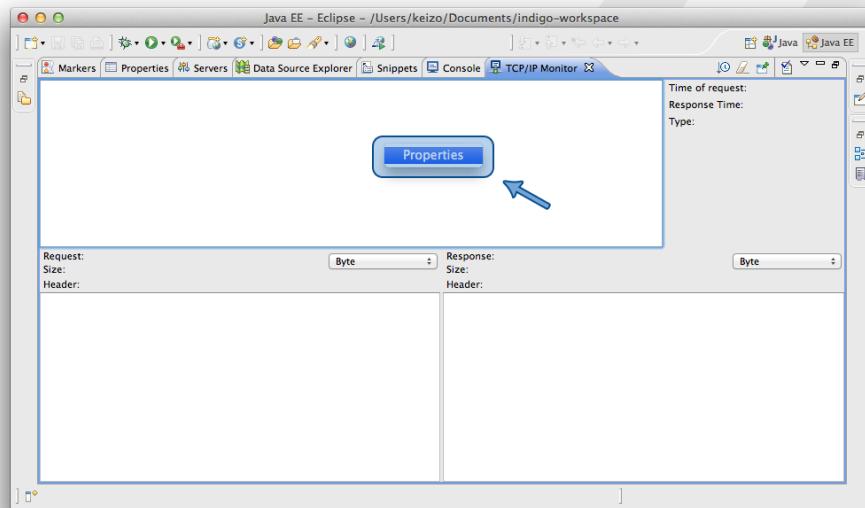
Altere a classe **RandomPublisher**.

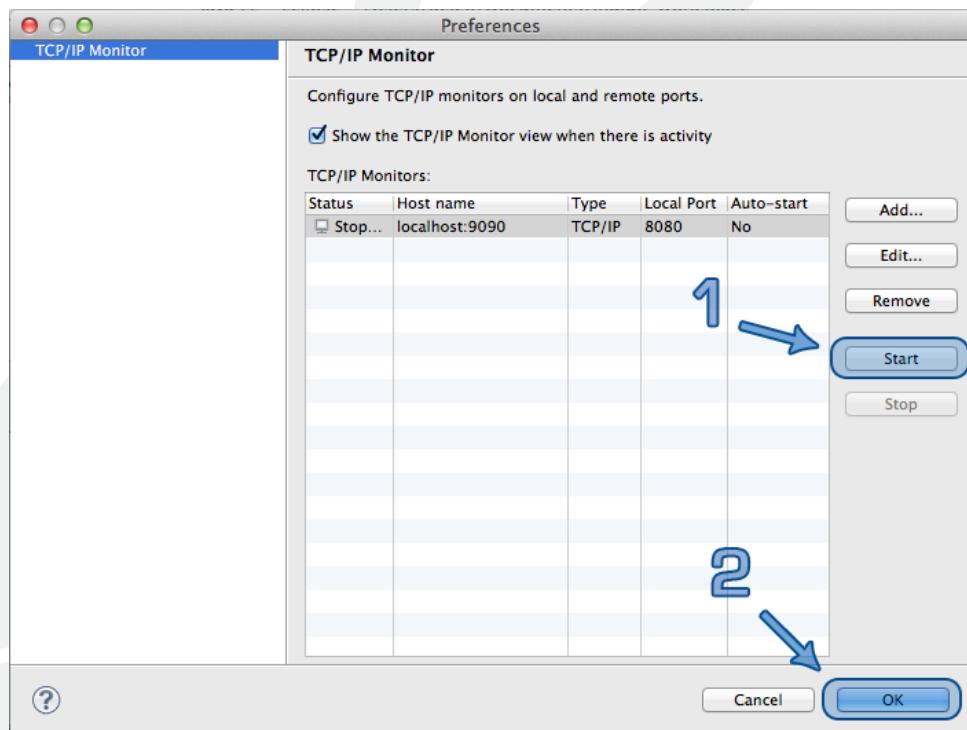
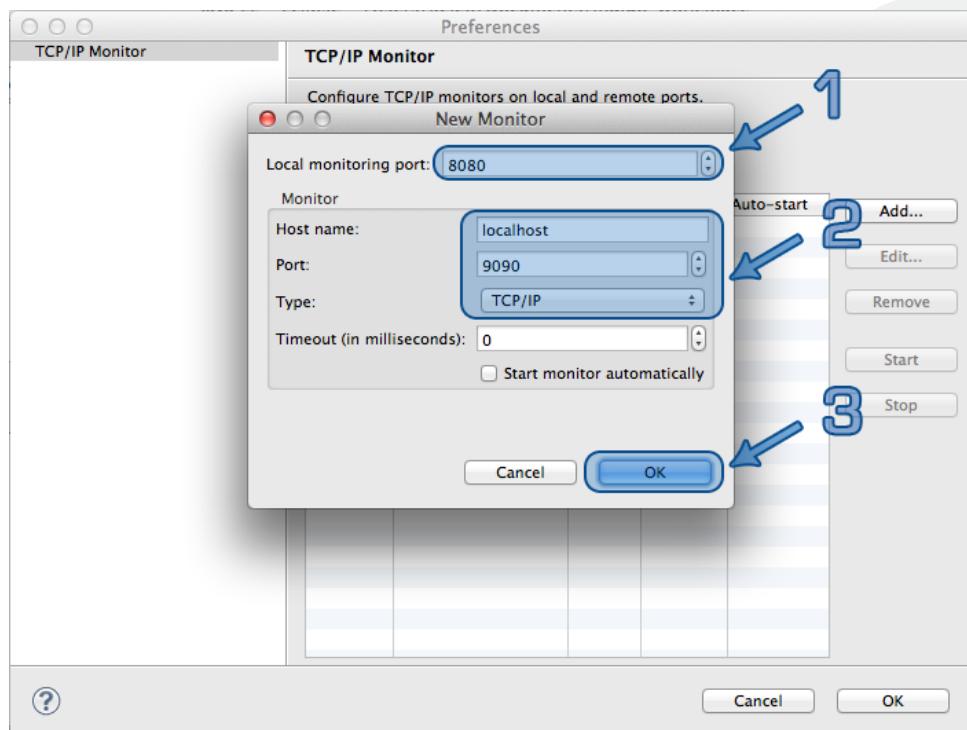
```

1 public class RandomPublisher {
2     public static void main(String[] args) {
3         System.out.println("Random web service start...");
4         Random random = new Random();
5         Endpoint.publish("http://localhost:9090/random", random);
6     }

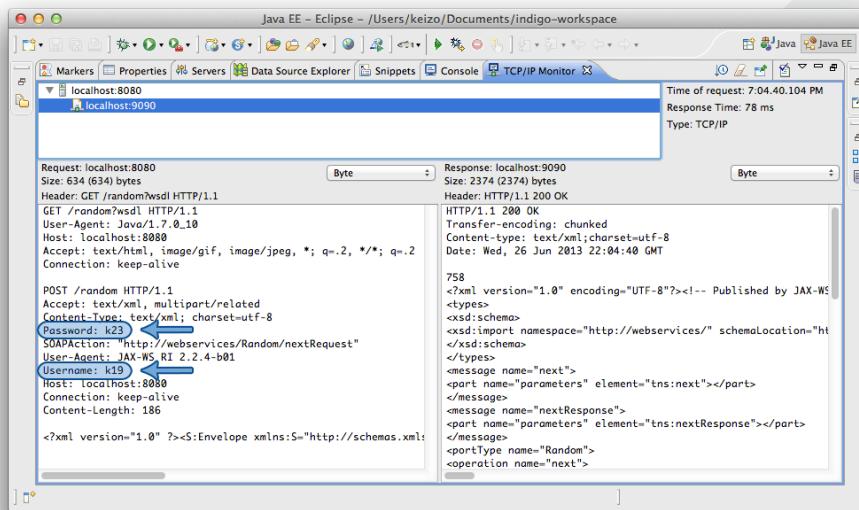
```

7 }

Código Java 2.27: RandomPublisher.java



- 2 Execute a classe **RandomPublisher** e depois a classe **Consumer**. Observe os headers contendo os dados para autenticação na view TCP/IP Monitor.



JAX-WS e EJB

Os recursos da arquitetura EJB podem ser utilizados juntamente com os recursos definidos pela especificação JAX-WS. Podemos expor um Stateless Session Bean ou um Singleton Session Bean como um Web Service que segue os padrões da W3C pois as duas especificações, EJB e JAX-WS, estão relacionadas.

Do ponto de vista da aplicação, basta aplicar a anotação **@WebService** na classe que implementa um Stateless Session Bean ou um Singleton Session Bean.

```

1 @WebService
2 @Stateless
3 public class Random {
4     public double next(double max){
5         return Math.random() * max;
6     }
7 }
```

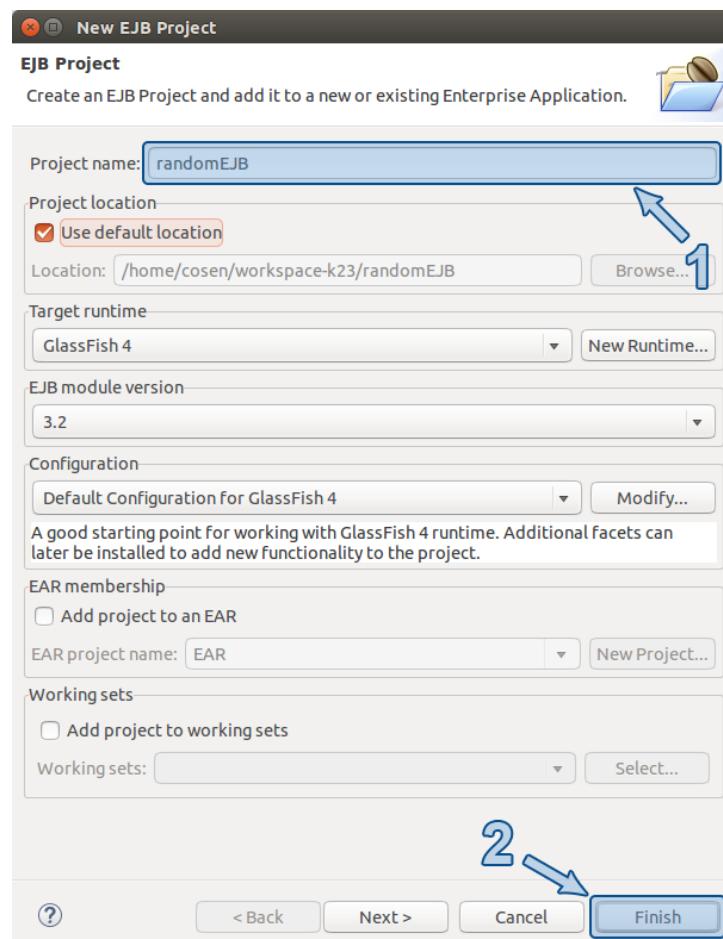
Código Java 2.28: Random.java

O container EJB publicará o web service ao inicializar a aplicação.



Exercícios de Fixação

- 28** Crie um EJB Project no eclipse para implementar um Stateless Session Bean que gere números aleatórios e exponha o como um web service. Você pode digitar “CTRL + 3” em seguida “new EJB Project” e “ENTER”. Depois, siga exatamente a imagem abaixo.



- 29** Crie um pacote chamado **webservices** no projeto **randomEJB**.
- 30** Adicione no pacote **webservices** uma classe para implementar o Stateless Session Bean.
- ```

1 @WebService
2 @Stateless
3 public class Random {
4 public double next(double max){
5 return Math.random() * max;
6 }
7 }
```
- Código Java 2.29: Random.java*
- 31** Implante o projeto **randomEJB** no Glassfish através da view **Servers**.
- 32** Acesse a url <http://localhost:8080/RandomService/Random?WSDL> para obter o WSDL que define o web service.
- 33** Acesse a url <http://localhost:8080/RandomService/Random?Tester> para testar o web service

através de uma página criada automaticamente pelo Glassfish.

- 34 Analogamente, implemente um Stateless Session Bean que ofereça as operações fundamentais da matemática.
- 35 (Opcional) Implemente um cliente Java SE para consumir esse web service.



## Projeto - Táxi no Aeroporto

---

Para melhorar o atendimento aos seus clientes, a diretoria de um aeroporto decidiu implantar um serviço próprio de táxi. Ela acredita que dessa forma conseguirá oferecer preços menores e maior qualidade para os seus passageiros. O aeroporto funcionará como um intermediário entre os passageiros e os taxistas. Os pagamentos serão realizados para o aeroporto e depois repassados para o proprietário do táxi. A contratação desse serviço pode ser realizada na chegada do passageiro ao aeroporto ou através da internet.

Como o aeroporto recebe pessoas de todas as partes do mundo, a diretoria quer oferecer a possibilidade dos seus passageiros efetuarem o pagamento do serviço de táxi com dinheiro de qualquer país. Daí surge a necessidade de obter a cotação das moedas. A equipe de TI decidiu obter as cotações através de um web service.

Para a emitir nota fiscal para brasileiros, o serviço de táxi deve verificar a validade do CPF do passageiro ou do CNPJ da empresa que ficará responsável pelo pagamento. Essa verificação deve ser realizada através um web service.

O valor cobrado dos passageiros dependerá do preço atual da gasolina e da distância a ser percorrida. Para obter o preço do combustível e calcular a distância, o serviço de táxi deverá consultar web services.



## Exercícios de Fixação

---

- 36 Implemente com dados fictícios um web service em JAX-WS que informe a cotação das moedas.
- 37 Implemente com lógica fictícia um web service em JAX-WS que realize a validação de CPF ou CNPJ.
- 38 Implemente com dados fictícios um web service em JAX-WS que informe o preço da gasolina.
- 39 Implemente com dados fictícios um web service em JAX-WS que calcule a distância entre dois locais.
- 40 Crie uma aplicação Java SE para implementar o serviço de táxi do aeroporto. Obtenha da entrada

padrão as seguintes informações

1. Moeda utilizada para o pagamento
2. CPF ou CNPJ
3. Endereço de destino

Acesse os web services criados anteriormente para obter as informações necessárias e imprima na saída padrão o valor a ser pago pelo passageiro.

**41** (Opcional) Substitua o web service de cotação de moeda implementado anteriormente pelo web service da **WebsericeX.NET**. O WSDL desse web service pode ser obtido através da <http://www.webservicex.net/CurrencyConvertor.asmx?WSDL>.

**42** (Opcional) Substitua a lógica fictícia de validação de CPF ou CNPJ pela lógica real.

**43** (Opcional) Pesquise algum serviço que calcule a distância entre dois pontos e implemente um cliente.

# JAX-RS



## REST vs Padrões W3C

No capítulo 2, vimos como implementar web services seguindo os padrões da W3C (WSDL, SOAP e XML). Em geral, a complexidade de evoluir um web service que segue os padrões W3C é alta pois qualquer alteração no serviço implica em uma nova definição em WSDL. Consequentemente, os proxies dos clientes devem ser atualizados.

Como alternativa, podemos desenvolver web services seguindo apenas os princípios do estilo arquitetural REST(Representational State Transfer - [http://www.ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm)). Em geral, web services que seguem os princípios REST são mais fáceis de implementar e evoluir.

Na plataforma Java, há especificações que definem o modelo de programação de web services Java que seguem os princípios REST. A principal especificação para esse tipo de web service é a **API for RESTful Web Services JAX-RS**.

A seguir mostraremos os conceitos principais do estilo arquitetural REST e o funcionamento básico dos recursos definidos pela especificação JAX-RS.



## Resources, URIs, Media Types e Operações

No estilo arquitetural REST, qualquer informação disponível é um **Resource**. O cadastro de uma pessoa, uma imagem, um documento e a cotação de uma moeda são exemplos de resources.

Cada resource deve possuir um identificador único. Esse identificador será utilizado para que o resource possa ser acessado. Na internet ou em uma intranet um web resources é identificado por uma **URI**(Uniform Resource Identifier - <http://tools.ietf.org/html/rfc3986>). Por exemplo, a URI [www.k19.com.br/cursos](http://www.k19.com.br/cursos) identifica na internet a página com os cursos da K19.

Os resources também podem ser representados em diversos formatos (**Media Type**). Novamente, na internet ou em uma intranet, seria normal que o cadastro de uma pessoa pudesse ser obtido em **html**, **xml** e **json**.

```
1 <html>
2 <head>
3 <title>Rafael Cosentino</title>
4 </head>
5
6 <body>
7 <h1>Rafael Cosentino</h1>
8 <p>Líder de treinamentos da K19</p>
9 </body>
```

```
10 </html>
```

```
1 <pessoa>
2 <nome>Rafael Cosentino</nome>
3 <descricao>Líder de treinamentos da K19</descricao>
4 </pessoa>
```

```
1 { "nome": "Rafael Cosentino", "descricao": "Líder de treinamentos da K19"}
```

Em uma arquitetura REST, um conjunto pequeno e fixo de operações deve ser definido previamente. As operações são utilizadas para manipular os recursos de alguma forma.

Por exemplo, na internet ou em uma intranet, os recursos são manipulados pelos métodos do protocolo HTTP. Podemos atribuir uma semântica diferente para cada método HTTP.

| Resource              | Método HTTP | Semântica                  |
|-----------------------|-------------|----------------------------|
| www.k19.com.br/cursos | GET         | pega a lista de cursos     |
| www.k19.com.br/cursos | POST        | adiciona um curso na lista |



## Web service com JAX-RS

A especificação JAX-RS define um modelo de programação para a criação de web services restful (web service que seguem os princípios do estilo arquitetural REST).



## Resources

De acordo com a JAX-RS, os web resources são implementados por classes Java (resource classes). Todo web resource deve possuir uma URI que é definida parcialmente pela anotação **@Path**.

```
1 @Path("/Cotacao")
2 public class CotacaoResource {
3 ...
4 }
```

Código Java 3.1: CotacaoResource.java

Os métodos HTTP podem ser mapeados para métodos Java de uma resource class. As anotações **@GET**, **@PUT**, **@POST**, **@DELETE** e **@HEAD** são utilizadas para realizar esse mapeamento.

```
1 @Path("/Cotacao")
2 class CotacaoResource {
3
4 @GET
5 public String getCotacao(){
6 // implementação
7 }
8 }
```

*Código Java 3.2: CotacaoResource.java*

O Media Type que será utilizado para a representação do resource pode ser definido através da anotação **@Produces** e o do enum **MediaType**.

```

1 @Path("/Cotacao")
2 class CotacaoResource {
3
4 @GET
5 @Produces(MediaType.TEXT_PLAIN)
6 public String getCotacao(){
7 // implementação
8 }
9 }
```

*Código Java 3.3: CotacaoResource.java*



## Subresource

A princípio, uma resource class define apenas um resource. Porém, podemos definir subresources dentro de uma resource class através de métodos anotados com **@Path**.

```

1 @Path("/Cotacao")
2 class CotacaoResource {
3
4 @GET
5 @Path("/DollarToReal")
6 @Produces(MediaType.TEXT_PLAIN)
7 public String getCotacaoDollarToReal(){
8 // implementação
9 }
10
11 @GET
12 @Path("/EuroToReal")
13 @Produces(MediaType.TEXT_PLAIN)
14 public String getCotacaoEuroToReal(){
15 // implementação
16 }
17 }
```

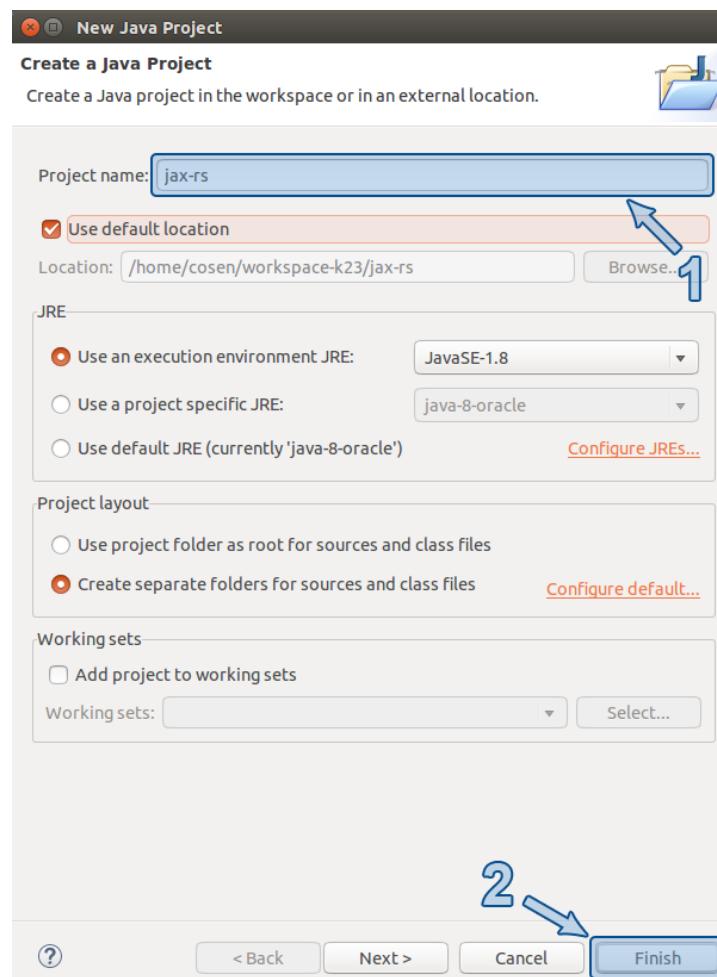
*Código Java 3.4: CotacaoResource.java*

O sufixo da URI de um subresource é definido pela concatenação do valor da anotação **@Path** aplicada na resource class com o valor da anotação **@Path** aplicada no método correspondente ao subresource. No exemplo acima, temos dois subresources com URI que possuem os seguintes sufixos: **/Cotacao/DollarToReal** e **/Cotacao/EuroToReal**.



## Exercícios de Fixação

- 1 Crie um Java Project no eclipse para testar o funcionamento básico dos recursos definidos pela especificação JAX-RS. Você pode digitar “CTRL + 3” em seguida “new Java Project” e “ENTER”. Depois, siga exatamente a imagem abaixo.



2 JAX-RS é uma especificação. Para testar os recursos definidos nessa especificação temos que escolher uma implementação. Aqui, utilizaremos o projeto Jersey que implementa a JAX-RS. Crie uma pasta chamada **lib** no projeto **jax-rs**.

3 Copie todos os JARs contidos no arquivo **jersey-libs.zip** para a pasta **lib**. Esse arquivo pode ser encontrado na pasta **K19-Arquivos** na sua Área de Trabalho.



#### Importante

Você também pode obter esse arquivo através do site da K19: [www.k19.com.br/arquivos](http://www.k19.com.br/arquivos).

4 Adicione esses arquivos no classpath do projeto.

5 Crie um pacote chamado **resources** no projeto **jax-rs**.

- 6 Adicione no pacote **br.com.k19.resources** uma classe para implementar um resource de cotação de moeda.

```

1 package br.com.k19.resources;
2
3 import javax.ws.rs.GET;
4 import javax.ws.rs.Path;
5 import javax.ws.rs.Produces;
6 import javax.ws.rs.core.MediaType;
7
8 @Path("/Cotacao")
9 public class CotacaoResource {
10
11 @GET
12 @Path("/DollarToReal")
13 @Produces(MediaType.TEXT_PLAIN)
14 public String getCotacaoDollarToReal() {
15 return "3.37";
16 }
17
18 @GET
19 @Path("/EuroToReal")
20 @Produces(MediaType.TEXT_PLAIN)
21 public String getCotacaoEuroToReal() {
22 return "3.68";
23 }
24 }
```

Código Java 3.5: CotacaoResource.java

- 7 Crie um pacote chamado **br.com.k19.main** no projeto **jax-rs**.

- 8 Adicione no pacote **br.com.k19.main** uma classe para publicar o web service.

```

1 package br.com.k19.main;
2
3 import java.io.IOException;
4 import java.net.URI;
5
6 import org.glassfish.jersey.jdkhttp.JdkHttpServerFactory;
7 import org.glassfish.jersey.server.ResourceConfig;
8
9 import com.sun.net.httpserver.HttpServer;
10
11 public class Publicador {
12 public static void main(String[] args) throws IllegalArgumentException, IOException {
13 ResourceConfig rc = new ResourceConfig().
14 packages(new String[] { "br.com.k19.resources" });
15
16 HttpServer server = JdkHttpServerFactory.
17 createHttpServer(URI.create("http://localhost:8080/"), rc);
18 }
19 }
```

Código Java 3.6: Publicador.java

- 9 Execute a classe **Publicador**.

- 10 Abra o firefox e acesse as seguintes urls: <http://localhost:8080/Cotacao/DollarToReal> e

<http://localhost:8080/Cotacao/EuroToReal>.

- 11 No firefox abra o Poster (clique no link no canto inferior direito) e faça uma requisição para cada uma das seguintes urls: <http://localhost:8080/Cotacao/DollarToReal> e <http://localhost:8080/Cotacao/EuroToReal>.



## Parâmetros

A especificação JAX-RS define um mecanismo bem simples para injetar os dados contidos nas requisições HTTP nos métodos Java das resource classes. Veremos a seguir que tipo de dados podem ser injetados e como injetá-los.

### PathParam

Suponha que desejamos utilizar uris com o seguinte formato para realizar a cotação de moedas.

**/Cotacao/Dollar/Real:** Valor do Dollar em Real.

**/Cotacao/Euro/Real:** Valor do Euro em Real.

**/Cotacao/Moeda1/Moeda2:** Valor da Moeda1 na Moeda2.

Para trabalhar com uris com esse formato, podemos definir parâmetros na URI de um resource através da anotação `@PathParam`.

1 `@Path("/{M1}/{M2}")`

Os parâmetros definidos através da anotação `@Path` podem ser recuperados através da anotação `@PathParam` que deve ser aplicada nos argumentos dos métodos de uma resource class.

```
1 @Path("/Cotacao")
2 public class CotacaoResource {
3
4 @GET
5 @Path("/{M1}/{M2}")
6 @Produces(MediaType.TEXT_PLAIN)
7 public String cotacao(@PathParam("M1") String m1, @PathParam("M2") String m2){
8 // implementacao
9 }
10 }
```

Código Java 3.8: CotacaoResource.java

### MatrixParam

Suponha que desejamos utilizar uris com o seguinte formato para realizar a cotação de moedas.

**/Cotacao;M1=dollar;M2=real:** Valor do Dollar em Real.

**/Cotacao;M1=euro;M2=real:** Valor do Euro em Real.

**/Cotacao;M1=moeda1;M2=moeda2:** Valor da Moeda1 na Moeda2.

As URIs acima possuem dois **matrix params**: M1 e M2. Esses parâmetros podem ser recuperados através da anotação **@MatrixParam** que deve ser aplicada nos argumentos dos métodos de uma resource class.

```

1 @Path("/Cotacao")
2 public class CotacaoResource {
3
4 @GET
5 @Produces(MediaType.TEXT_PLAIN)
6 public String cotacao(@MatrixParam("M1") String m1, @MatrixParam("M2") String m2){
7 // implementacao
8 }
9 }
```

Código Java 3.9: CotacaoResource.java

## QueryParam

Suponha que desejamos utilizar uris com o seguinte formato para realizar a cotação de moedas.

**/Cotacao?M1=dollar&M2=real:** Valor do Dollar em Real.

**/Cotacao?M1=euro&M2=real:** Valor do Euro em Real.

**/Cotacao?M1=moeda1&M2=moeda2:** Valor da Moeda1 na Moeda2.

As URIs acima possuem dois **query params**: M1 e M2. Esses parâmetros podem ser recuperados através da anotação **@QueryParam** que deve ser aplicada nos argumentos dos métodos de uma resource class.

```

1 @Path("/Cotacao")
2 public class CotacaoResource {
3
4 @GET
5 @Produces(MediaType.TEXT_PLAIN)
6 public String cotacao(@QueryParam("M1") String m1, @QueryParam("M2") String m2){
7 // implementacao
8 }
9 }
```

Código Java 3.10: CotacaoResource.java

## FormParam

Parâmetros enviados através de formulários HTML que utilizam o método POST do HTTP podem ser recuperados através da anotação **@FormParam**.

```

1 <form action="http://www.k19.com.br/Cotacao" method="POST">
2 Moeda1: <input type="text" name="M1">
3 Moeda2: <input type="text" name="M2">
4 </form>
```

```

1 @Path("/Cotacao")
2 public class CotacaoResource {
3
4 @POST
5 @Produces(MediaType.TEXT_PLAIN)
6 public String cotacao(@FormParam("M1") String m1, @FormParam("M2") String m2){
7 // implementacao
8 }
9 }
```

Código Java 3.11: CotacaoResource.java

## HeaderParam

Os headers HTTP podem ser recuperados através da anotação **@HeaderParam**.

```

1 @Path("/test-header-params")
2 public class UserAgentResource {
3
4 @GET
5 @Produces(MediaType.TEXT_PLAIN)
6 public String userAgent(@HeaderParam("User-Agent") String userAgent){
7 return userAgent;
8 }
9 }
```

Código Java 3.12: UserAgentResource.java

## CookieParam

Os valores dos cookies enviados nas requisições HTTP podem ser recuperados através da anotação **@CookieParam**.

```

1 @Path("/cookie")
2 public class CookieResource {
3
4 @GET
5 @Produces(MediaType.TEXT_PLAIN)
6 public String userAgent(@CookieParam("clienteID") String clienteID){
7 return clienteID;
8 }
9 }
```

Código Java 3.13: CookieResource.java



## Exercícios de Fixação

- 12** Adicione no pacote **br.com.k19.resources** do projeto **jax-rs** uma classe para testar os path params.

```

1 package br.com.k19.resources;
2
3 import javax.ws.rs.GET;
4 import javax.ws.rs.Path;
5 import javax.ws.rs.PathParam;
6 import javax.ws.rs.Produces;
7 import javax.ws.rs.core.MediaType;
```

```

8 @Path("/path-param")
9 public class TestaPathParamResource {
10
11 @GET
12 @Path("/{p1}/{p2}")
13 @Produces(MediaType.TEXT_PLAIN)
14 public String pathParam(@PathParam("p1") String p1,
15 @PathParam("p2") String p2) {
16 return "P1 = " + p1 + ", P2 = " + p2;
17 }
18 }
19

```

Código Java 3.14: TestaPathParamResource.java

**13** Execute a classe **Publicador**.

**14** Acesse as seguintes URIs para testar:

- <http://localhost:8080/path-param/1/2>
- <http://localhost:8080/path-param/java/csharp>

**15** Adicione no pacote **br.com.k19.resources** do projeto **jax-rs** uma classe para testar os matrix params.

```

1 package br.com.k19.resources;
2
3 import javax.ws.rs.GET;
4 import javax.ws.rs.MatrixParam;
5 import javax.ws.rs.Path;
6 import javax.ws.rs.Produces;
7 import javax.ws.rs.core.MediaType;
8
9 @Path("/matrix-param")
10 public class TestaMatrixParamResource {
11
12 @GET
13 @Produces(MediaType.TEXT_PLAIN)
14 public String pathParam(@MatrixParam("p1") String p1,
15 @MatrixParam("p2") String p2) {
16 return "P1 = " + p1 + ", P2 = " + p2;
17 }
18 }

```

Código Java 3.15: TestaMatrixParamResource.java

**16** Execute a classe **Publicador**.

**17** Acesse as seguintes URIs para testar:

- <http://localhost:8080/matrix-param;p1=1;p2=2>
- <http://localhost:8080/matrix-param;p1=java;p2=csharp>

- 18** Adicione no pacote **br.com.k19.resources** do projeto **jax-rs** uma classe para testar os query params.

```

1 package br.com.k19.resources;
2
3 import javax.ws.rs.GET;
4 import javax.ws.rs.Path;
5 import javax.ws.rs.Produces;
6 import javax.ws.rs.QueryParam;
7 import javax.ws.rs.core.MediaType;
8
9 @Path("/query-param")
10 public class TestaQueryParamResource {
11
12 @GET
13 @Produces(MediaType.TEXT_PLAIN)
14 public String pathParam(@QueryParam("p1") String p1,
15 @QueryParam("p2") String p2) {
16 return "P1 = " + p1 + ", P2 = " + p2;
17 }
18 }
```

Código Java 3.16: *TestaQueryParamResource.java*

- 19** Execute a classe **Publicador**.

- 20** Acesse as seguintes URIs para testar:

- <http://localhost:8080/query-param?p1=1&p2=2>
- <http://localhost:8080/query-param?p1=java&p2=csharp>



## URI Matching

Podemos utilizar expressão regular para restringir o formato das URIs associadas aos Resources.

```

1 @Path("/uri-matching")
2 public class URIMatchingResource {
3
4 @GET
5 @Path("/{a: \d*}/{b: \d*}")
6 @Produces(MediaType.TEXT_PLAIN)
7 public String soma(@PathParam("a") double a, @PathParam("b") double b) {
8 return a + b + "";
9 }
10
11 @GET
12 @Path("/{a}/{b}")
13 @Produces(MediaType.TEXT_PLAIN)
14 public String concatena(@PathParam("a") String a, @PathParam("b") String b) {
15 return a + b;
16 }
17 }
```

Código Java 3.17: *URIMatchingResource.java*



## Exercícios de Fixação

- 21** Adicione no pacote **br.com.k19.resources** do projeto **jax-rs** uma classe para testar o URI Matching.

```

1 package br.com.k19.resources;
2
3 import javax.ws.rs.GET;
4 import javax.ws.rs.Path;
5 import javax.ws.rs.PathParam;
6 import javax.ws.rs.Produces;
7 import javax.ws.rs.core.MediaType;
8
9 @Path("/uri-matching")
10 public class URIMatchingResource {
11
12 @GET
13 @Path("/{a: \d*}/{b: \d*}")
14 @Produces(MediaType.TEXT_PLAIN)
15 public String soma(@PathParam("a") double a, @PathParam("b") double b) {
16 return a + b + "";
17 }
18
19 @GET
20 @Path("/{a}/{b}")
21 @Produces(MediaType.TEXT_PLAIN)
22 public String concatena(@PathParam("a") String a, @PathParam("b") String b) {
23 return a + b;
24 }
25 }
```

Código Java 3.18: URIMatchingResource.java

- 22** Execute a classe **Publicador**.

- 23** Acesse as seguintes URIs para testar:

- <http://localhost:8080/uri-matching/10/20>
- <http://localhost:8080/uri-matching/107/3>
- <http://localhost:8080/uri-matching/k/19>
- <http://localhost:8080/uri-matching/rafael/cosentino>



## HTTP Headers com @Context

Podemos recuperar os headers HTTP através da injeção com **@Context**

```

1 @Path("/http-headers")
2 public class HttpHeadersResource {
```

```

4 @GET
5 public String addUser(@Context HttpHeaders headers) {
6
7 String userAgent = headers.getRequestHeader("user-agent").get(0);
8
9 return "user-agent: " + userAgent;
10 }
11 }
```

Código Java 3.19: *HttpHeadersResource.java*



## Exercícios de Fixação

- 24** Adicione no pacote **br.com.k19.resources** do projeto **jax-rs** uma classe para testar a injeção do **HttpHeaders**.

```

1 package br.com.k19.resources;
2
3 import javax.ws.rs.GET;
4 import javax.ws.rs.Path;
5 import javax.ws.rs.core.Context;
6 import javax.ws.rs.core.HttpHeaders;
7
8 @Path("/http-headers")
9 public class HttpHeadersResource {
10
11 @GET
12 public String addUser(@Context HttpHeaders headers) {
13
14 String userAgent = headers.getRequestHeader("user-agent").get(0);
15
16 return "user-agent: " + userAgent;
17 }
18 }
```

Código Java 3.20: *HttpHeadersResource.java*

- 25** Execute a classe **Publicador**.

- 26** Acesse a seguintes URI para testar:

- <http://localhost:8080/http-headers>



## Download de arquivos

Podemos realizar downloads de arquivos de texto, imagens, pdfs, entre outros com jax-rs.

```

1 @Path("/download")
2 public class DownloadResource {
3
4 @GET
5 @Path("/texto")
```

```

6 @Produces("text/plain")
7 public Response getTexto() {
8
9 File file = new File("texto.txt");
10
11 ResponseBuilder response = Response.ok((Object) file);
12 response.header("Content-Disposition",
13 "attachment; filename=\"texto.txt\"");
14 return response.build();
15
16 }
17
18 @GET
19 @Path("/imagem")
20 @Produces("image/png")
21 public Response getImagen() {
22
23 File file = new File("k19-logo.png");
24
25 ResponseBuilder response = Response.ok((Object) file);
26 response.header("Content-Disposition",
27 "attachment; filename=\"k19-logo.png\"");
28 return response.build();
29
30 }
31 }
```

Código Java 3.21: DownloadResource.java



## Exercícios de Fixação

- 27 Adicione no pacote **br.com.k19.resources** do projeto **jax-rs** uma classe para testar o download de arquivos.

```

1 package br.com.k19.resources;
2
3 import java.io.File;
4
5 import javax.ws.rs.GET;
6 import javax.ws.rs.Path;
7 import javax.ws.rs.Produces;
8 import javax.ws.rs.core.Response;
9 import javax.ws.rs.core.ResponseBuilder;
10
11 @Path("/download")
12 public class DownloadResource {
13
14 @GET
15 @Path("/texto")
16 @Produces("text/plain")
17 public Response getTexto() {
18
19 File file = new File("texto.txt");
20
21 ResponseBuilder response = Response.ok((Object) file);
22 response.header("Content-Disposition",
23 "attachment; filename=\"texto.txt\"");
24 return response.build();
25
26 }
27
28 @GET
29 @Path("/imagem")
```

```
30 @Produces("image/png")
31 public Response getImagen() {
32
33 File file = new File("k19-logo.png");
34
35 ResponseBuilder response = Response.ok((Object) file);
36 response.header("Content-Disposition",
37 "attachment; filename=\"k19-logo.png\"");
38 return response.build();
39
40 }
41 }
```

Código Java 3.22: DownloadResource.java

- 28 Crie um arquivo chamado **texto.txt** no projeto **jax-rs**. Adicione algum conteúdo aleatório nesse arquivo.
- 29 Copie o arquivo **k19-logo.png** da pasta **K19-Arquivos/imagens** da Área de Trabalho para o projeto **jax-rs**.



### Importante

Você também pode obter o arquivo **k19-logo.png** através do site da K19: [www.k19.com.br/arquivos](http://www.k19.com.br/arquivos).

- 30 Execute a classe **Publicador**.
- 31 Acesse a seguintes URI para testar:
  - <http://localhost:8080/download/texto>
  - <http://localhost:8080/download/imagen>
- 32 Analogamente aos exercícios anteriores, implemente o download de um arquivo PDF.



## Produzindo XML ou JSON

A especificação JAX-RS utiliza como base os recursos definidos pela especificação JAXB para produzir XML e JSON. A princípio os recursos do JAXB possibilitam apenas a produção de XML. Contudo, a arquitetura JAXB é flexível e pode ser facilmente estendida para produzir JSON também.

Suponha que seja necessário implementar um web service que manipule a seguinte entidade:

```
1 class Produto {
2 private String nome;
3
4 private Double preco;
```

```

5 private Long id;
6
7 // GETTERS AND SETTERS
8
9 }
```

*Código Java 3.23: Produto.java*

Adicionando a anotação **@XMLRootElement** da especificação JAXB na classe Produto, podemos gerar produtos em XML ou JSON.

```

1 @XMLElement
2 class Produto {
3 ...
4 }
```

*Código Java 3.24: Produto.java*

Agora, basta definir o Media Type nos métodos de uma resource class de acordo com o formato que desejamos utilizar, XML ou JSON.

```

1 @Path("/produtos")
2 class ProdutoResource {
3
4 @GET
5 @Path("/{id}/xml")
6 @Produces(MediaType.APPLICATION_XML)
7 public Produto getProdutoAsXML(@PathParam("id") int id) {
8 // implementacao
9 }
10
11 @GET
12 @Path("/{id}/json")
13 @Produces(MediaType.APPLICATION_JSON)
14 public Produto getProdutoAsJSON(@PathParam("id") int id) {
15 // implementacao
16 }
17 }
```

*Código Java 3.25: Produto.java*



## Consumindo XML ou JSON

Os recursos do JAXB também são utilizados para consumir XML ou JSON. Novamente suponha a seguinte entidade anotada com **@XMLElement**:

```

1 @XMLElement
2 class Produto {
3 private String nome;
4
5 private Double preco;
6
7 private Long id;
8
9 // GETTERS AND SETTERS
10 }
```

*Código Java 3.26: Produto.java*

Nos métodos da resource class, devemos aplicar a anotação **@Consumes** nos métodos.

```

1 @Path("/produtos")
2 public class ProdutoResource {
3 @POST
4 @Consumes(MediaType.APPLICATION_XML)
5 public void adiciona(Produto p) {
6 // implementacao
7 }
8 }
```

*Código Java 3.27: ProdutoResource.java*

```

1 @Path("/produtos")
2 public class ProdutoResource {
3 @POST
4 @Consumes(MediaType.APPLICATION_JSON)
5 public void adiciona(Produto p) {
6 // implementacao
7 }
8 }
```

*Código Java 3.28: ProdutoResource.java*

## Exercícios de Fixação

- 33 Crie um pacote chamado **br.com.k19.entities** no projeto **jax-rs**.
- 34 Adicione no pacote **br.com.k19.entities** uma classe para modelar produtos.

```

1 package br.com.k19.entities;
2
3 import javax.xml.bind.annotation.XmlRootElement;
4
5 @XmlRootElement
6 public class Produto {
7 private String nome;
8
9 private Double preco;
10
11 private Long id;
12
13 // GETTERS E SETTERS
14 }
```

*Código Java 3.29: Produto.java*

- 35 Adicione no pacote **br.com.k19.resources** uma classe para produzir produtos em XML e JSON.

```

1 package br.com.k19.resources;
2
3 import javax.ws.rs.GET;
4 import javax.ws.rs.Path;
5 import javax.ws.rs.PathParam;
6 import javax.ws.rs.Produces;
7 import javax.ws.rs.core.MediaType;
8
9 import br.com.k19.entities.Produto;
10
```

```

11 @Path("/produtos")
12 public class ProdutoResource {
13
14 @GET
15 @Path("/{id}/xml")
16 @Produces(MediaType.APPLICATION_XML)
17 public Produto getProdutoAsXML(@PathParam("id") long id) {
18 return this.geraProdutoFalso(id);
19 }
20
21 @GET
22 @Path("/{id}/json")
23 @Produces(MediaType.APPLICATION_JSON)
24 public Produto getProdutoAsJSON(@PathParam("id") long id) {
25 return this.geraProdutoFalso(id);
26 }
27
28 public Produto geraProdutoFalso(long id) {
29 Produto p = new Produto();
30 p.setNome("produto" + id);
31 p.setPreco(50.0 * id);
32 p.setId(id);
33
34 return p;
35 }
36 }
```

*Código Java 3.30: ProdutoResource.java*

**36** Execute a classe **Publicador**.

**37** Acesse as seguintes URIs para testar:

- <http://localhost:8080/produtos/1/xml>
- <http://localhost:8080/produtos/1/json>

**38** Adicione no pacote **br.com.k19.resources** uma classe converter produtos de XML para JSON e vice versa.

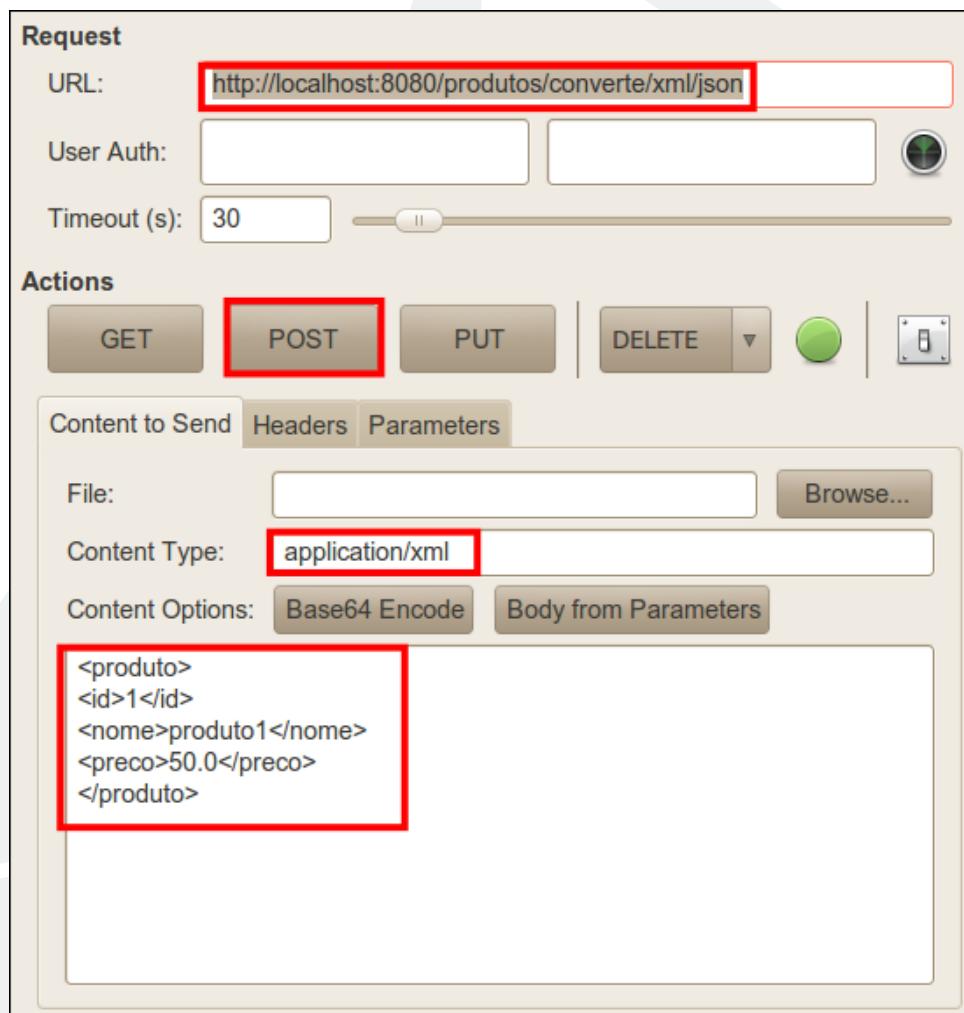
```

1 package br.com.k19.resources;
2
3 import javax.ws.rs.Consumes;
4 import javax.ws.rs.POST;
5 import javax.ws.rs.Path;
6 import javax.ws.rs.Produces;
7 import javax.ws.rs.core.MediaType;
8
9 import br.com.k19.entities.Produto;
10
11 @Path("/produtos/converte")
12 public class ConversorDeProdutoResource {
13
14 @POST
15 @Path("/json/xml")
16 @Consumes(MediaType.APPLICATION_JSON)
17 @Produces(MediaType.APPLICATION_XML)
18 public Produto transformToXML(Produto p) {
19 return p;
20 }
}
```

```
21
22 @POST
23 @Path("/xml/json")
24 @Consumes(MediaType.APPLICATION_XML)
25 @Produces(MediaType.APPLICATION_JSON)
26 public Produto transformToJSON(Produto p) {
27 return p;
28 }
29 }
```

Código Java 3.31: ConversorDeProdutoResource.java

- 39 Execute a classe **Publicador**.
- 40 Abra o firefox e depois o Poster (clique no canto inferior direito).
- 41 Faça uma requisição através do Poster como mostra a imagem abaixo:



- 42 Agora, faça outra requisição através do Poster como mostra a imagem abaixo:

**Request**

URL:  (highlighted)

User Auth:

Timeout (s):

**Actions**

|   |

Content to Send

File:

Content Type:  (highlighted)

Content Options:



## Implementando um Cliente

A partir da versão 2.0, a especificação JAX-RS define uma API para padronizar o desenvolvimento de clientes Java de web services restful.



### Exercícios de Fixação

- 43 Crie um pacote chamado **br.com.k19.client** no projeto **jax-rs**.
- 44 Adicione no pacote **client** uma classe para interagir com o resource de cotação de moeda.

```

1 package br.com.k19.client;
2
3 import javax.ws.rs.client.Client;
4 import javax.ws.rs.client.ClientBuilder;
5 import javax.ws.rs.client.WebTarget;
6 import javax.ws.rs.core.Response;
```

```

7
8 public class TesteCotacaoResource {
9 public static void main(String[] args) {
10 Client client = ClientBuilder.newClient();
11 WebTarget target = client.target("http://localhost:8080/Cotacao/DollarToReal");
12 Response response = target.request().get();
13 String cotacao = response.readEntity(String.class);
14 System.out.println(cotacao);
15 }
16 }
```

Código Java 3.32: TesteCotacaoResource.java

- 45** Adicione no pacote **br.com.k19.client** uma classe para interagir com o resource de produtos.

```

1 package br.com.k19.client;
2
3 import javax.ws.rs.client.Client;
4 import javax.ws.rs.client.ClientBuilder;
5 import javax.ws.rs.client.WebTarget;
6 import javax.ws.rs.core.Response;
7
8 import br.com.k19.entities.Produto;
9
10 public class TesteProdutoResource {
11 public static void main(String[] args) {
12 Client client = ClientBuilder.newClient();
13 WebTarget target = client.target("http://localhost:8080/produtos/1/xml");
14
15 Response response = target.request().get();
16 String xml = response.readEntity(String.class);
17 System.out.println(xml);
18
19 response = target.request().get();
20 Produto produto = response.readEntity(Produto.class);
21 System.out.println(produto.getId());
22 System.out.println(produto.getNome());
23 System.out.println(produto.getPreco());
24
25 target = client.target("http://localhost:8080/produtos/1/json");
26
27 response = target.request().get();
28 String json = response.readEntity(String.class);
29 System.out.println(json);
30
31 response = target.request().get();
32 produto = response.readEntity(Produto.class);
33 System.out.println(produto.getId());
34 System.out.println(produto.getNome());
35 System.out.println(produto.getPreco());
36 }
37 }
```

Código Java 3.33: TesteProdutoResource.java

- 46** Adicione no pacote **br.com.k19.client** uma classe para interagir com o resource de conversão de formato dos produtos.

```

1 package br.com.k19.client;
2
3 import javax.ws.rs.client.Client;
4 import javax.ws.rs.client.ClientBuilder;
5 import javax.ws.rs.client.Entity;
6 import javax.ws.rs.client.WebTarget;
```

```

7 import javax.ws.rs.core.Response;
8
9 import br.com.k19.entities.Produto;
10
11 public class TesteConversorDeProdutoResource {
12 public static void main(String[] args) {
13 Client client = ClientBuilder.newClient();
14
15 Produto p = new Produto();
16 p.setId(1L);
17 p.setNome("Bola");
18 p.setPreco(45.67);
19
20 WebTarget target = client.target("http://localhost:8080/produtos/converte/json/←
21 xml");
22 Response response = target.request().post(Entity.json(p));
23 String xml = response.readEntity(String.class);
24 System.out.println(xml);
25
26 target = client.target("http://localhost:8080/produtos/converte/xml/json");
27 response = target.request().post(Entity.xml(p));
28 String json = response.readEntity(String.class);
29 System.out.println(json);
30 }
30 }
```

Código Java 3.34: TesteConversorDeProdutoResource.java



## Exercícios Complementares

- Para consolidar os recursos da JAX-RS e do projeto Jersey, implemente um web service restful para funcionar como CRUD de clientes. Os dados podem ser mantidos apenas em memória.

Utilize os seguinte esquema de URIs e operações para os resources do seu web service:

| Resource                   | Método HTTP | Semântica           |
|----------------------------|-------------|---------------------|
| localhost:8080/clientes    | GET         | lista dos clientes  |
| localhost:8080/clientes    | POST        | adiciona um cliente |
| localhost:8080/clientes/id | PUT         | atualiza um cliente |
| localhost:8080/clientes/id | DELETE      | remove um cliente   |

Consulte o artigo sobre web services restful da K19.

<http://www.k19.com.br/artigos/criando-um-webservice-restful-em-java/>

- Implementes clientes através da API do projeto Jersey para testar o web service.