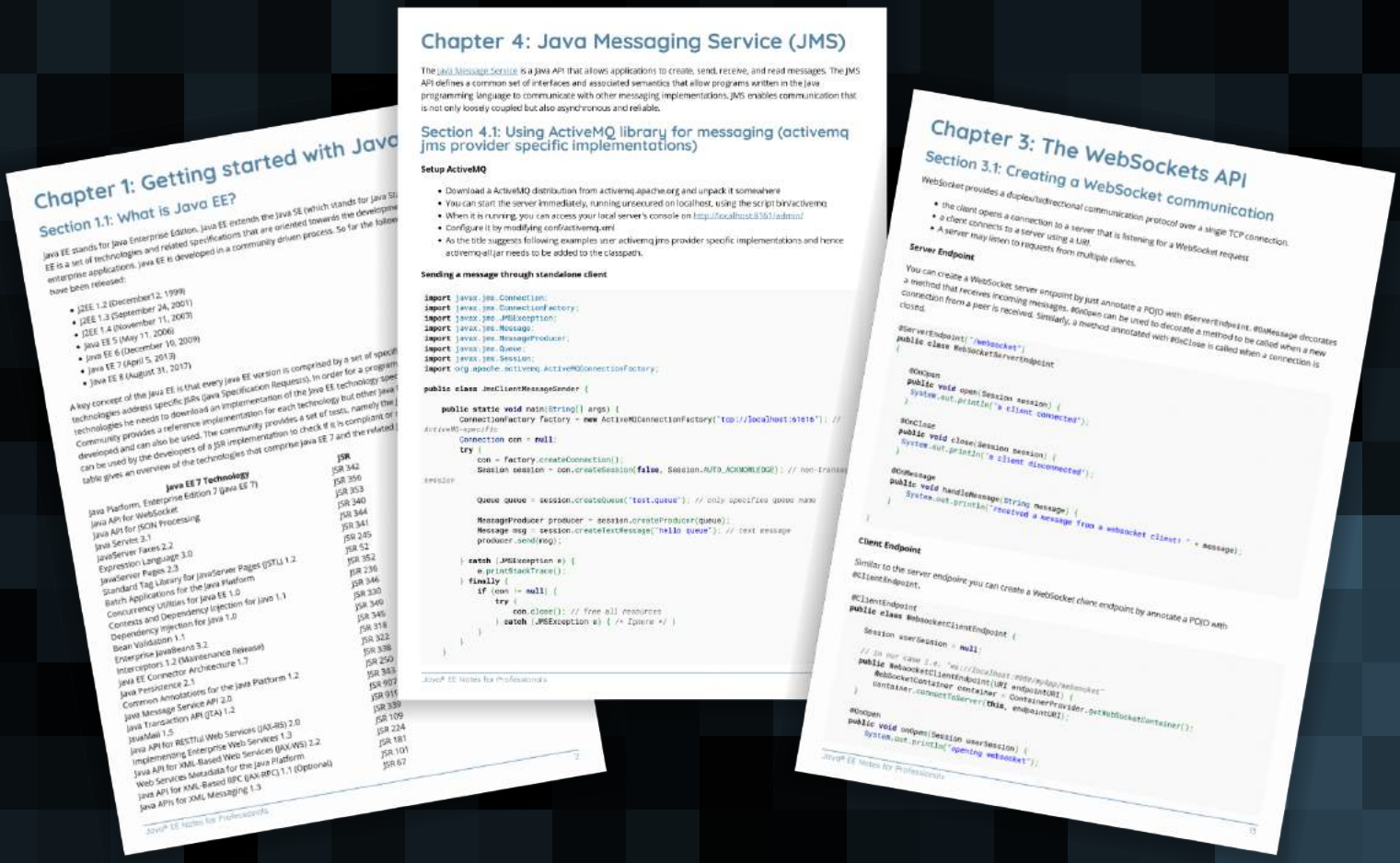
Java® EE
Notes for Professionals



20+ pages
of professional hints and tricks

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
http://GoalKicker.com/JavaEEBook

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with Java EE

Section 1.1: What is Java EE?

Java EE stands for Java Enterprise Edition. Java EE extends the Java SE (which stands for Java Standard Edition). Java EE is a set of technologies and related specifications that are oriented towards the development of large-scale enterprise applications. Java EE is developed in a community driven process. So far the following versions of Java EE have been released:

- J2EE 1.2 (December12, 1999)
- J2EE 1.3 (September 24, 2001)
- J2EE 1.4 (November 11, 2003)
- Java EE 5 (May 11, 2006)
- Java EE 6 (December 10, 2009)
- Java EE 7 (April 5, 2013)
- Java EE 8 (August 31, 2017)

A key concept of the Java EE is that every Java EE version is comprised by a set of specific technologies. These technologies address specific JSRs (Java Specification Requests). In order for a programmer to use these technologies he needs to download an implementation of the Java EE technology specifications. The Java Community provides a reference implementation for each technology but other Java EE compliant technologies are developed and can also be used. The community provides a set of tests, namely the Java Compatibility Kit (JCK) that can be used by the developers of a JSR implementation to check if it is compliant or not with the JSR. The following table gives an overview of the technologies that comprise Java EE 7 and the related JSR that define the specs.

Java EE 7 Technology	JSR
Java Platform, Enterprise Edition 7 (Java EE 7)	JSR 342
Java API for WebSocket	JSR 356
Java API for JSON Processing	JSR 353
Java Servlet 3.1	JSR 340
JavaServer Faces 2.2	JSR 344
Expression Language 3.0	JSR 341
JavaServer Pages 2.3	JSR 245
Standard Tag Library for JavaServer Pages (JSTL) 1.2	JSR 52
Batch Applications for the Java Platform	JSR 352
Concurrency Utilities for Java EE 1.0	JSR 236
Contexts and Dependency Injection for Java 1.1	JSR 346
Dependency Injection for Java 1.0	JSR 330
Bean Validation 1.1	JSR 349
Enterprise JavaBeans 3.2	JSR 345
Interceptors 1.2 (Maintenance Release)	JSR 318
Java EE Connector Architecture 1.7	JSR 322
Java Persistence 2.1	JSR 338
Common Annotations for the Java Platform 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
Java API for RESTful Web Services (JAX-RS) 2.0	JSR 339
Implementing Enterprise Web Services 1.3	JSR 109
Java API for XML-Based Web Services (JAX-WS) 2.2	JSR 224
Web Services Metadata for the Java Platform	JSR 181
Java API for XML-Based RPC (JAX-RPC) 1.1 (Optional)	JSR 101
Java APIs for XML Messaging 1.3	JSR 67

Java API for XML Registries (JAXR) 1.0	JSR 93
Java Authentication Service Provider Interface for Containers 1.1	JSR 196
Java Authorization Contract for Containers 1.5	JSR 115
Java EE Application Deployment 1.2 (Optional)	JSR 88
J2EE Management 1.1	JSR 77
Debugging Support for Other Languages 1.0	JSR 45
Java Architecture for XML Binding (JAXB) 2.2	JSR 222
Java API for XML Processing (JAXP) 1.3	JSR 206
Java Database Connectivity 4.0	JSR 221
Java Management Extensions (JMX) 2.0	JSR 003
JavaBeans Activation Framework (JAF) 1.1	JSR 925
Streaming API for XML (StAX) 1.0	JSR 173

Section 1.2: Installation

First of all, you cannot "install" Java EE. Java EE consists of a number of specifications. You can install implementations of those specifications however.

Depending on your needs, there are lots of possibilities. To install most (or all) of the specifications, you can choose a Java EE 7 compatible Application Server. Depending on your needs, you can choose between application servers that implement the web profile or application servers that implement the full profile. For a list of Java EE7 compatible application servers see Java EE Compatibility.

Section 1.3: Installing Payara Server Full

Prerequisites

- JDK 1.7 or later installed. You can find the Oracle JDK's here.

Steps

- Download Payara Server Full.
- Unzip the Payara Server at some location on your computer. We will use `C:\payara41` as INSTALL_DIR for Windows users and `/payara41` for Linux/Mac users.

Starting / stopping Payara from the command prompt

- Windows: Open a command prompt and execute the following command to start/stop Payara:

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

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

- Linux/Max: Open a terminal and execute the following command to start/stop Payara:

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

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

Starting Payara from Netbeans

- Add the Payara server to Netbeans (see previous chapter)
- Go to the 'Services' tab (Windows -> Services).

- Expand the 'Servers' item.
- Right-click on the Payara server and select 'Start'.
- (Optional) Right-click on the Payara server and select 'View Domain Admin Console' to go to the console.

To check that you're running the Application Server, open a browser and go to http://localhost:4848 to see the Payara Server Console.

Voila! Now it's time to build your first application using JavaEE and deploy it to your server!

Section 1.4: Building my First JavaEE Application (Hello World)

Let's understand something. JavaEE consists of a number of specifications. When you install an Application Server (Payara for example), you install all of the specifications at once. For example there's a specification for an ORM called **JPA** (Java Persistence API), a specification to build *Component Based* Web Applications called **JSF** (Java Server Faces), a specification to build REST web services and clients called **JAX-RS**.

So as you might guess, there's not only one way to build a simple Hello World application in JavaEE.

Secondly, the JavaEE spec has a specific structure of folders that looks something like this (simplified):

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

Inside the `/projectname/src/main/java` we put all the java classes that we need.

Inside the `/projectname/src/main/webapp` we put html files, css files, javascript files, etc.

Inside the `/projectname/src/main/webapp/WEB-INF`goes some optional configuration files, such as *web.xml* and *beans.xml*.

For simplicity we will use the NetBeans IDE (it's free) to build our first JavaEE Application. You can find it here. Choose the JavaEE version and install it.

Create new project

- Open NetBeans IDE
- Go to File > New Project > Maven > Web Application and click Next.
- Change **project name** to **HelloJavaEE**, then click Next and Finish.

Clean and build the project

- Go to Run > Clean and Build Project (HelloJavaEE).

Deploy the WAR file

- In a browser, go to http://localhost:4848 (follow instructions to install payara server).
- Go to Applications > Click Deploy, Click on Select File and choose your war file (HelloJavaEE-1.0-SNAPSHOT.war) under `../NetBeansProjects/HelloJavaEE/target`.

Deploy the WAR file directly from Netbeans

- Install Payara (see next chapter).
- In Netbeans, go to the 'Services' tab (Window-> Services if you don't see it).

- Right-click on Servers and select 'Add Server...'
- Select 'GlassFish Server', give it a name and click next.
- Point to your local Payara installation, select 'Local Domain' and click next.
- Leave the domain location settings as they are (Domain: domain1, Host: localhost, DAS Port: 4848, HTTP Port: 8080).
- Go to the 'Projects' tab (Windows -> Projects).
- Right-click on your project and select 'Properties'.
- In the left hand pane, go to 'Run' and select the Server you just added.
- (Optional) Change the context path. If you set the context path to '/MyFirstApplication' the default URL will be 'http://localhost:8080/MyFirstApplication'.

View results

- In a browser, go to http://localhost:8080/HelloJavaEE-1.0-SNAPSHOT

Voila! That's your first app using JavaEE technology. You should now start creating other "Hello World" apps using different specifications like JPA, EJB, JAX-RS, JavaBatch, etc...

Chapter 2: Java RESTful Web Services (JAX-RS)

Section 2.1: Simple Resource

First of all for a JAX-RS application must be set a base URI from which all the resources will be available. For that purpose the `javax.ws.rs.core.Application` class must be extended and annotated with the `javax.ws.rs.ApplicationPath` annotation. The annotation accepts a string argument which defines the base URI.

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

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

}
```

Resources are simple POJO classes which are annotated with the `@Path` annotation.

```
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;
    }
}
```

When a `HTTP GET` request is sent to `/hello`, the resource responds with a `Hello StackOverflow!` message.

Section 2.2: GET method types

```
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` without a parameter gives all content ("Hello World!") and `GET` with path parameter gives the specific letter out of that String.

Some examples:

$ curl http://localhost/hello Hello World! $ curl http://localhost/hello/0 H $ curl http://localhost/hello/4 o

Note: if you leave out the method-type annotation (e.g. the `@GET` above), a request method defaults to being a GET request handler.

Section 2.3: POST Method

```
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();
    }
}
```

First method can be invoked through HTML form submission by sending captured input parameters. Form submit action should point to -

```
/hello/receiveParams
```

Second method requires Message POJO with getters/setters. Any REST client can call this method with JSON input as -

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

POJO should have the same property as JSON to make serialization work.

```
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;
    }
}
```

Section 2.4: Name binding

Name binding is a concept that allows to say to a JAX-RS runtime that a specific filter or interceptor will be executed only for a specific resource method. When a filter or an interceptor is limited only to a specific resource method we say that it is *name-bound*. Filters and interceptors that do not have such a limitation are called *global*.

Defining a name binding annotation

Filters or interceptors can be assigned to a resource method using the `@NameBinding` annotation. This annotation is used as meta annotation for other user implemented annotations that are applied to a providers and resource methods. See the following example:

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

The example above defines a new `@Compress` annotation which is a name binding annotation as it is annotated with `@NameBinding`. The `@Compress` annotation can be used to bind filters and interceptor to endpoints.

Binding a filter or interceptor to an endpoint

Consider you have an interceptor that performs GZIP compression and you want to bind such interceptor to a resource method. To do it, annotate both the resource method and the interceptor, as following:

```
@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;
    }
}
```

The `@Compress` is applied on the resource method `getVeryLongString()` and on the interceptor `GZIPWriterInterceptor`. The interceptor will be executed only if any resource method with such a annotation will be executed.

In above example, the interceptor will be executed only for the `getVeryLongString()` method. The interceptor will not be executed for method `getHello()`. In this example the reason is probably clear. We would like to compress only long data and we do not need to compress the short response of `"Hello World!"`.

Name binding can be applied on a resource class. In the example `HelloWorldResource` would be annotated with `@Compress`. This would mean that all resource methods will use compression in this case.

Note that global filters are executed always, so even for resource methods which have any name binding annotations.

Documentation

- `@NameBinding` annotation documentation
- Jersey documentation about filters and interceptors

Section 2.5: Exception Mapper

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

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

This exception mapper will catch all IllegalArgumentExceptions thrown in the application, and show the user a clear message instead of a stacktrace.

Section 2.6: DELETE method

```
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();
    }
}
```

Consume it with curl:

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

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


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

Section 2.7: Custom parameter converters

This is an example of how to implement custom parameter converters for JAX-RS endpoints. The example shows two classes from Java 8's `java.time` library.

```
@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;
  }
}
```

Section 2.8: SubResources

Sometimes for organizational or other reasons it makes sense to have your top level resource return a sub-resource that would look like this. (Your sub-resource does not need to be an inner class)

```
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;
        }
    }
}
```

Section 2.9: UriInfo

In order to get information about the URI the user agent used to access your resource, you can use the `@Context` parameter annotation with a `UriInfo` parameter. The `UriInfo` object has a few methods that can be used to get different parts of the 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();
    }
}
```

output of a GET to `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
```

Chapter 3: The WebSockets API

Section 3.1: Creating a WebSocket communication

WebSocket provides a duplex/bidirectional communication protocol over a single TCP connection.

- the client opens a connection to a server that is listening for a WebSocket request
- a client connects to a server using a URI.
- A server may listen to requests from multiple clients.

Server Endpoint

You can create a WebSocket server entpoint by just annotate a POJO with `@ServerEndpoint`. `@OnMessage` decorates a method that receives incoming messages. `@OnOpen` can be used to decorate a method to be called when a new connection from a peer is received. Similarly, a method annotated with `@OnClose` is called when a connection is closed.

```
@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);
    }

}
```

Client Endpoint

Similar to the server endpoint you can create a WebSocket client endpoint by annotate a POJO with `@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);
    }
}
```

Section 3.2: Encoders and Decoder: Object-Oriented WebSockets

Thanks to encoders and decoders, the JSR 356 offers a object oriented communication models.

Messages definition

Let's assume all received messages have to be transformed by the server before being sent back to all connected sessions:

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

Let's now assume that the server manage two message types: a text-based message and an integer-based message.

Integer messages multiply the content by itself.

```
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;
    }
```

```
}
```

String message prepend some text:

```
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;
    }
}
```

Encoders and Decoder

There is one encoder per message type and a single decoder for all messages. Encoders must implements `Encoder.XXX<Type>` interface when Decoder must implements `Decoder.XXX<Type>`.

Encoding is fairly straightforward: from a message, the `encode` method must output a JSON formatted String. Here is the example for `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");
    }
}
```

Similar encoding for `StringMsg` class. Obviously, encoders can be factorized via abstract classes.

```
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");
    }

}
```

Decoder proceeds in two steps: checking if the received message fits the excepted format with `willDecode` and then transform the received raw message into a object with `decode`:

public class MsgDecoder implements 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

The Server EndPoint pretty looks like the *WebSocket communication* with three main differences:

1. ServerEndPoint annotation has the `encoders` and `decoders` attributes
2. Messages are not sent with `sendText` but with `sendObject`
3. OnError annotation is used. If there was an error thrown during `willDecode`, it will be processed here and error information is sent back to the client

   @ServerEndpoint(value = "/webSocketObjectEndPoint", decoders = {MsgDecoder.class}, encoders = {StringMsgEncoder.class, IntegerMsgEncoder.class}) public class 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);
        });
    }

}
```

As I was quite verbose, here is a basic JavaScript client for those who want to have a visual example. Please note that this is a chat-like example: all the connected parties will received the answer.

```
<!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>
```

Code is complete and was tested under Payara 4.1. Example is pure standard (no external library/framework)

Chapter 4: Java Messaging Service (JMS)

The Java Message Service is a Java API that allows applications to create, send, receive, and read messages. The JMS API defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations. JMS enables communication that is not only loosely coupled but also asynchronous and reliable.

Section 4.1: Using ActiveMQ library for messaging (activemq jms provider specific implementations)

Setup ActiveMQ

- Download a ActiveMQ distribution from activemq.apache.org and unpack it somewhere
- You can start the server immediately, running unsecured on localhost, using the script bin/activemq
- When it is running, you can access your local server's console on http://localhost:8161/admin/
- Configure it by modifying conf/activemq.xml
- As the title suggests following examples user activemq jms provider specific implementations and hence activemq-all.jar needs to be added to the classpath.

Sending a message through standalone client

```
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 */ }
            }
        }
    }
```

```
}
```

Polling for messages

```
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 */ }
        }
    }
}
```

Using 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();
            }
        }
    }
}
```

Section 4.2: Creating ConnectionFactory

Connection factories are the managed objects that allows application to connect to provider by creating `Connection` object. `javax.jms.ConnectionFactory` is an interface that encapsulates configuration parameters defined by an administrator.

For using `ConnectionFactory` client must execute JNDI lookup (or use injection). The following code gets JNDI `InitialContext` object and uses it to lookup for `ConnectionFactory` object under JNDI name:

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

The methods available in this interface are `createConnection()` methods that return a `Connection` object and new JMS 2.0 `createContext()` methods that return a `JMSContext`.

It's possible to create a `Connection` or a `JMSContext` either with the default user identity or by specifying a username and password:

```
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);
}
```

Section 4.3: Using jndi based lookup for messaging (Non-implementation-specific example)

This method allows non-implementation-specific code to be written and deployed across multiple jms platforms. Below basic example connects to activemq jms server and sends a message.

```
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();
            }
        }
    }
}
```

Chapter 5: Java Connector Architecture (JCA)

Section 5.1: Example Resource Adapter

```
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){..}
}
```

Chapter 6: The Javamail API

Section 6.1: Send HTML Formatted Mail

You can use the same Example above **Send Simple Mail** with a little modification. Use `msg.setContent()` instead of `msg.setText()` and use content type **html** as `text/html`.

check this

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

instead of

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

Section 6.2: Send Simple Email

```
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);
        }
    }

}
```

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

Adrian Krebs	Chapter 3
Al1	Chapter 3
Barathon	Chapter 1
Cassio Mazzochi Molin	Chapter 2
Daniel Käfer	Chapter 2
DimaSan	Chapter 4
efreitora	Chapter 2
Eric Finn	Chapter 2
Inzimam Tariq IT	Chapter 6
Jeff Hutchins	Chapter 2
Jesse van Bekkum	Chapter 2
Johannes B	Chapter 6
justderb	Chapter 2
kann	Chapter 2
M. A. Kishawy	Chapter 5
mico	Chapter 2
mylenereiners	Chapter 1
Pablo Andrés Martínez Vargas	Chapter 1
RobAu	Chapter 2
sargue	Chapter 2
Setu	Chapter 4
Spyros K	Chapter 1
Stephen C	Chapters 2, 3, 4 and 6

You may also like