REST Backend, JAXB, JAX-RS and Caching

WS Slides #4

Java Architecture for XML Binding, JAXB

API to convert between XML/XML Schema and Java classes/objects

- Mostly handled in background … (for service based approach)
- … but sometimes explicit need to specify convert conversion …
- … add annotations on class to define mapping
- Mapping details (from reference implementation **Metro**)

JAXB Example

```
// Specify mapping of objects to XML using JAXB annotations
@XmlRootElement(name="person")
@XmlAccesorType(XmlAccessType.PROPERTY) //Annotation on methods (or XmlAccessType.
FIELD)
public class Person {

    // Must have non private default ctor!
    public Person(){
    }

    private int id;
    private String fName;

    @XmlAttribute
    public void setId( int id ){
        this.id = id;
    }

    @XmlElement(name="fname")
    public void getFName(){
        return fName;
    }
}
```

Testing JAXB

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

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

REST Server Side

To implement a RESTful service we use **Java API for RESTful Web Services**, JAX-RS
- Reference implementation is Jersey (included in GlassFish)

Basically
- Connect Java **root resource** class/methods to URLs using annotation
- Annotate Java methods with HTTP-method (GET, PUT, …)
- Specify in/out MIME-types for methods
- Automatic extraction of parameters from URL
- Type conversion of parameters and results
- Possible asynchronous (similar to servlet, java.util.concurrent.*)

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

JAX-RS Root Resource Class

Root resource, the top level resource class

Basically
- Must use @Path class annotation…
- Or at least one **resource method** with @Path or a request method designator (= annotation on method): @GET, @POST, @PUT, @DELETE
- Non-private default ctor
- Life cycle handled by JAX-RS runtime
- Possible asynchronous calls

Content Negotiation (Conneg)

Different client needs different representation
- XML, JSON, YAML,...,

Resource methods can handle different MIME types. Specify with annotations (possible more type for single method)
- @Consumes (mime type(s) in request...)
- @Produces (...in response)
- Must match HTTP header "Accept" (else "Not Found" returned)

Root Resource Example

```
@Path("/persons")  // URL
public class PersonResource {  // Any Java class
    ...

    @GET
    //@Consumes often not needed more later
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public List<Person> selectAll() {
    ... // Will return List<Person> as XML or JSON (JAXB in
background)
    }

}
```

Person class <u>must</u> be JAXB marsh:able (because of APPLICATION_XML)

Root Resource Async Example

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

    private final ExecutorService executorService = newCachedThreadPool();

    @GET
    @Produces(value = {MediaType.APPLICATION_JSON})
    public void findAll(@Suspended final AsyncResponse asyncResponse) {
        executorService.submit(new Runnable() {
            @Override
            public void run() {
                asyncResponse.resume(doFindAll());
            }
        });
    }

}
```

Deploying a Root Resource

Important log message from GlassFish when running (check!)

```
INFO: Root resource classes found:                         // Good!
  class edu.chl.hajo.bjj.resource.PersonResource
INFO: No provider classes found.             // Normally no problem!
```

Trigger JAX-RS

JAX-RS a subsystem. Must specify when to “kick-in”
- Could have other kind of request (JSP)
- Handled by application config class (NetBeans will generate)

```
// Specifying path to trigger JAX-RS @javax.ws.rs.ApplicationPath
("webresources")
public class ApplicationConfig extends Application {
 ...
}

// Use, must have correct path
http://localhost:8080/rest_backend/webresources/persons
```

Testing a Service

There are JAX-RS client API (upcoming) and testing frameworks but too complex for now

Simpler (but manual): cURL

- Command line tool for transferring data with URI syntax
- Example

```
// Make a http request from command line
curl -v  http://localhost:8080/service_based/rs/persons --request
POST  --data "pnumb=99&fname=XX&age=99"
```

Or use the Chrome REST Console (or other)

URI Path Templates

URI's with embedded parameters

- Annotations on methods, values substituted runtime

```
// username is a parameter in request
@Path("/users/{username}")
// Possible using regex to be more specific
@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]}")
```

- Possible to retrieve request parameter as path (method) parameter

```
@GET
@Produces("text/xml")
@Path("/users/{username}")
public String getUser(@PathParam("username")
                String userName) {
     ...
}
```

Form Parameters

@FormParam, parameter annotation, extract posted form data (matching names of values)

```
@POST
@Consumes("application/x-www-form-urlencoded")
public Response post(@FormParam("pnumb") String pnumb,
        @FormParam("fname") String fname,
        @FormParam("age") int age) {
    ...
}
```

More options for parameters

Type Conversions

Automatic type conversion from/to Java

- All media types (*/*)

byte[], String, Reader (inbound only), File, DataSource, StreamingOutput (outbound only)

- XML media types (text/xml, application/xml and application/…+xml)

Source, JAXBElement, application supplied JAXB classes (types annotated with @XmlRootElement or@XmlType)

- Plain text (text/plain)

Boolean, Character, Number

If not satisfied implement custom MessageBodyReader/Writer

JSON Support

Conversion Java objects from/to JSON

POJO based i.e. transparently handled
- The default. No coding. Will normally suffice
- Incoming JSON → JsonObject
- Outgoing Object → JSON

If in need both XML and JSON
- Use JAXB annotations
- Note: There are different mappings of XML to JSON. Possible need to customize

Primitive Types

No natural XML/JSON representation of primitive types.

How to return?

- Solution: Wrap in JAXB annotated object or JsonObject, see samples

Standard HTTP Response Codes

Typical successful response codes for HTTP methods
- “200 OK”, if return value not null, message has body
- “204 No Content” if return value null (but ok), no message body

Errors
- Client error 4xx
- Server Error 5xx
- Examples “404 Not Found”, “406 Not Acceptable”, wrong data format, “405 Method Not Allowed”, bad method, 500 Internal Server Error (...NullPointerException ..:-) possible...)

JAX-RS Response Codes

JAX-RS default response codes

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

HTTP/1.1 Specification

- 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

Return Type Examples

By using the **Response** class we can modify the response codes
- Response will also wrap the return data

```
// POST to create new resource
// There's a ResponseBuilder object in background
Person p = new Person(id, fname, age);
try {
   reg.create(p);
   // Tell client where new resource is (URI to)
   URI uri =
   uriInfo.getAbsolutePathBuilder().path(String.valueOf(id)).build
(p);
   return Response.created(uri).build();  // 201
} catch (IllegalArgumentException e) {
   return Response.status(Status.INTERNAL_SERVER_ERROR).build();
}
```

Response and Generic Types

If using Response as return type

```
// Get a list
List<Person> ps = ... (List is generic class)

// Wrap list (note anonymous subclass)
GenericEntity<List<Person>> ge =
   new GenericEntity<List<Person>>(ps) {};
return Response.ok(ge).build();  // Response return type
```

Context

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

```
@Context
private HttpHeaders headers;

@Context
private UriInfo uriInfo

@Context
private Request request;
```

.. and more

REST Filters

Same idea as Servlet filters

- ContainerRequestFilter
    - Pre Match before resource method executed
    - Post Match after
    - Hit by any call, have to do URL matching/manipulation in code
- ContainerResponseFilter

JAX-RS Client Side

Client Side API to consume RESTful services

```
// Class PersonRegClient
private final WebTarget webTarget;
private final Client client;
private static final String BASE_URI = "http://localhost:
8080/rest_backend/webresources";

public PersonRegClient() {
    client = ClientBuilder.newClient();
    webTarget = client.target(BASE_URI).path("response");
}

public <T> T count(Class<T> responseType) throws ClientErrorException {
    WebTarget resource = webTarget;
    resource = resource.path("count");
    return resource.request(MediaType.APPLICATION_JSON).get(responseType);
}
```

Caching

" … If, every time a browser processed a URI beginning http://…, it actually fired off a GET at a server, and if, every time a GET hit a server, the server actually recomputed and sent the data, the Web <u>would melt down</u> PDQ. There is a lot of machinery available, on both the client and server side, to detect when the work of computing results and sending them over the network can be avoided. Normally, we use the term "**caching**" to refer to all this stuff." // Tim Bray

I.e.

- improve speed, because we want to deliver fast content to our consumer
- fault tolerance, because we want our service to deliver content also when it encounters internal failures
- scalability, because the WWW scales to billions of consumers through hypermedia documents and we just want to do the same thing
- reduce server load, because we don't want our servers to compute without the need of it

Types of Cache

Local cache, your browser's local copy

Proxy cache, a copy on some server on the way to the origin (the original Server), a middleman
- Content Delivery Network (CDN), large distributed system of servers deployed in multiple data centers in the Internet. The goal of a CDN is to serve content to end-users with high availability and high performance (example: Akamai)

Caching Strategy

Good candidates for caching are pages that
- Are accessed frequently
- Are stable for a period of time
- Contain a majority of contents that can be reused by a variety of users

A good example would be catalog display pages

Pages with sensitive data shouldn't be cached

HTTP Header: Cache-Control

HTTP response Cache-Control , some few parameters

Value (set by server)	**Description**
private	A cache mechanism may cache this page in a Private cache and resend it only to a single client. This is the default value. Most proxy servers will not cache pages with this setting.
public	Shared caches, such as proxy servers, will cache pages with this setting. The cached page can be sent to any user.
no-cache	Do not cache this page at all, even if for use by the same client.
no-store	The response and the request that created it must not be stored on any cache, whether shared or private. The storage inferred here is non-volatile storage, such as tape backups. This is not an infallible security measure.

Example Server Response

```
HTTP/1.1 200 OK
Content-type: application/xml
Cache-Control: private, no-store, max-age=300
```

- Only client may cache
- Must not be stored on disk
- Valid for 300 seconds

Cache Inconsistency

Cache introduces inconsistency!
- Resource previously served to a consumer is different from the one actually held by the server

To improve data's consistency, **Web Cache validation**
- Using conditional requests
- If resource has not changed server sends 304 Not Modified status
… (reduces workload/bandwidth)
- …else a full response

Conditional Requests

A conditional GET*) method requests that the entity be transferred only under the circumstances described by the conditional header field(s). The conditional GET method is intended to reduce unnecessary network usage by allowing cached entities to be refreshed without requiring multiple requests or transferring data already held by the client

- Headers are: If-Modified-Since, If-Unmodified-Since, If-Match, If-None-Match, or If-Range (If-* headers)

*) Also conditional updates (PUT)

If-* headers and ETag

GET request flow

- Request from client
- Web server return resource along with its corresponding ETag (Entity tag, a hash) in header (ETag: "6868976 96a7c876b7e")
- At next request client sends ETag (If-None-Match: "686897696a7c876b7e")
- If ETags match server sends HTTP 304 Not Modified i.e. ok to use cached object
- Else send current version of object

JAX-RS Cache Validation

```
public interface Request {
  ResponseBuilder.evaluatePreconditions( ETag eTag)
  ResponseBuilder.evaluatePreconditions( Date lastModified)
  ResponseBuilder.evaluatePreconditions( Date lastModified, ETag
eTag)
}
```

Use on Server side

```
// Create an ETag object (etag) for actual object then…
// (compare with request)
ResponseBuilder builder = request.evaluatePreconditions(etag);
```

If precondition true builder == null
If precondition false builder == A ResponseBuilder with appropriate status (possible 412 Precondition failed)