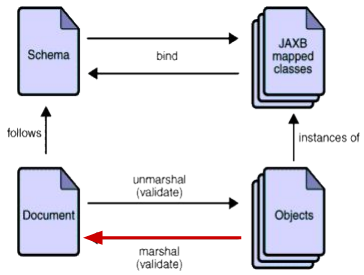
REST Backend

WS Slides #5

Content

- JAXB
- JAX-RS 2.x
- Root Resources
- JsonObject
- HATEOAS
- Client API

Java Architecture for XML Binding, JAXB



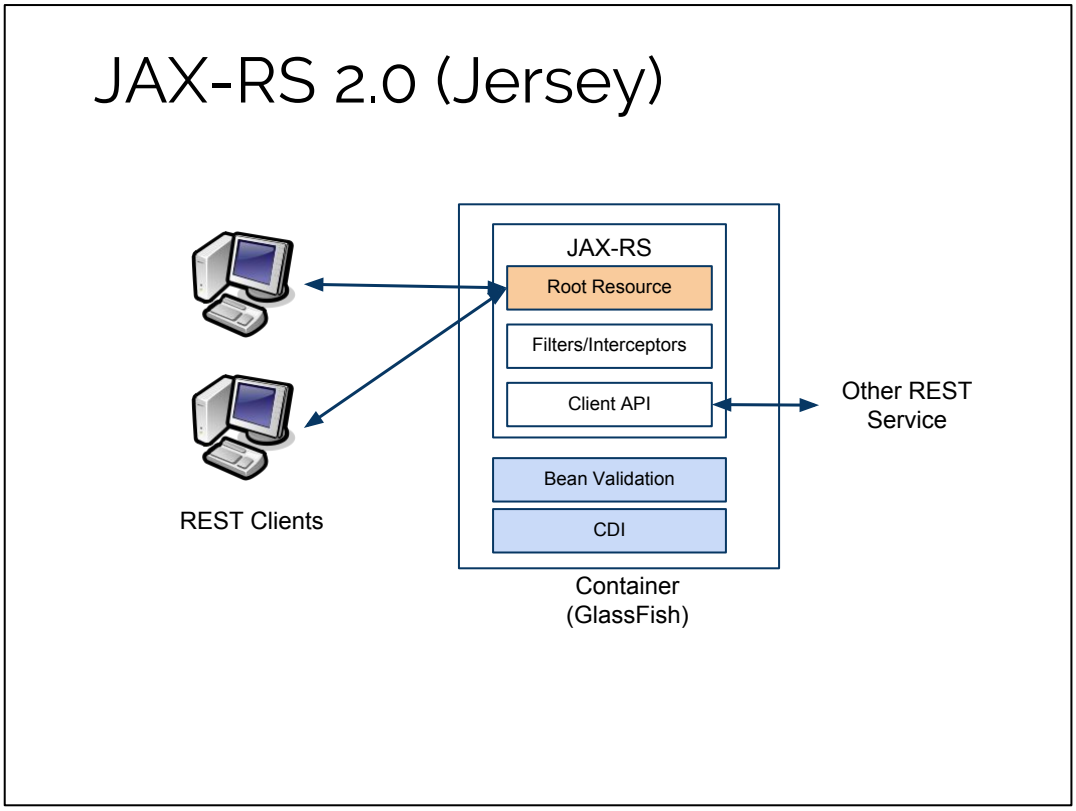

```java
@XmlRootElement
@XmlAccessorType(XmlAccessType.PROPERTY)
@XmlType(name = "PersonType", propOrder = {
    "pnumb","fname","age"
})
public class Person {

    private String fname;
    private int age;

    @XmlElement(required = true)
    public String getFname() {
        return fname;
    }

    @XmlAttribute(required = true)
    public Long getId() {
        return id;
    }
...
```

JAXB is an API to convert between XML Schema/XML documents and Java classes/objects

- Define mappings with annotations on class
- Mapping details (from reference implementation)
- Default constructor mandatory (not shown)
- Used in background by JAX-RX (upcoming).
- We mostly don't need explicit JAXB but exceptions may pop up.

JAXB part of Java SE, possible to run JUnit test without any dependencies!

- Possible to check the XML result of the annotations (marshalling the object)
- Details



Java API for RESTful Web Services, JAX-RS 2.0

- JAX-RS defines a set of Java APIs for the development of Web services built according to the Representational State Transfer (REST) architectural style
- Reference implementation is Jersey.
- JAX-RS Jersey developer guide
- ApacheCXF other implementation

Major parts

- Server side resource classes to handle request (Root Resource)
- Filters and Interceptors (for client and server) similar idea as Web application filters (cross cutting concerns)
- Client API (call RESTful service from Java)
- Content negotiation to decide on representation (XML, JSON …)
- HATEOAS support
- Asynchronous request and responses

Other JEE APIs involved

- Injection of resources (using JEE Context and Dependency injection, more later)
- Validation of values (using other JEE "Bean validation" more later)

Root Resource Class

```
@Path("/persons")
public class PersonResource {

    private final static PersonRegistry reg = …;
    @GET
    @Produces(value = {MediaType.APPLICATION_JSON})
    public Collection<Person> findAll() {
        return reg.findAll();
    }

    @PUT
    @Path(value = "{id}")
    @Consumes(value = MediaType.APPLICATION_FORM_URLENCODED)
    @Produces(value = {MediaType.APPLICATION_JSON})
    public void update(@PathParam("id") final Long id,
            @FormParam("fname") final String fname,
           @FormParam("age") final int age) {
        reg.update(new Person(id, fname, age));
    }
```

Root resource, the top level resource class, a class to handle HTTP requests (cousin to Servlet)

- A POJO, plain old Java object but …
- … must use @Path class annotation (relative URL, leading or trailing "/" doesn't matter) …
- … or at least one resource method with @Path … (not shown)
- … or a request method designator (= annotation on method): @GET, @POST, @PUT, @DELETE
- Non-private default ctor (not shown)
- Life cycle handled by JAX-RS runtime
    - Default is request: So this is stateless!

URI path templates (URIs with variables embedded within the URI syntax).

- Syntax { … }.
- Possible to use regexes like "{username: [a-zA-Z][a-zA-Z_0-9]*}" to specify constraints
- Automatic extraction of parameters
    - From URLs (@PathParam)
    - Form data (@FormParam)
    - Type conversion of parameters and results (objects must be JAXB mashable)

Content negotiation (conneg)

- @Consumes, @Produces to decide on representations (can be more)
- Must match HTTP header "Accept" (else "Not Found" returned)

Default HTTP responses

If a root resource class in application, NetBeans will find and add special icon in project (RESTful Web Services)

- Look for log message during deploy “INFO: Root resource classes found:”

Responses

Return 200 OK no content

```
return Response.ok().build();
```

Return 200 OK with content

```
return Response.ok(value).build();
```

Return 201 Created with location

```
return Response.created(uri).build();
```

Return 500

```
return Response.status(Status.INTERNAL_SERVER_ERROR)
                                                    .build();
```

Return 200 OK with generic value

```
GenericEntity ge =
 new GenericEntity<List<ProductWrapper>>(pp) {};
 return Response.ok(ge).build();
```

JAX-RS methods will return HTTP responses.

- The default HTTP response status codes not always what you want (and not always same as HTTP 1.1 Specification)
- Use class Response to tailor HTTP status codes
- Also useful for exceptions: Convert exception into custom response.
- If returning generic types (List<String>) must wrap Response in Generic Entity

JAX-RS default response codes

- GET: If found 200, else 204 (null)
- POST: 204, No content
- PUT: 204
- DELETE: 204

HTTP/1.1 suggests

- GET: same as above
- POST: 201, Created and location URI in response header (200 or 204 if URI not possible)
- PUT: If new resource 201 if modified 204
- DELETE: 200, 202 or 204

Application Config

```
@javax.ws.rs.ApplicationPath("webresources")
public class ApplicationConfig extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<>();
        addRestResourceClasses(resources);
        return resources;
    }

private void addRestResourceClasses(Set<Class<?>> resources){
    resources.add(edu.gu.hajo.rest.PersonResource.class);
    ...
}

...
```

`http://localhost:8080/.../webresources/persons`

JAX-RS provides a deployment agnostic abstract class Application for declaring root resource and provider classes, and root resource and provider singleton instances.

- A Web service may extend this class to declare root resource and provider classes.
- Root resource class must be "added" here, else not found
- NetBeans will generate
- NOTE: Will affect URL of application

Testing A Root Resource

GET request (single line)

```
curl -v -H "Accept: application/json"
   http://localhost:8080/rest_backend/
      webresources/persons
```

POST request (single line)

```
$ curl -v http://localhost:8080/rest_backend/
        webresources/persons --request POST --data
                  "pnumb=99&fname=XX&age=99"
```

More ways to test

- Simplest (but manual) using cURL, command line tool for transferring data with URL syntax
- Also possible and manual Chrome REST Console (or other)
- Testing from inside NetBeans
- There are testing frameworks but too complex for now

JsonObject

As parameter

```
@Consumes(value = MediaType.APPLICATION_JSON)
public void update(@PathParam(value = "id") final Long id,
                                            JsonObject json) {
    int age = json.getInt("age");
    Person p = new Person(id, json.getString("fname"), age);
    reg.update(p);
}
```

A return value

```
@Produces(value = {MediaType.APPLICATION_JSON})
public JsonObject count() {
   int c = reg.count();
   JsonObject value = Json.
       createObjectBuilder().add("value", c).build();
    return value;
}
```

JsonObject API

- Possible to use JSON objects as parameters return values
- Can't return primitive data, solution it to wrap in JsonObject

Context

Inject resources

```
@Context
private HttpHeaders headers;

@Context
private Request request;

@Context
private UriInfo uriInfo
```

Possible to inject "low level" objects in resource classes using @Context annotation (similar to ServletContext)

- UriInfo, used to build URIs (and more), for example when returning 201 created (URI to created resource)
- Should avoid low level…!

JAX-RS Filters

```
@Provider // Application scoped
public class PersonFilter implements ContainerRequestFilter,
                  ContainerResponseFilter {

    @Override
    public void filter(ContainerRequestContext requestContext) {
        ...
    }

    @Override
    public void filter(ContainerRequestContext requestContext,
              ContainerResponseContext responseContext) {
        ...
    }

}
```

Same idea as Servlet filters.

HATEAOS

Use of uriInfo

```
URI uri = uriInfo.getAbsolutePathBuilder()
                .path(String.valueOf(id)).build(p);
```

Use of Link

```
return Response.ok(p)
     .link("http://oracle.com", "oracle")
     .link(new URI("http://jersey.java.net"),
                          "framework").build();
```

A very important aspect of REST is hyperlinks, URIs, in representations that clients can use to transition the Web service to new application states

UriBuilder

- Class that makes it simple and easy to build URIs safely.
- UriBuilder can be used to build new URIs or build from existing URIs. For resource classes it is more than likely that URIs will be built from the base URI the web service is deployed at or from the request URI. The class UriInfo provides such information (in addition to further information, see next section).
- Example: Add URI of created resource to Response when returning Created 201.

Link

JAX-RS 2.0 introduces Link class, which serves as a representation of Web Link defined in RFC 5988. The JAX-RS Link class adds API support for providing additional metadata in HTTP messages (Link header), for example, if you are consuming a REST interface of a public library, you might have a resource returning description of a single book. Then you can include links to related resources, such as a book category, author, etc. to make the produced response concise but complete at the same time.

Client API

```
Client client =
          ClientBuilder.newBuilder().build();

WebTarget target = client
          .target("http://.../response/1");

Response response = target
    .request(MediaType.APPLICATION_JSON)
    .accept(MediaType.APPLICATION_JSON).get();

Person person =
            response.readEntity(Person.class);
```

Client API

- To consume a REST service from Java

Asynchronous JEE

```
@Path("/")
public class Resource {

    @Inject
     private Executor executor;

    @GET
     public void asyncGet(@Suspended final AsyncResponse
                                              asyncResponse) {
         executor.execute(() -> {
           String result = service.veryExpensiveOperation();
           asyncResponse.resume(result);
          });
     }
}
```

[Java EE: Asynchronous constructs and capabilities]

Here is where Asynchronous execution comes into play

- If the server thread processing the client request can be released/suspended and the actual business logic is executed in a separate thread (different than that of the original one), performance and scalability can be improved immensely !
- E.g. if a HTTP listener thread allocated to listen to client requests is released immediately, then it is free to attend to requests from other clients and the business logic can be executed in a separate container thread which can then return the response via appropriate methods such as java.util.concurrent. Future object or via callback handlers registered by the client.
- Think from an end user perspective – responsiveness matters a lot!

JEE APIs with async possibilities

- JAX-RS 2.0 (Java EE 7)
- Websocket 1.0 (Java EE 7)
- Concurrency Utilities 1.0 (Java EE 7)
- EJB 3.1 (Java EE 6)
- Servlet 3.0 (Java EE 6)

Also a [Future class]