Modern Web Development for Java Programmers

Unit 7. Java EE 7: overview of the new features. Creating the Java EE version of the server-side auction (JAX-RS, CDI). Intro to WebSockets. Pushing the auction data to the client using WebSocket.

Prerequisites: The Wildfly 8 server is installed and configured to run inside IntelliJ IDEA as shown in this video: https://vimeo.com/91668238

Unit 7 Timeline

- Java EE 7 Overview 15 min
- JSON Processing 15 min
  - Walkthrough 1 15 min
- Using REST API 10 min
  - Walkthrough 2 15 min
- CDI 10 min
- Break 10 min
- Building RESTful service skeleton
  - Walkthrough 3 20 min
- WebSocket 25 min
  - Walkthrough 4 20 min

Java EE 7

Overview

Java EE Highlights

- Released in June of 2013.
- The main improvements are in the Web development and HTML5
- JAX-RS 2.0
- JSON processing
- WebSocket support

GlassFish 4 and WildFly 8 support Java EE 7
18 servers support Java EE 6.

New and Updated JSRs

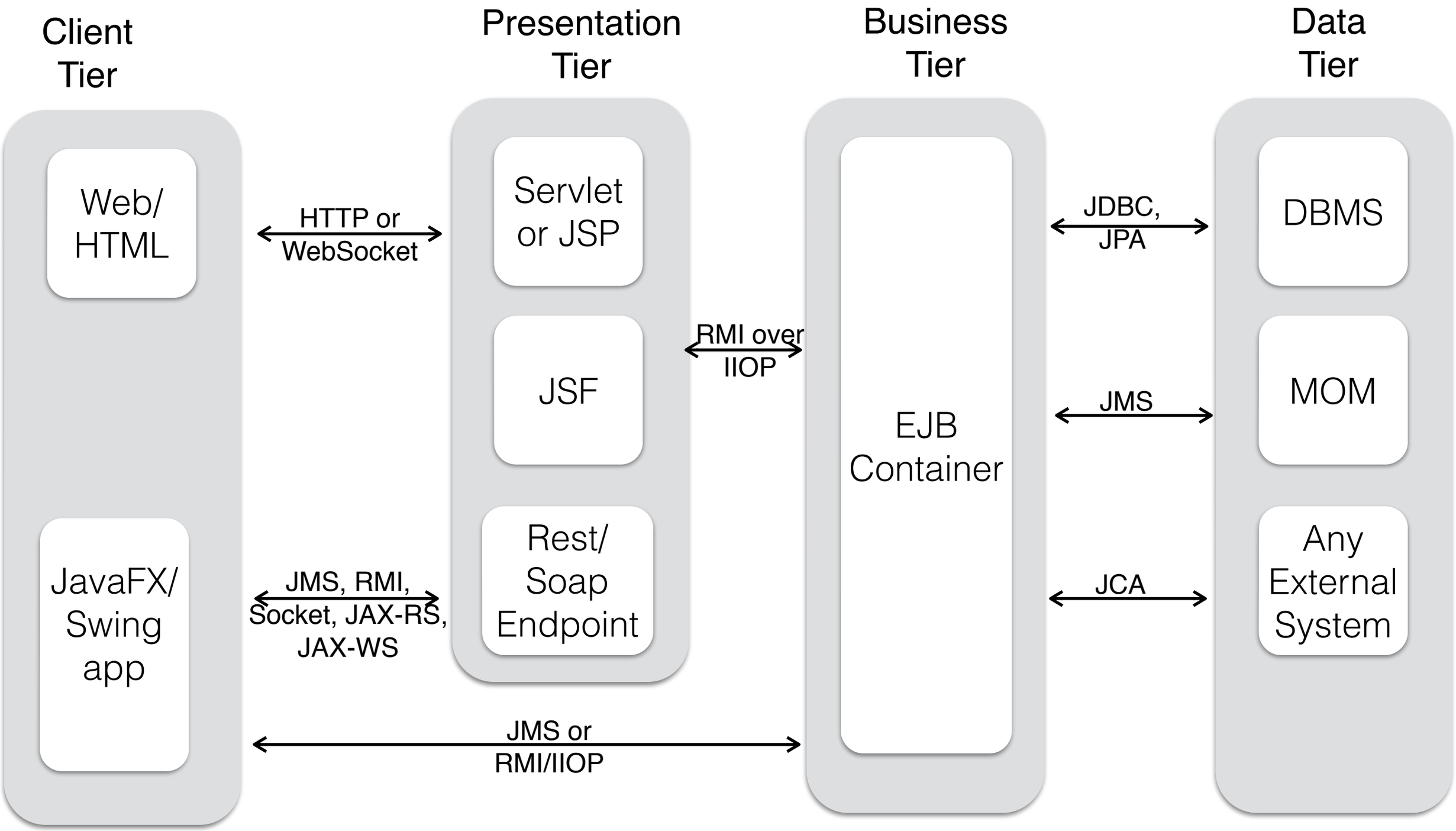
- JSR 236: Concurrency Utilities for Java EE 1.0
- JSR 338: Java Persistence API 2.1
- JSR 339: Java API for RESTful Web Services 2.0
- JSR 340: Java Servlet 3.1
- JSR 341: Expression Language 3.0
- JSR 343: Java Message Service 2.0
- JSR 344: JavaServer Faces 2.2
- JSR 345: Enterprise JavaBeans 3.2
- JSR 346: Contexts and Dependency Injection for Java EE 1.1
- JSR 349: Bean Validation 1.1
- JSR 352: Batch Applications for the Java Platform 1.0
- JSR 353: Java API for JSON Processing 1.0
- JSR 356: Java API for WebSocket 1.0

Let's talk blue

Selected Features Briefly

- You can use threads in Java EE containers (`ManagedExecutorService, ManagedThreadFactory`, and more).
- JMS 2.0 has simplified API
- Servlets 3.1 has non-blocking I/O (ReadListener, WriteListener)
- HttpServletRequest has a new method upgrade()
- JSON Processing API

Architecting Java EE Applications



JSON Processing (JSR 353)

The package javax.json includes classes supporting two ways of producing JSON from Java:

1. Using Object Model API. It creates a tree representing JSON data in memory

2. Using Streaming API is event driven. It stops for processing when an object begins or ends, when if finds a key or a value. It generates output into a given stream, e.g into a file.

JSON Processing Tutorial: http://docs.oracle.com/javaee/7/tutorial/doc/jsonp.htm#GLRBB

Reading JSON with Streaming API

```
JsonParser parser = Json.createParser(new StringReader(jsonData));
while (parser.hasNext()) {
    JsonParser.Event event = parser.next();
    switch(event) {
        case START_ARRAY:
        case END_ARRAY:
        case START_OBJECT:
        case END_OBJECT:
        case VALUE_FALSE:
        case VALUE_NULL:
        case VALUE_TRUE:
            System.out.println(event.toString());
            break;
        case KEY_NAME:
            System.out.print(event.toString() + " " +
                    parser.getString() + " - ");
            break;
        case VALUE_STRING:
        case VALUE_NUMBER:
            System.out.println(event.toString() + " " +
                    parser.getString());
            break;
    }
}
```

Walkthrough 1

- Import the project JSONSample into IDEA.

- Add the external library javax.json-1.0.3.jar from your wildly installation (it's in wildfly-8.0.0.Final/modules/system/layers/base/org/glassfish/javax/json/main).

- Run the class JavaToJSONStreaming. Check the content of the newly created file product_as_stream.json.

- Run the class JavaToJSONObject. Check the content of the newly created file product_as_object.json.

REST and JAX-RS

REpresentational State of Transfer

REST Principles (Roy Fielding)

- Every resource on the Web has an ID (URI)
- Use uniform interface: HTTP **Get, Post, Put, Delete.** Separation of concerns.
- A resource can have multiple representations (text, JSON, XML, PDF, etc.)
- Requests are stateless – no client-specific info is stored between requests
- You can link one resource to another(s)
- Resources should be cacheable
- A REST app can be layered

HTTP Methods

- GET Safe, Idempotent, cacheable
- PUT Idempotent
- DELETE Idempotent
- HEAD Safe, Idempotent
- POST None of the above

GET is for retrieval, POST for inserts, PUT – updates, DELETE - removal.

Idempotent: regardless of how many times a given method is invoked, the end result is the same.

JAX RS 2.0 (JSR 339)

- Rest endpoint - a POJO, typically deployed inside WAR
- Has Client API
- Message Filters and Entity Interceptors (e.g. Login Filter, encryptions et al.)
- Async processing on both client and server
- Validation

Besides app servers, Jersey framework implements JAX-RS 2.0 https://jersey.java.net/

Selected JAX-RS Annotations

- @ApplicationPath - defines the URL mapping for the application packaged in a war. It's the base URI for all @Path annotations.
- @Path - a root resource class (POJO),that has at least one method annotated with @Path.
- @PathParam - injects values from request into a method parameter (e.g. Product ID)
- @GET - the class method that handles HTTP Get. You can have multiple methods annotated with @GET, and each produces different MIME type.
- @POST - the class method that handles HTTP Post
- @Put - the class method that handles HTTP Put
- @Delete - the class method that handles HTTP Delete
- @Produces - specifies the MIME type for response (e.g. "application/json"). The client's Accept header of the HTTP request declares what's acceptable. The client gets 406 if no methods that produce required is found.
- @Consumes - specifies the MIME types that a resource can consume when sent by the client. If a resource is unable to consume the requested MIME type, the clients get HTTP error 415.
- @QueryParam - if a request URL has parameters, each param will be placed in the provided Java variable.

Naming Rest Resources

Don't include verbs in resource names, e.g. **getProducts** is bad. Just name it **Product**. The HTTP methods Get/Put/Post/Delete clearly shows what the client wants.

```
// The endpoint URL path
@ApplicationPath(“webresources")
public class MyApplication extends Application {
}
```

```
// The endpoint resource to handle products
@Path("/product")
public class ProductService {


  // The method to handle HTTP Get requests
  @GET
  @Path("{name}")
  public String get(@PathParam("name")String payload) {
    System.out.println("in get method of ProductService");
    return Database.get(payload);
  }
}
```

To find the handbag, the client will specify the following URL as target:
`"http://localhost:8080/webresources/product/Handbag"`

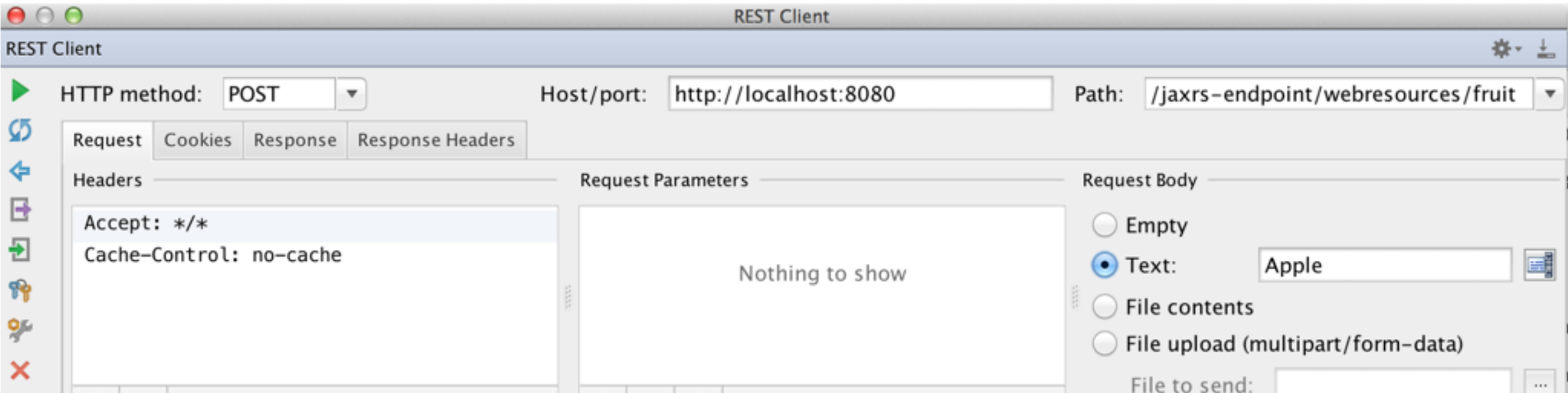
Walkthrough 2(start)

- Clone the GitHub project https://github.com/javaee-samples/javaee7-samples.git from IDEA using the menu VCS | Import from Version Control | Github.

- In Menu Run configure Wildfly selecting the application jaxrs-endpoint as an artifact for deployment (fast forward the video https://vimeo.com/91668238 to 3m30sec for instructions).

- Start the server using the menu Run. Observe that the Run window (View | Tool Windows | Run) includes the message “Deployed “jaxrs-endpoint.war”.

- Test the app using Views | Tool Windows | Rest Client.
  - Add an apple using Post request. Add a lemon using another Post request.
  - Retrieve the fruits using Get request.



Walkthrough 2(end)

- Review the code of the MyResource class from jaxrs-endpoint app. Observe that the Run window printed the messages from methods get() and post().

- Copy/paste the class MyResource into the class ProductService.

- Modify the class @Path annotation to use the /product value:

```
@Path("/product")
public class ProductService {…}
```

- Using the menu Run restart the server in the debug mode. Set the breakpoints in both methods get() and post() in the class ProductService.

- In Rest Client set the path to `/jaxrs-endpoint/webresources/product`. Add the product Handbag. Add another product Bracelet. The debugger should stop at breakpoints. Check the value of the variable payload in method post().

- Retrieve the product Handbag by setting the path in Rest Client to `/jaxrs-endpoint/webresources/product/Handbag`.

- Do you see anything worn with implementation of the PUT method in this sample code?

JAX-RS Java Client

- Need to know the URI of the endpoint, e.g. `/jaxrs-endpoint/webresources/product`
- Create and cache a new client factory and a new client
- JavaEE7-Samples project include jaxrs-client app:

```
private URL base = new URL("http://localhost:8080");

Client client = ClientBuilder.newClient();
target = client.target(URI.create(new URL(base, "webresources/product").toExternalForm()));
target.register(Person.class);
Person[] list = target.request().get(Person[].class);
```


More JAX-RX Client Samples

```
private WebTarget target;

Client client = ClientBuilder.newClient();
target = client.target("http://localhost:8080//jaxrs-endpoint/webresources/
product");

// Getting all products
target.request("application/json").get();
```

```
// Adding a new handbag
client.target(base)
        .path("webresources/product")
        .path("Handbag")
        .request(MediaType.APPLICATION_JSON)
        .post(Entity.text("Gucci Handbag"));
```


Context and Dependency Injection (JSR 346)

With CDI, a container injects dependencies into object.

With CDI managed object become decoupled - an object A Does not create an Object B with new. Containers will create an instance and five it to the object.

The Hollywood Principle: don’t call us. We’ll call you.

Injecting a CDI Bean

```
public interface Message {
    public String get();
}
```

```
public class MessageB implements Message {
    public MessageB() { }

    public String get() {
        return "message B";
    }
}
```

```
@WebServlet("/cdiservlet")
public class NewServlet extends HttpServlet {

    @Inject private Message message;

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                            throws IOException {

        response.getWriter().write(message.get());
    }
}
```

CDI beans are Java classes that can be managed by container

Qualifiers

If more than one class implement an interface, use qualifiers to specify which object to inject.

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Informal {}
```

```
@Informal
public class MessageC implements Message {
    public MessageC() { }

    public String get() {
        return "message C";
    }
}
```

```
@WebServlet("/cdiservlet")
public class NewServlet extends HttpServlet {

    @Inject @Informal private Message message;

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                              throws IOException {

        response.getWriter().write(message.get());
    }
}
```

CDI Scopes

Scope	Annotation	Duration
Request	`@RequestScoped`	A user's interaction with a web application in a single HTTP request.
Session	`@SessionScoped`	A user's interaction with a web application across multiple HTTP requests.
Application	`@ApplicationScoped`	Shared state across all users' interactions with a web application.
Dependent	`@Dependent`	The default scope if none is specified; it means that an object exists to serve exactly one client (bean) and has the same lifecycle as that client (bean).
Conversation	`@ConversationScoped`	A user's interaction with a servlet, including JavaServer Faces applications. The conversation scope exists within developer-controlled boundaries that extend it across multiple requests for long-running conversations. All long-running conversations are scoped to a particular HTTP servlet session and may not cross session boundaries.

Additional Materials

- Installing WildFly server and integrating it with IntelliJ IDEA: https://vimeo.com/91668238
- Oracle's tutorial on Java EE 7: http://docs.oracle.com/javaee/7/tutorial/doc/home.htm
- Java EE 7 Samples: https://github.com/javaee-samples/javaee7-samples
- Testing RESTful Web Services in IntelliJ IDEA: http://www.jetbrains.com/idea/webhelp/testing-restful-web-services.html
- Arquillian: An Integration testing framework for Java EE: http://docs.jboss.org/arquillian/reference/1.0.0.Alpha1/en-US/html_single/
- Postman Rest Client add-on for Google Chrome: http://bit.ly/18JpMha

Walkthrough 3

RESTful auction application skeleton

Walkthrough 3

- Import auction into IntelliJ IDEA using Gradle
- Overview of REST endpoints
  - @Path, @GET, etc
  - Filters - CORS filter https://developer.mozilla.org/en-US/docs/HTTP/Access_control_CORS
- Using CDI

WebSockets

Bi-directional communication for the Web

What the problem with HTTP?

- HTTP is request-based protocol
- HTTP is a very verbose protocol
- Fallback and hacks for achieving «real-time» feel
  - Polling
  - Long Polling
  - HTTP Streaming

Meet the WebSocket

- STANDARD PROTOCOL: WebSocket is a standardized technology (RFC6455).

- CLIENT-SIDE API: New `window.WebSocket` object. No plugin required

- SERVER-SIDE API: Part of Java EE 7 specification (JSR 356)

WebSocket allows

- Establish a connection
- Send message in both directions (Bi-directional)
- Send messages independent (Full Duplex)
- Close the connection

WebSocket handshake

- Here is sequence of the steps of initial handshake
- Client sends UPGRADE HTTP-request
- Server confirms UPGRADE
- Client receives UPGRADE response
- Client changes `readyState` property of WebSocket object to open

No plugin, only browser

- You can find browsers support chart http://caniuse.com/websockets

Java API for WebSocket Highlights

- Create WebSocket endpoints
  - with annotations (`@ServerEndpoint`, `@OnMessage`, etc)
  - API (Endpoint)
- Integration with other Java EE technologies

Walkthrough 4

Java API for WebSocket

Walkthrough 4

- Review `BidEndpointApi.java` for programmatic endpoint configuration
  - `AuctionWebSocketConfig.java`
- Review `BidEndpoint.java` for declarative endpoint configuration
- Debugging WebSocket messages with Chrome Dev Tools

Additional reading on WebSocket

- Dedicated chapter about WebSocket: Upgrading HTTP To WebSocket http://enterprisewebbook.com/ch8_websockets.html

- Dedicated book about WebSockets: The Definitive Guide to HTML5 WebSocket http://goo.gl/yCvpZU

- WebSocket book from the lead of JSR 356 spec: Java WebSocket Programming http://goo.gl/Vvspel

Homework

- Using the provided classes create the Java REST endpoint for receiving user's bids on a product in the auction in Java.
- Deploy it under WildFly server using IntelliJ IDEA.
- Integrate provided AngularJS Product page with this endpoint:
    1. The user select the product and places a bid
    2. The AngularJS app creates a JSON object Bid
    3. The AngularJS app makes a Rest call to the endpoint implemented in WildFly
    4. The Java code validates the received Bid, and if the price below the minimal price on the Product, the bid is rejected. If the price is more than a reserved price, the user receives congratulation on purchasing the product. In any other case the Bid is added to the collection of Bids, which has to be re-sorted, and the top bid price and the top bidder's ID is returned to the front end.
    5. The AngularJS app displays the Bid status top bid and top bidder on the Product page.
- Implement missing methods marked with TODO (`getJsonObject()` with `javax.json` API)