EBook Gratis

APRENDIZAJE java-ee

Free unaffiliated eBook created from **Stack Overflow contributors.**

#java-ee

Tabla de contenido

Acerca de

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

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

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

Capítulo 1: Empezando con java-ee

Observaciones

Esta sección proporciona una descripción general de qué es java-ee y por qué un desarrollador puede querer usarlo.

También debe mencionar cualquier tema grande dentro de java-ee, y vincular a los temas relacionados. Dado que la Documentación para java-ee es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

Examples

Instalación

En primer lugar, no puede "instalar" Java EE. Java EE consiste en una serie de especificaciones. Sin embargo, puede instalar implementaciones de esas especificaciones.

Dependiendo de sus necesidades, hay muchas posibilidades. Para instalar la mayoría (o todas) de las especificaciones, puede elegir un servidor de aplicaciones compatible con Java EE 7. Dependiendo de sus necesidades, puede elegir entre los servidores de aplicaciones que implementan el perfil web o los servidores de aplicaciones que implementan el perfil completo. Para obtener una lista de servidores de aplicaciones compatibles con Java EE7, consulte Compatibilidad con Java EE .

¿Qué es Java EE?

Java EE significa Java Enterprise Edition. Java EE extiende Java SE (que significa Java Standard Edition). Java EE es un conjunto de tecnologías y especificaciones relacionadas que están orientadas hacia el desarrollo de aplicaciones empresariales a gran escala. Java EE se desarrolla en un proceso impulsado por la comunidad. Hasta ahora se han lanzado las siguientes versiones de Java EE:

- J2EE 1.2 (12 de diciembre de 1999)
- J2EE 1.3 (24 de septiembre de 2001)
- J2EE 1.4 (11 de noviembre de 2003)
- Java EE 5 (11 de mayo de 2006)
- Java EE 6 (10 de diciembre de 2009)
- Java EE 7 (5 de abril de 2013)

Y se espera que Java EE 8 sea lanzado en el primer semestre de 2017.

Un concepto clave de Java EE es que cada versión de Java EE está compuesta por un conjunto de tecnologías específicas. Estas tecnologías abordan JSR específicos (solicitudes de especificación de Java). Para que un programador utilice estas tecnologías, necesita descargar

una implementación de las especificaciones de la tecnología Java EE. La comunidad Java proporciona una implementación de referencia para cada tecnología, pero otras tecnologías compatibles con Java EE se desarrollan y también se pueden usar. La comunidad proporciona un conjunto de pruebas, a saber, el Kit de compatibilidad de Java (JCK) que los desarrolladores de una implementación de JSR pueden usar para verificar si cumple o no con el JSR. La siguiente tabla proporciona una descripción general de las tecnologías que comprenden Java EE 7 y el JSR relacionado que define las especificaciones.

Tecnología Java EE 7	JSR
Plataforma Java, Enterprise Edition 7 (Java EE 7)	JSR 342
API de Java para WebSocket	JSR 356
API de Java para procesamiento JSON	JSR 353
Java Servlet 3.1	JSR 340
JavaServer Faces 2.2	JSR 344
Lenguaje de expresion 3.0	JSR 341
JavaServer Pages 2.3	JSR 245
Biblioteca de etiquetas estándar para JavaServer Pages (JSTL) 1.2	JSR 52
Aplicaciones por lotes para la plataforma Java	JSR 352
Utilidades concurrentes para Java EE 1.0	JSR 236
Contextos y inyección de dependencias para Java 1.1	JSR 346
Inyección de dependencia para Java 1.0	JSR 330
Validación de frijol 1.1	JSR 349
Enterprise JavaBeans 3.2	JSR

Tecnología Java EE 7	JSR
	345
Interceptores 1.2 (versión de mantenimiento)	JSR 318
Arquitectura del conector Java EE 1.7	JSR 322
Persistencia de Java 2.1	JSR 338
Anotaciones comunes para la plataforma Java 1.2	JSR 250
Java Message Service API 2.0	JSR 343
Java Transaction API (JTA) 1.2	JSR 907
JavaMail 1.5	JSR 919
API de Java para servicios web RESTful (JAX-RS) 2.0	JSR 339
Implementando Enterprise Web Services 1.3	JSR 109
API de Java para servicios web basados en XML (JAX-WS) 2.2	JSR 224
Metadatos de servicios web para la plataforma Java	JSR 181
API de Java para RPC basado en XML (JAX-RPC) 1.1 (opcional)	JSR 101
API de Java para mensajería XML 1.3	JSR 67
API de Java para registros XML (JAXR) 1.0	JSR 93
Interfaz del proveedor de servicios de autenticación de Java para contenedores 1.1	JSR 196
Contrato de Autorización de Java para Contenedores 1.5	JSR 115
Implementación de aplicaciones Java EE 1.2 (opcional)	JSR 88

Tecnología Java EE 7	JSR
J2EE Management 1.1	JSR 77
Soporte de depuración para otros idiomas 1.0	JSR 45
Arquitectura de Java para enlace XML (JAXB) 2.2	JSR 222
API de Java para procesamiento XML (JAXP) 1.3	JSR 206
Java Database Connectivity 4.0	JSR 221
Extensiones de administración de Java (JMX) 2.0	JSR 003
JavaBeans Activation Framework (JAF) 1.1	JSR 925
Streaming API para XML (StAX) 1.0	JSR 173

Instalación de Payara Server Full

Prerrequisitos

- JDK 1.7 o posterior instalado. Puedes encontrar el Oracle JDK's aquí.

Pasos

- Descargar Payara Server Full .
- Descomprima el servidor Payara en algún lugar de su computadora. Usaremos `C:\payara41` como INSTALL_DIR para usuarios de Windows y `/payara41` para `/payara41` de Linux / Mac.

Iniciar / detener Payara desde el símbolo del sistema

- Windows: abra un símbolo del sistema y ejecute el siguiente comando para iniciar / detener Payara:

```
"C:\payara41\bin\asadmin" start-domain

"C:\payara41\bin\asadmin" stop-domain
```

- Linux / Max: abra un terminal y ejecute el siguiente comando para iniciar / detener Payara:

```
/payara41/bin/asadmin start-domain

/payara41/bin/asadmin stop-domain
```

Comenzando Payara desde Netbeans

- Agregue el servidor de Payara a Netbeans (vea el capítulo anterior)
- Vaya a la pestaña 'Servicios' (Windows -> Servicios).
- Expanda el elemento 'Servidores'.
- Haga clic con el botón derecho en el servidor de Payara y seleccione 'Iniciar'.
- (Opcional) Haga clic con el botón derecho en el servidor de Payara y seleccione "Ver consola de administración de dominio" para ir a la consola.

Para verificar que está ejecutando el servidor de aplicaciones, abra un navegador y vaya a http: // localhost: 4848 para ver la consola de Payara Server.

Voila! ¡Ahora es el momento de crear su primera aplicación utilizando JavaEE y desplegarla en su servidor!

Construyendo mi primera aplicación JavaEE (Hello World)

Entendamos algo JavaEE consiste en una serie de especificaciones. Cuando instala un servidor de aplicaciones (Payara, por ejemplo), instala todas las especificaciones a la vez. Por ejemplo, hay una especificación para un ORM llamado **JPA** (Java Persistence API), una especificación para construir aplicaciones web *basadas en componentes* llamada **JSF** (Java Server Faces), una especificación para construir servicios web REST y clientes llamados **JAX-RS** .

Entonces, como puede adivinar, no hay solo una forma de crear una aplicación simple Hello World en JavaEE.

En segundo lugar, la especificación de JavaEE tiene una estructura específica de carpetas que se parece a esto (simplificado):

```
/projectname/src/main/java
/projectname/src/main/resources
/projectname/src/main/resources/META-INF
/projectname/src/main/webapp
/projectname/src/main/webapp/WEB-INF
```

Dentro de `/projectname/src/main/java` ponemos todas las clases de java que necesitamos.

Dentro de `/projectname/src/main/webapp` ponemos archivos html, css, javascript, etc.

Dentro de `/projectname/src/main/webapp/WEB-INF` hay algunos archivos de configuración opcionales, como *web.xml* y *beans.xml* .

Para simplificar, usaremos el IDE de NetBeans (es gratis) para crear nuestra primera aplicación JavaEE. Puedes encontrarlo aquí. Elige la versión de JavaEE e instálala.

Crear nuevo proyecto

- Open NetBeans IDE
- Vaya a Archivo> Nuevo proyecto> Maven> Aplicación web y haga clic en Siguiente.
- Cambie el **nombre del proyecto** a **HelloJavaEE** , luego haga clic en Siguiente y Finalizar.

Limpia y construye el proyecto.

- Vaya a Ejecutar> Limpiar y generar proyecto (HelloJavaEE).

Despliegue el archivo WAR

- En un navegador, vaya a http: // localhost: 4848 (siga las instrucciones para instalar el servidor payara).
- Vaya a Aplicaciones> Haga clic en Implementar, haga clic en Seleccionar archivo y elija su archivo war (HelloJavaEE-1.0-SNAPSHOT.war) en `../NetBeansProjects/HelloJavaEE/target` .

Despliega el archivo WAR directamente desde Netbeans

- Instale Payara (vea el siguiente capítulo).
- En Netbeans, vaya a la pestaña 'Servicios' (Ventana-> Servicios si no lo ve).
- Haga clic con el botón derecho en Servidores y seleccione 'Agregar servidor ...'
- Seleccione 'Servidor GlassFish', asígnele un nombre y haga clic en siguiente.
- Seleccione su instalación local de Payara, seleccione 'Dominio local' y haga clic en siguiente.
- Deje la configuración de la ubicación del dominio como está (Dominio: dominio1, Host: localhost, Puerto DAS: 4848, Puerto HTTP: 8080).
- Vaya a la pestaña 'Proyectos' (Windows -> Proyectos).
- Haga clic derecho en su proyecto y seleccione 'Propiedades'.
- En el panel de la izquierda, vaya a 'Ejecutar' y seleccione el servidor que acaba de agregar.
- (Opcional) Cambiar la ruta de contexto. Si configura la ruta del contexto en '/ MyFirstApplication', la URL predeterminada será ' http: // localhost: 8080 / MyFirstApplication '.

Ver resultados

- En un navegador, vaya a http: // localhost: 8080 / HelloJavaEE-1.0-SNAPSHOT

Voila! Esa es tu primera aplicación con tecnología JavaEE. Ahora deberías comenzar a crear otras aplicaciones "Hello World" usando diferentes especificaciones como JPA, EJB, JAX-RS, JavaBatch, etc.

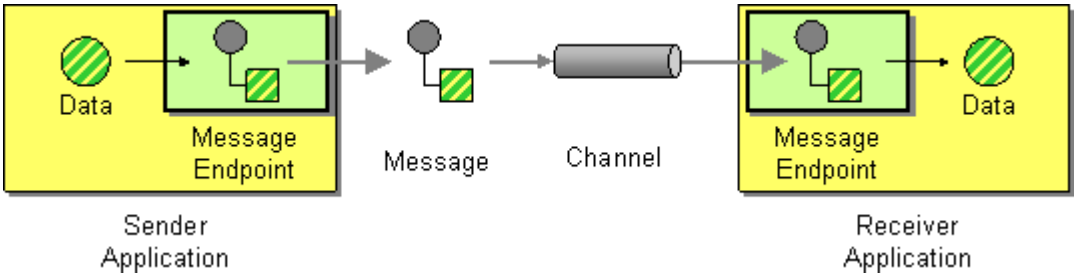
Lea Empezando con java-ee en línea: https://riptutorial.com/es/java-ee/topic/2265/empezando-con-java-ee

Capítulo 2: Arquitectura del conector de Java (JCA)

Observaciones

Aclaremos algunas terminologías primero:

- **La mensajería de salida** es donde el mensaje comienza desde el servidor (para ser más precisos, se inicia desde la aplicación que tiene en el servidor, `WebSphere Liberty` en este caso) y finaliza en el EIS.
- **La mensajería entrante** es donde el mensaje comienza desde el EIS y finaliza en el servidor.
- **Punto final del mensaje** en general, el lugar donde el mensaje termina sentado / recibiendo en una etapa específica de su ciclo de vida.



Así que con la conectividad de salida, nos referimos a la situación en la que una aplicación obtiene una conexión a un EIS externo y le lee o escribe datos. Con la conectividad de entrada, nos referimos a la situación en la que el Adaptador de recursos (RA) escucha los eventos del EIS externo y llama a su aplicación cuando ocurre tal evento.

Ilustración de un RA saliente

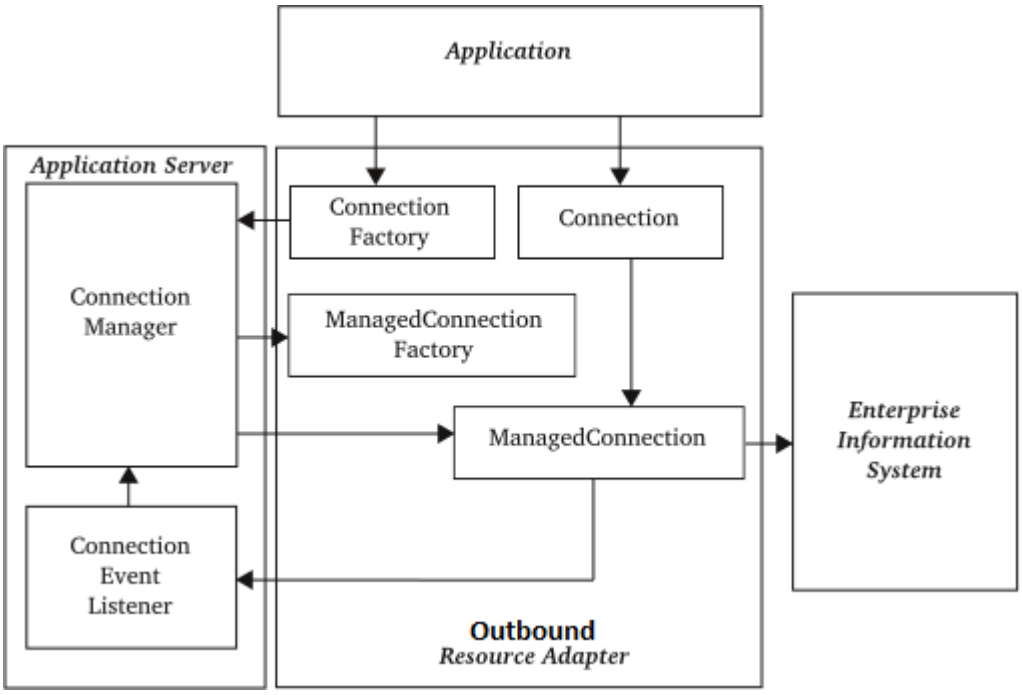
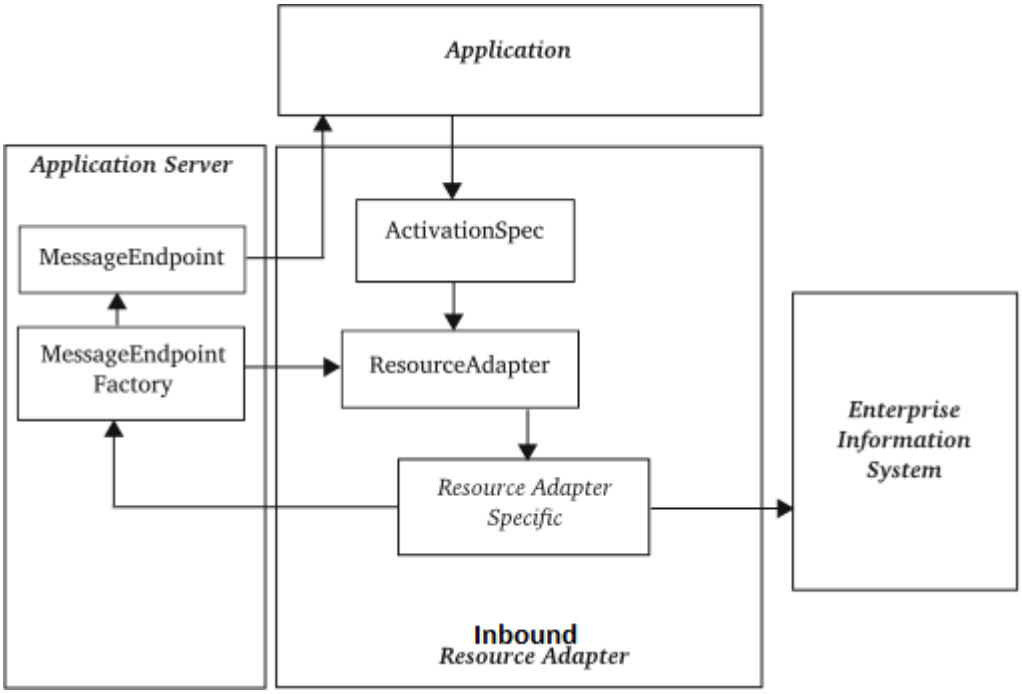


Ilustración de un RA entrante



¿Qué es un medio MessageEndPoint en JCA?

El servidor de aplicaciones (por ejemplo, `WebSphere Liberty`) proporciona MBeans de punto final de mensaje para ayudarlo a administrar la entrega de un mensaje a sus beans controlados por mensajes que actúan como escuchas en puntos finales específicos, que son destinos, y en la administración de los recursos EIS que son Utilizado por estos frijoles impulsados por mensajes. Los beans controlados por mensajes que se implementan como puntos finales de mensajes no son lo mismo que los beans controlados por mensajes que se configuran en un puerto de escucha. Los beans controlados por mensajes que se utilizan como puntos finales de mensajes deben implementarse utilizando una `ActivationSpecification` que se define dentro de una configuración RA para JCA (Se encuentra en el archivo `ra.xml`).

¿Qué significa activar un MessageEndPoint?

Con los MBeans de punto final de mensajes, puede activar y desactivar puntos finales específicos dentro de sus aplicaciones para asegurarse de que los mensajes se envíen solo a los frentes controlados por mensajes que interactúan con recursos de EIS en buen estado. Esta capacidad le permite optimizar el rendimiento de sus aplicaciones JMS en situaciones donde un recurso EIS no se comporta como se espera. La entrega de mensajes a un punto final generalmente falla cuando el bean controlado por mensajes que está escuchando invoca una operación contra un recurso que no está en buen estado. Por ejemplo, un proveedor de mensajería, que es un adaptador de recursos de entrada que cumple con JCA, puede fallar en la entrega de mensajes a un punto final cuando su bean subyacente impulsado por mensajes intenta confirmar transacciones contra un servidor de base de datos que no responde.

¿MessageEndPoint necesita ser un bean?

Debería. De lo contrario, terminará en un gran lío al crear su propia forma poco convencional de hacer cosas que cumplen con el propósito de seguir las especificaciones de Java EE en primer lugar. Diseñe sus beans controlados por mensajes para delegar el procesamiento empresarial a otros beans empresariales. No acceda a los recursos EIS directamente en el bean controlado por mensajes, sino hágalo indirectamente a través de un bean delegado.

¿Puede mostrar algún ejemplo simple sobre cómo trabajar / implementar un MessageEndPoint?

Revise el segundo recurso que menciono a continuación para un ejemplo útil.

Recursos de aprendizaje útiles:

- Gestionar mensajes con puntos finales de mensajes
- Desarrollar conectores de entrada

Examples

Ejemplo de adaptador de recursos

```
class MyResourceAdapter
   implements javax.resource.spi.ResourceAdapter {

   public void start(BootstrapContext ctx){..}
   public void stop(){..}

   public void endpointActivation (MessageEndpoingFactory mf, ActivationSpec a){..}
   public void endpointDeactivation (MessageEndpoingFactory mf, ActivationSpec a){..}
   public void getXAResources(ActivationSpec[] activationSpecs){..}
}
```

Lea Arquitectura del conector de Java (JCA) en línea: https://riptutorial.com/es/java-ee/topic/7309/arquitectura-del-conector-de-java--jca-

Capítulo 3: La API de Javamail

Observaciones

La página JavaMail en el sitio web de Oracle lo describe de la siguiente manera

> La API de JavaMail proporciona un marco independiente de la plataforma y del protocolo para crear aplicaciones de correo y mensajería. La API de JavaMail está disponible como un paquete opcional para usar con la plataforma Java SE y también se incluye en la plataforma Java EE.

El sitio principal para el proyecto JavaMail ahora está en java.net . Desde allí puede encontrar los javadocs para muchas versiones de las API, enlaces a los repositorios de código fuente, enlaces para descargas, ejemplos y sugerencias para usar JavaMail con algunos proveedores de servicios de correo electrónico populares.

Examples

Enviando un correo electrónico a un servidor de correo electrónico POP3

Este ejemplo muestra cómo establecer una conexión a un servidor de correo electrónico POP3 habilitado para SSL y enviar un correo electrónico simple (solo texto).

```
// Configure mail provider
Properties props = new Properties();
props.put("mail.smtp.host", "smtp.mymailprovider.com");
props.put("mail.pop3.host", "pop3.mymailprovider.com");
// Enable SSL
props.put("mail.pop3.ssl.enable", "true");
props.put("mail.smtp.starttls.enable", "true");

// Enable SMTP Authentication
props.put("mail.smtp.auth","true");

Authenticator auth = new PasswordAuthentication("user", "password");
Session session = Session.getDefaultInstance(props, auth);

// Get the store for authentication
final Store store;
try {
    store = session.getStore("pop3");
} catch (NoSuchProviderException e) {
    throw new IllegalStateException(e);
}

try {
    store.connect();
} catch (AuthenticationFailedException | MessagingException e) {
    throw new IllegalStateException(e);
}
```

```
    try {
      // Setting up the mail
      InternetAddress from = new InternetAddress("sender@example.com");
      InternetAddress to = new InternetAddress("receiver@example.com");

      MimeMessage message = new MimeMessage(session);
      message.setFrom(from);
      message.addRecipient(Message.RecipientType.TO, to);

      message.setSubject("Test Subject");
      message.setText("Hi, I'm a Mail sent with Java Mail API.");

      // Send the mail
      Transport.send(message);
    } catch (AddressException | MessagingException e)
        throw new IllegalStateException(e);
    }
```

Advertencias:

- Varios detalles han sido incorporados en el código anterior con fines ilustrativos.
- El manejo de excepciones NO es ejemplar. La `IllegalStateException` es una mala elección, para empezar.
- No se ha intentado manejar los *recursos* correctamente.

Enviar correo electrónico simple

```
public class GoogleMailTest {

    GoogleMailTest() {

    }

    public static void Send(final String username, final String password, String
recipientEmail, String title, String message) throws AddressException, MessagingException {
        GoogleMailTest.Send(username, password, recipientEmail, "", title, message);
    }

    public static void Send(final String username, final String password, String
recipientEmail, String ccEmail, String title, String message) throws AddressException,
MessagingException {
        Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
        final String SSL_FACTORY = "javax.net.ssl.SSLSocketFactory";
        // Get a Properties object
        Properties props = System.getProperties();
        props.setProperty("mail.smtps.host", "smtp.gmail.com");
        props.setProperty("mail.smtp.socketFactory.class", SSL_FACTORY);
        props.setProperty("mail.smtp.socketFactory.fallback", "false");
        props.setProperty("mail.smtp.port", "465");
        props.put("mail.debug", "true");
        props.setProperty("mail.smtp.socketFactory.port", "465");
        props.setProperty("mail.smtps.auth", "true");
        props.put("mail.smtps.quitwait", "false");
        Session session = Session.getInstance(props, null);
        // -- Create a new message --
        final MimeMessage msg = new MimeMessage(session);
        // -- Set the FROM and TO fields --
        msg.setFrom(new InternetAddress(username));
```

```
        msg.setRecipients(Message.RecipientType.TO, InternetAddress.parse(recipientEmail,
false));
        JOptionPane.showMessageDialog(null, msg.getSize());
        if (ccEmail.length() > 0) {
            msg.setRecipients(Message.RecipientType.CC, InternetAddress.parse(ccEmail,
false));
        }
        msg.setSubject(title);
        msg.setText(message);
        msg.setSentDate(new Date());
        SMTPTransport t = (SMTPTransport) session.getTransport("smtps");
        t.connect("smtp.gmail.com", username, password);
        t.sendMessage(msg, msg.getAllRecipients());
        t.close();
    }
    //    And use this code in any class, I'm using it in the same class in main method
    public static void main(String[] args) {
        String senderMail = "inzi769@gmail.com"; //sender mail id
        String password = "769inzimam-9771"; // sender mail password here
        String toMail = "inzi.rogrammer@gmail.com"; // recepient  mail id here
        String cc = ""; // cc mail id here
        String title = "Java mail test"; // Subject of the mail
        String msg = "Message here"; // message to be sent

        GoogleMailTest gmt = new GoogleMailTest();

        try {
            if (cc.isEmpty()) {
                GoogleMailTest.Send(senderMail, password, toMail, title, msg);
            } else {
                GoogleMailTest.Send(senderMail, password, toMail, cc, title, msg);
            }
        } catch (MessagingException ex) {
            Logger.getLogger(GoogleMailTest.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

}
```

Enviar correo HTML con formato

Puede usar el mismo ejemplo anterior **Enviar correo simple** con una pequeña modificación. Use `msg.setContent()` lugar de `msg.setText()` y use el tipo de contenido **html** como `text/html` .

Mira esto

```
msg.setContent(message, "text/html; charset=utf-8");
```

en lugar de

```
msg.setText(message);
```

Lea La API de Javamail en línea: https://riptutorial.com/es/java-ee/topic/8089/la-api-de-javamail

Capítulo 4: La API de WebSockets

Observaciones

WebSocket es un protocolo que permite la comunicación entre el cliente y el servidor / punto final mediante una única conexión TCP.

WebSocket está diseñado para implementarse en navegadores web y servidores web, pero puede ser utilizado por cualquier aplicación cliente o servidor.

Este tema sobre las API de Java para websockets fue desarrollado por JSR 356 e incorporado en las especificaciones de Java EE 7.

Examples

Creación de una comunicación WebSocket

WebSocket proporciona un protocolo de comunicación dúplex / bidireccional a través de una única conexión TCP.

- el cliente abre una conexión a un servidor que está escuchando una solicitud de WebSocket
- un cliente se conecta a un servidor utilizando un URI.
- Un servidor puede escuchar solicitudes de múltiples clientes.

Punto final del servidor

Puede crear un punto de acceso de servidor WebSocket simplemente anotando un POJO con `@ServerEndpoint` . `@OnMessage` decora un método que recibe mensajes entrantes. `@OnOpen` se puede usar para decorar un método que se llamará cuando se reciba una nueva conexión de un par. De manera similar, se llama a un método anotado con `@OnClose` cuando se cierra una conexión.

```
@ServerEndpoint("/websocket")
public class WebSocketServerEndpoint
{

    @OnOpen
    public void open(Session session) {
     System.out.println("a client connected");
    }

    @OnClose
    public void close(Session session) {
     System.out.println("a client disconnected");
    }

    @OnMessage
    public void handleMessage(String message) {
        System.out.println("received a message from a websocket client! " + message);
    }
```

```
}
```

Cliente Enpoint

Similar al punto final del servidor, puede crear un punto final de cliente WebSocket anotando un POJO con `@ClientEndpoint` .

```
@ClientEndpoint
public class WebsocketClientEndpoint {

    Session userSession = null;

    // in our case i.e. "ws://localhost:8080/myApp/websocket"
    public WebsocketClientEndpoint(URI endpointURI) {
        WebSocketContainer container = ContainerProvider.getWebSocketContainer();
        container.connectToServer(this, endpointURI);
    }

    @OnOpen
    public void onOpen(Session userSession) {
        System.out.println("opening websocket");
        this.userSession = userSession;
    }

    @OnClose
    public void onClose(Session userSession, CloseReason reason) {
        System.out.println("closing websocket");
        this.userSession = null;
    }

    @OnMessage
    public void onMessage(String message) {
        System.out.println("received message: "+ message);
    }

    public void sendMessage(String message) {
        System.out.println("sending message: "+ message);
        this.userSession.getAsyncRemote().sendText(message);
    }
}
```

Codificadores y decodificadores: WebSockets orientados a objetos

Gracias a los codificadores y decodificadores, el JSR 356 ofrece modelos de comunicación orientados a objetos.

Definición de mensajes

Asumamos que todos los mensajes recibidos deben ser transformados por el servidor antes de ser enviados de vuelta a todas las sesiones conectadas:

```
public abstract class AbstractMsg {
    public abstract void transform();
}
```

Supongamos ahora que el servidor administra dos tipos de mensajes: un mensaje basado en texto y un mensaje basado en enteros.

Los mensajes enteros multiplican el contenido por sí mismo.

```
public class IntegerMsg extends AbstractMsg {

    private Integer content;

    public IntegerMsg(int content) {
        this.content = content;
    }

    public Integer getContent() {
        return content;
    }

    public void setContent(Integer content) {
        this.content = content;
    }

    @Override
    public void transform() {
        this.content = this.content * this.content;
    }
}
```

Mensaje de cadena anteponer texto:

```
public class StringMsg extends AbstractMsg {

    private String content;

    public StringMsg(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    @Override
    public void transform() {
        this.content = "Someone said: " + this.content;
    }
}
```

Codificadores y Decodificadores

Hay un codificador por tipo de mensaje y un solo decodificador para todos los mensajes. Los codificadores deben implementar la `Encoder.XXX<Type>` cuando Decoder debe implementar `Decoder.XXX<Type>` .

La codificación es bastante sencilla: desde un mensaje, el método de `encode` debe generar una cadena con formato JSON. Aquí está el ejemplo para `IntegerMsg` .

```
public class IntegerMsgEncoder implements Encoder.Text<IntegerMsg> {

    @Override
    public String encode(IntegerMsg object) throws EncodeException {
        JsonObjectBuilder builder = Json.createObjectBuilder();

        builder.add("content", object.getContent());

        JsonObject jsonObject = builder.build();
        return jsonObject.toString();
    }

    @Override
    public void init(EndpointConfig config) {
        System.out.println("IntegerMsgEncoder initializing");
    }

    @Override
    public void destroy() {
        System.out.println("IntegerMsgEncoder closing");
    }
}
```

Codificación similar para la clase `StringMsg` . Obviamente, los codificadores se pueden factorizar a través de clases abstractas.

```
public class StringMsgEncoder implements Encoder.Text<StringMsg> {

    @Override
    public String encode(StringMsg object) throws EncodeException {
        JsonObjectBuilder builder = Json.createObjectBuilder();

        builder.add("content", object.getContent());

        JsonObject jsonObject = builder.build();
        return jsonObject.toString();
    }

    @Override
    public void init(EndpointConfig config) {
        System.out.println("StringMsgEncoder initializing");
    }

    @Override
    public void destroy() {
        System.out.println("StringMsgEncoder closing");
    }

}
```

El decodificador procede en dos pasos: verifica si el mensaje recibido se ajusta al formato exceptuado con `willDecode` y luego transforma el mensaje crudo recibido en un objeto con `decode` :

La clase pública MsgDecoder implementa Decoder.Text {

```
@Override
public AbstractMsg decode(String s) throws DecodeException {
    // Thanks to willDecode(s), one knows that
    // s is a valid JSON and has the attribute
    // "content"
    JsonObject json = Json.createReader(new StringReader(s)).readObject();
    JsonValue contentValue = json.get("content");

    // to know if it is a IntegerMsg or a StringMsg,
    // contentValue type has to be checked:
    switch (contentValue.getValueType()) {
        case STRING:
            String stringContent = json.getString("content");
            return new StringMsg(stringContent);
        case NUMBER:
            Integer intContent = json.getInt("content");
            return new IntegerMsg(intContent);
        default:
            return null;
    }

}

@Override
public boolean willDecode(String s) {

    // 1) Incoming message is a valid JSON object
    JsonObject json;
    try {
        json = Json.createReader(new StringReader(s)).readObject();
    }
    catch (JsonParsingException e) {
        // ...manage exception...
        return false;
    }
    catch (JsonException e) {
        // ...manage exception...
        return false;
    }

    // 2) Incoming message has required attributes
    boolean hasContent = json.containsKey("content");

    // ... proceed to additional test ...
    return hasContent;
}

@Override
public void init(EndpointConfig config) {
    System.out.println("Decoding incoming message...");
}

@Override
public void destroy() {
    System.out.println("Incoming message decoding finished");
}
```

}

ServerEndPoint

El Server EndPoint se parece bastante a la *comunicación de WebSocket* con tres diferencias principales:

1. La anotación ServerEndPoint tiene los atributos de los `encoders` y `decoders`
2. Los mensajes no se envían con `sendText` sino con `sendObject`
3. Se utiliza la anotación OnError. Si se produjo un error durante `willDecode` , se procesará aquí y la información del error se enviará al cliente

   @ServerEndpoint (value = "/ webSocketObjectEndPoint", decoders = {MsgDecoder.class}, encoders = {StringMsgEncoder.class, IntegerMsgEncoder.class}) clase pública ServerEndPoint {

```
@OnOpen
public void onOpen(Session session) {
    System.out.println("A session has joined");
}

@OnClose
public void onClose(Session session) {
    System.out.println("A session has left");
}

@OnMessage
public void onMessage(Session session, AbstractMsg message) {
    if (message instanceof IntegerMsg) {
        System.out.println("IntegerMsg received!");
    } else if (message instanceof StringMsg) {
        System.out.println("StringMsg received!");
    }

    message.transform();
    sendMessageToAllParties(session, message);
}

@OnError
public void onError(Session session, Throwable throwable) {
    session.getAsyncRemote().sendText(throwable.getLocalizedMessage());
}

private void sendMessageToAllParties(Session session, AbstractMsg message) {
    session.getOpenSessions().forEach(s -> {
        s.getAsyncRemote().sendObject(message);
    });
}
```

   }

Como era bastante detallado, aquí hay un cliente de JavaScript básico para aquellos que desean tener un ejemplo visual. Tenga en cuenta que este es un ejemplo tipo chat: todas las partes conectadas recibirán la respuesta.

```
<!DOCTYPE html>
<html>
    <head>
```

```
        <title>Websocket-object</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <!-- start of BAD PRACTICE! all style and script must go into a
             dedicated CSS / JavaScript file-->
        <style>
            body{
                background: dimgray;
            }

            .container{
                width: 100%;
                display: flex;
            }

            .left-side{
                width: 30%;
                padding: 2%;
                box-sizing:  border-box;
                margin: auto;
                margin-top: 0;
                background: antiquewhite;
            }
            .left-side table{
                width: 100%;
                border: 1px solid black;
                margin: 5px;
            }
            .left-side table td{
                padding: 2px;
                width: 50%;
            }
            .left-side table input{
                width: 100%;
                box-sizing: border-box;
            }

            .right-side{
                width: 70%;
                background: floralwhite;
            }
        </style>

        <script>
            var ws = null;
            window.onload = function () {
                // replace the 'websocket-object' with the
                // context root of your web application.
                ws = new WebSocket("ws://localhost:8080/websocket-
object/webSocketObjectEndPoint");
                ws.onopen = onOpen;
                ws.onclose = onClose;
                ws.onmessage = onMessage;
            };

            function onOpen() {
                printText("", "connected to server");
            }

            function onClose() {
                printText("", "disconnected from server");
            }
```

```
            function onMessage(event) {
                var msg = JSON.parse(event.data);
                printText("server", JSON.stringify(msg.content));
            }

            function sendNumberMessage() {
                var content = new Number(document.getElementById("inputNumber").value);
                var json = {content: content};
                ws.send(JSON.stringify(json));
                printText("client", JSON.stringify(json));
            }

            function sendTextMessage() {
                var content = document.getElementById("inputText").value;
                var json = {content: content};
                ws.send(JSON.stringify(json));
                printText("client", JSON.stringify(json));
            }

            function printText(sender, text) {
                var table = document.getElementById("outputTable");
                var row = table.insertRow(1);
                var cell1 = row.insertCell(0);
                var cell2 = row.insertCell(1);
                var cell3 = row.insertCell(2);

                switch (sender) {
                    case "client":
                        row.style.color = "orange";
                        break;
                    case "server":
                        row.style.color = "green";
                        break;
                    default:
                        row.style.color = "powderblue";
                }
                cell1.innerHTML = new Date().toISOString();
                cell2.innerHTML = sender;
                cell3.innerHTML = text;
            }
        </script>

        <!-- end of bad practice -->
    </head>
    <body>

        <div class="container">
            <div class="left-side">
                <table>
                    <tr>
                        <td>Enter a text</td>
                        <td><input id="inputText" type="text" /></td>
                    </tr>
                    <tr>
                        <td>Send as text</td>
                        <td><input type="submit" value="Send"
onclick="sendTextMessage();"/></td>
                    </tr>
                </table>
```

```
                <table>
                    <tr>
                        <td>Enter a number</td>
                        <td><input id="inputNumber" type="number" /></td>
                    </tr>
                    <tr>
                        <td>Send as number</td>
                        <td><input type="submit" value="Send"
onclick="sendNumberMessage();"/></td>
                    </tr>
                </table>
            </div>
            <div class="right-side">
                <table id="outputTable">
                    <tr>
                        <th>Date</th>
                        <th>Sender</th>
                        <th>Message</th>
                    </tr>
                </table>
            </div>
        </div>
    </body>
</html>
```

El código está completo y fue probado bajo Payara 4.1. El ejemplo es estándar puro (sin biblioteca / framework externo)

Lea La API de WebSockets en línea: https://riptutorial.com/es/java-ee/topic/7009/la-api-de-websockets

Capítulo 5: Servicio de mensajería de Java (JMS)

Introducción

Java Message Service es una API de Java que permite a las aplicaciones crear, enviar, recibir y leer mensajes. La API de JMS define un conjunto común de interfaces y semánticas asociadas que permiten que los programas escritos en el lenguaje de programación Java se comuniquen con otras implementaciones de mensajería. JMS permite la comunicación que no solo está acoplada de forma flexible, sino también asíncrona y confiable.

Observaciones

Java Message Service (JMS) es una API estándar de Java que permite a las aplicaciones crear, enviar, recibir y leer mensajes de forma asíncrona.

JMS define un conjunto general de interfaces y clases que permiten a las aplicaciones interactuar con otros proveedores de mensajes.

JMS es similar a JDBC: JDBC se conecta a diferentes bases de datos (Derby, MySQL, Oracle, DB2, etc.) y JMS se conecta con diferentes proveedores (OpenMQ, MQSeries, SonicMQ, etc.).

La implementación de referencia de JMS es Open Message Queue (OpenMQ). Es un proyecto de código abierto y se puede usar en aplicaciones independientes o se puede construir en un servidor de aplicaciones. Es el proveedor de JMS predeterminado integrado en GlassFish.

Examples

Creando ConnectionFactory

Las fábricas de conexiones son los objetos administrados que permiten que la aplicación se conecte al proveedor creando un objeto de `Connection` . `javax.jms.ConnectionFactory` es una interfaz que encapsula los parámetros de configuración definidos por un administrador.

Para usar `ConnectionFactory` cliente debe ejecutar la búsqueda JNDI (o usar la inyección). El siguiente código obtiene el objeto JNDI `InitialContext` y lo usa para buscar el objeto `ConnectionFactory` bajo el nombre JNDI:

```
Context ctx = new InitialContext();
ConnectionFactory connectionFactory =
                    (ConnectionFactory) ctx.lookup("jms/javaee7/ConnectionFactory");
```

Los métodos disponibles en esta interfaz son métodos `createConnection()` que devuelven un objeto `Connection` y nuevos métodos `createContext()` JMS 2.0 que devuelven un `JMSContext` .

Es posible crear una `Connection` o un `JMSContext` con la identidad de usuario predeterminada o especificando un nombre de usuario y contraseña:

```
public interface ConnectionFactory {
    Connection createConnection() throws JMSException;
    Connection createConnection(String userName, String password) throws JMSException;

    JMSContext createContext();
    JMSContext createContext(String userName, String password);
    JMSContext createContext(String userName, String password, int sessionMode);
    JMSContext createContext(int sessionMode);
}
```

Uso de la biblioteca ActiveMQ para mensajería (implementaciones específicas del proveedor de activemq jms)

Configurar ActiveMQ

- Descargue una distribución ActiveMQ de activemq.apache.org y descomprímala en algún lugar
- Puede iniciar el servidor inmediatamente, ejecutándose sin seguridad en localhost, usando el script bin / activemq
- Cuando se está ejecutando, puede acceder a la consola de su servidor local en http: // localhost: 8161 / admin /
- Configúralo modificando conf / activemq.xml
- Como el título sugiere los siguientes ejemplos, el usuario activemq jms implementaciones específicas del proveedor y, por lo tanto, activemq-all.jar debe agregarse a la ruta de clase.

Enviando un mensaje a través de un cliente independiente

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.Session;
import org.apache.activemq.ActiveMQConnectionFactory;

public class JmsClientMessageSender {

    public static void main(String[] args) {
        ConnectionFactory factory = new ActiveMQConnectionFactory("tcp://localhost:61616"); //
ActiveMQ-specific
        Connection con = null;
        try {
            con = factory.createConnection();
            Session session = con.createSession(false, Session.AUTO_ACKNOWLEDGE); // non-
transacted session

            Queue queue = session.createQueue("test.queue"); // only specifies queue name

            MessageProducer producer = session.createProducer(queue);
            Message msg = session.createTextMessage("hello queue"); // text message
```

```
            producer.send(msg);

        } catch (JMSException e) {
            e.printStackTrace();
        } finally {
            if (con != null) {
                try {
                    con.close(); // free all resources
                } catch (JMSException e) { /* Ignore */ }
            }
        }
    }
}
```

Encuestas para mensajes

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.TextMessage;
import org.apache.activemq.ActiveMQConnectionFactory;

public class JmsClientMessagePoller {

    public static void main(String[] args) {
        ConnectionFactory factory = new ActiveMQConnectionFactory("tcp://localhost:61616"); //
ActiveMQ-specific
        Connection con = null;

        try {
            con = factory.createConnection();
            Session session = con.createSession(false, Session.AUTO_ACKNOWLEDGE); // non-
transacted session

            Queue queue = session.createQueue("test.queue"); // only specifies queue name

            MessageConsumer consumer = session.createConsumer(queue);

            con.start(); // start the connection
            while (true) { // run forever
                Message msg = consumer.receive(); // blocking!
                if (!(msg instanceof TextMessage))
                    throw new RuntimeException("Expected a TextMessage");
                TextMessage tm = (TextMessage) msg;
                System.out.println(tm.getText()); // print message content
            }
        } catch (JMSException e) {
            e.printStackTrace();
        } finally {
            try {
                con.close();
            } catch (JMSException e) {/* Ignore */ }
        }
    }
}
```

Usando MessageListener

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.TextMessage;
import org.apache.activemq.ActiveMQConnectionFactory;

public class JmsClientMessageListener {

    public static void main(String[] args) {
        ConnectionFactory factory = new ActiveMQConnectionFactory("tcp://localhost:61616"); //
ActiveMQ-specific
        Connection con = null;

        try {
            con = factory.createConnection();
            Session session = con.createSession(false, Session.AUTO_ACKNOWLEDGE); // non-
transacted session
            Queue queue = session.createQueue("test.queue"); // only specifies queue name

            MessageConsumer consumer = session.createConsumer(queue);

            consumer.setMessageListener(new MessageListener() {
                public void onMessage(Message msg) {
                    try {
                        if (!(msg instanceof TextMessage))
                            throw new RuntimeException("no text message");
                        TextMessage tm = (TextMessage) msg;
                        System.out.println(tm.getText()); // print message
                    } catch (JMSException e) {
                        System.err.println("Error reading message");
                    }
                }
            });
            con.start(); // start the connection
            Thread.sleep(60 * 1000); // receive messages for 60s
        } catch (JMSException e1) {
            e1.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            try {
                con.close();        // free all resources
            } catch (JMSException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Uso de la búsqueda basada en jndi para mensajería (ejemplo no específico de la implementación)

Este método permite escribir y desplegar código no específico de la implementación en múltiples plataformas jms. El siguiente ejemplo básico se conecta al servidor jms de activemq y envía un mensaje.

```
import java.util.Properties;

import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class JmsClientJndi {

    public static void main(String[] args) {

        Properties jndiProps = new Properties();
        // Following two could be set via a system property for flexibility in the code.
        jndiProps.setProperty(Context.INITIAL_CONTEXT_FACTORY,
"org.apache.activemq.jndi.ActiveMQInitialContextFactory");
        jndiProps.setProperty(Context.PROVIDER_URL, "tcp://localhost:61616");

        QueueConnection conn = null;
        QueueSession session = null;
        QueueSender sender = null;
        InitialContext jndi = null;
        try {
            jndi = new InitialContext(jndiProps);
            QueueConnectionFactory factory = (QueueConnectionFactory)
jndi.lookup("ConnectionFactory");
            conn = factory.createQueueConnection();
            conn.start();

            session = conn.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
            Queue queue = (Queue) jndi.lookup("dynamicQueues/test.queue");
            sender = session.createSender(queue);

            TextMessage msg = session.createTextMessage();
            msg.setText("Hello worlds !!!!! ");
            sender.send(msg);


        } catch (NamingException e) {
            e.printStackTrace();
        } catch (JMSException e) {
            e.printStackTrace();
        } finally {
            try {
                if (sender != null)
                    sender.close();
                if (session != null)
                    session.close();
                if (conn != null)
                    conn.close();
```

```
            } catch (JMSException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Lea Servicio de mensajería de Java (JMS) en línea: https://riptutorial.com/es/java-ee/topic/7011/servicio-de-mensajeria-de-java--jms-

Capítulo 6: Servicios Web RESTful de Java (JAX-RS)

Observaciones

A diferencia de SOAP y la pila WS, que se especifican como estándares W3C, REST es realmente un conjunto de principios para diseñar y usar una interfaz basada en web. Las aplicaciones REST / RESTful dependen en gran medida de otras normas:

```
HTTP
URI, URL
XML, JSON, HTML, GIF, JPEG, and so forth (resource representations)
```

La función de JAX-RS (API de Java para servicios web RESTful) es proporcionar API que admitan la creación de servicios RESTful. Sin embargo, JAX-RS es solo *una forma de hacer esto* . Los servicios RESTful pueden implementarse de otras formas en Java y (de hecho) en muchos otros lenguajes de programación.

Examples

Recurso simple

En primer lugar, para una aplicación JAX-RS debe establecerse un URI base desde el cual todos los recursos estarán disponibles. Para ese propósito, la clase `javax.ws.rs.core.Application` debe extenderse y anotarse con la anotación `javax.ws.rs.ApplicationPath` . La anotación acepta un argumento de cadena que define el URI base.

```
@ApplicationPath(JaxRsActivator.ROOT_PATH)
public class JaxRsActivator extends Application {

    /**
     * JAX-RS root path.
     */
    public static final String ROOT_PATH = "/api";

}
```

Los recursos son clases de POJO simples que se anotan con la anotación `@Path` .

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("/hello")
public class HelloWorldResource {
    public static final String MESSAGE = "Hello StackOverflow!";
```

```
    @GET
    @Produces("text/plain")
    public String getHello() {
        return MESSAGE;
    }
}
```

Cuando se envía una solicitud `HTTP GET` a `/hello` , el recurso responde con un `Hello StackOverflow!` mensaje.

Tipos de métodos GET

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("/hello")
public class HelloWorldResource {
    public static final String MESSAGE = "Hello World!";

    @GET
    @Produces("text/plain")
    public String getHello() {
        return MESSAGE;
    }

    @GET
    @Path("/{letter}")
    @Produces("text/plain")
    public String getHelloLetter(@PathParam("letter") int letter){
        if (letter >= 0 && letter < MESSAGE.length()) {
            return MESSAGE.substring(letter, letter + 1);
        } else {
            return "";
        }
    }
}
```

`GET` sin un parámetro proporciona todo el contenido ("Hello World!") Y `GET` con el parámetro de ruta proporciona la letra específica de esa cadena.

Algunos ejemplos:

```
$ curl http://localhost/hello
Hello World!
```

```
$ curl http://localhost/hello/0
H
```

```
$ curl http://localhost/hello/4
o
```

Nota: si `@GET` la anotación de tipo de método (por ejemplo, la `@GET` anterior), un método de solicitud de forma predeterminada es un controlador de solicitud GET.

Método de eliminación

```
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Response;

@Path("hello")
public class HelloWorldResource {

    private String message = "Hello StackOverflow!";

    @GET
    @Produces("text/plain")
    public String getHello() {
        return message;
    }

    @DELETE
    public Response deleteMessage() {
        message = null;
        return Response.noContent().build();
    }
}
```

Consumirlo con rizo:

```
$ curl http://localhost/hello
Hello StackOverflow!

$ curl -X "DELETE" http://localhost/hello


$ curl http://localhost/hello
null
```

Método POST

```
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;

@Path("hello")
public class HelloWorldResource {
    @POST
    @Path("/receiveParams")
    public Response receiveHello(@FormParam("name") String name, @FormParam("message") String
message) {
        //process parameters
        return Response.status(200).build();
    }

    @POST
    @Path("/saveObject")
    @Consumes("application/json")
```

```
    public Response saveMessage(Message message) {
        //process message json
        return Response.status(200).entity("OK").build();
    }
}
```

El primer método puede invocarse a través del envío de formularios HTML mediante el envío de parámetros de entrada capturados. La acción de envío de formulario debe apuntar a:

```
/hello/receiveParams
```

El segundo método requiere el mensaje POJO con captadores / configuradores. Cualquier cliente REST puede llamar a este método con entrada JSON como:

```
{"sender":"someone","message":"Hello SO!"}
```

POJO debe tener la misma propiedad que JSON para hacer que la serialización funcione.

```
public class Message {

    String sender;
    String message;

    public String getSender() {
        return sender;
    }
    public void setSender(String sender) {
        this.sender = sender;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
}
```

Asignador de excepciones

```
@Provider
public class IllegalArgumentExceptionMapper implements
ExceptionMapper<IllegalArgumentException> {

  @Override
  public Response toResponse(IllegalArgumentException exception) {
    return Response.serverError().entity("Invalid input: " + exception.getMessage()).build();
  }
}
```

Este mapeador de excepciones capturará todas las IllegalArgumentExceptions lanzadas en la aplicación y le mostrará al usuario un mensaje claro en lugar de un seguimiento de pila.

UriInfo

Para obtener información sobre el URI que el agente de usuario utilizó para acceder a su recurso, puede usar la anotación del parámetro `@Context` con un parámetro `UriInfo` . El objeto `UriInfo` tiene algunos métodos que pueden usarse para obtener diferentes partes de la URI.

```
//server is running on https://localhost:8080,
// webapp is at /webapp, servlet at /webapp/servlet
@Path("class")
class Foo {

    @GET
    @Path("resource")
    @Produces(MediaType.TEXT_PLAIN)
    public Response getResource(@Context UriInfo uriInfo) {
        StringBuilder sb = new StringBuilder();
        sb.append("Path: " + uriInfo.getPath() + "\n");
        sb.append("Absolute Path: " + uriInfo.getAbsolutePath() + "\n");
        sb.append("Base URI: " + uriInfo.getBaseUri() + "\n");
        sb.append("Request URI: " + uriInfo.getRequestUri() + "\n");
        return Response.ok(sb.toString()).build();
    }
}
```

salida de un GET a `https://localhost:8080/webapp/servlet/class/resource` :

```
Path: class/resource
Absolute Path: https://localhost:8080/webapp/servlet/class/resource#
Base URI: https://localhost:8080/webapp/servlet/
Request URI: https://localhost:8080/webapp/servlet/class/resource
```

SubRecursos

A veces, por razones organizativas u otras, tiene sentido que su recurso de nivel superior devuelva un sub-recurso que se vea así. (Su sub-recurso no necesita ser una clase interna)

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("items")
public class ItemsResource {

    @Path("{id}")
    public String item(@PathParam("id") String id) {
        return new ItemSubResource(id);
    }

    public static class ItemSubResource {

        private final String id;

        public ItemSubResource(String id) {
            this.id = id;
        }

        @GET
        @Produces("text/plain")
```

```
        public Item item() {
            return "The item " + id;
        }
    }
}
```

Convertidores de parametros personalizados

Este es un ejemplo de cómo implementar convertidores de parámetros personalizados para puntos finales JAX-RS. El ejemplo muestra dos clases de la biblioteca `java.time` de Java 8.

```
@Provider
public class ParamConverters implements ParamConverterProvider {
  @Override
  public <T> ParamConverter<T> getConverter(Class<T> rawType,
                                            Type genericType,
                                            Annotation[] annotations)
  {
    if (rawType == LocalDate.class)
      return (ParamConverter<T>) new ParamConverter<LocalDate>() {
        @Override
        public LocalDate fromString(String value) {
          return LocalDate.parse(value);
        }

        @Override
        public String toString(LocalDate value) {
          return null;
        }
      };
    else if (rawType == MonthDay.class)
      return (ParamConverter<T>) new ParamConverter<MonthDay>() {
        @Override
        public MonthDay fromString(String value) {
          int[] ddmm = Arrays.stream(value.split("/"))
                             .mapToInt(Integer::parseInt)
                             .toArray();
          return MonthDay.of(ddmm[1], ddmm[0]);
        }

        @Override
        public String toString(MonthDay value) {
          return null;
        }
      };
    return null;
  }
}
```

Enlace de nombre

El enlace de nombres es un concepto que permite decir a un tiempo de ejecución de JAX-RS que un filtro o interceptor específico se ejecutará solo para un método de recurso específico. Cuando un filtro o un interceptor se limita solo a un método de recurso específico, decimos que está *vinculado a un nombre* . Los filtros e interceptores que no tienen dicha limitación se denominan *globales* .

Definición de una anotación de enlace de nombre

Los filtros o interceptores pueden asignarse a un método de recurso utilizando la anotación `@NameBinding` . Esta anotación se utiliza como meta anotación para otras anotaciones implementadas por el usuario que se aplican a proveedores y métodos de recursos. Vea el siguiente ejemplo:

```
@NameBinding
@Retention(RetentionPolicy.RUNTIME)
public @interface Compress {}
```

El ejemplo anterior define una nueva anotación de `@Compress` que es una anotación de enlace de nombre como se anota con `@NameBinding` . La anotación `@Compress` se puede utilizar para enlazar filtros e interceptor a puntos finales.

Unir un filtro o un interceptor a un punto final

Considere que tiene un interceptor que realiza la compresión GZIP y desea vincular dicho interceptor a un método de recursos. Para hacerlo, anote tanto el método de recursos como el interceptor, de la siguiente manera:

```
@Compress
public class GZIPWriterInterceptor implements WriterInterceptor {

    @Override
    public void aroundWriteTo(WriterInterceptorContext context)
                    throws IOException, WebApplicationException {
        final OutputStream outputStream = context.getOutputStream();
        context.setOutputStream(new GZIPOutputStream(outputStream));
        context.proceed();
    }
}
```

```
@Path("helloworld")
public class HelloWorldResource {

    @GET
    @Produces("text/plain")
    public String getHello() {
        return "Hello World!";
    }

    @GET
    @Path("too-much-data")
    @Compress
    public String getVeryLongString() {
        String str = ... // very long string
        return str;
```

```
    }
}
```

El `@Compress` se aplica en el método de recurso `getVeryLongString()` y en el interceptor `GZIPWriterInterceptor` . El interceptor se ejecutará solo si se ejecutará algún método de recurso con dicha anotación.

En el ejemplo anterior, el interceptor se ejecutará solo para el método `getVeryLongString()` . El interceptor no se ejecutará para el método `getHello()` . En este ejemplo, la razón es probablemente clara. Nos gustaría comprimir solo datos largos y no necesitamos comprimir la respuesta corta de `"Hello World!"` .

El enlace de nombres se puede aplicar a una clase de recurso. En el ejemplo, `HelloWorldResource` se `@Compress` con `@Compress` . Esto significaría que todos los métodos de recursos usarán la compresión en este caso.

Tenga en cuenta que los filtros globales se ejecutan siempre, incluso para los métodos de recursos que tienen anotaciones de enlace de nombre.

Documentación

- `@NameBinding` anotación `@NameBinding`
- Documentación de Jersey sobre filtros e interceptores.

Lea Servicios Web RESTful de Java (JAX-RS) en línea: https://riptutorial.com/es/java-ee/topic/7008/servicios-web-restful-de-java--jax-rs-

Creditos

S. No	Capítulos	Contributors
1	Empezando con java-ee	Barathon, Community, mylenereiners, Pablo Andrés Martínez Vargas, Spyros K
2	Arquitectura del conector de Java (JCA)	M. A. Kishawy
3	La API de Javamail	Inzimam Tariq IT, Johannes B, Stephen C, Stephen Leppik
4	La API de WebSockets	Adrian Krebs, Al1, Stephen C
5	Servicio de mensajería de Java (JMS)	DimaSan, Setu, Stephen C
6	Servicios Web RESTful de Java (JAX-RS)	Cassio Mazzochi Molin, Daniel Käfer, efreitora, Eric Finn, Jeff Hutchins, Jesse van Bekkum, justderb, kann, mico, RobAu, sargue, Stephen C