Jax-rs 2.0

https://dennis-xlc.gitbooks.io/restful-java-with-jax-rs-2-0-en/cn/part1/chapter8/building_and_invoking_requests.html

Contents

JAX-RS szerver

Inicializálás

A JAX-RS szerver futtatásához el kell indítsuk a Jersy servlet-et a WEB alkalmazásunkban, ami kiválóan megfér más servlet-ek mellett, pl JSF. A Jersey servletnek meg fogjuk mondani, hogy milyen PATH tartozik hozzá, így minden más er?forrás kérés továbbra is a JSF servlet-hez fog beesni.

Maven dependenciák

```
<dependency>
<groupId>org.glassfish.jersey.containers</groupId>
<artifactId>jersey-container-servlet</artifactId>
<version>2.28</version>
</dependency>
<dependency>
<groupId>org.glassfish.jersey.inject</groupId>
<artifactId>jersey-hk2</artifactId>
<version>2.28</version>
</dependency>
<dependency>
<groupId>org.glassfish.jersey.core</groupId>
<artifactId>jersey-common</artifactId>
<version>2.28</version>
</dependency>
```

web.xml

Ha JSF-et akarunk párhuzamosan JAX-RS-el használni, akkor nincs más dolgunk, mint hogy mint két serlvet implementációt felvegyünk a web.xml-be. Els?ként a JSF implementációt, ami az esetünkben PrimeFaces lesz, aztán meg felvesszük a JAX-RS serlvetet, ami az esetünkben Glassfish-Jeresy lesz.

A **Glassfish-Jeresy**-t többféle képen lehet paraméterezni. Vagy itt, a web.xml-ben adjuk meg a szükséges paramétereket (provide-erek, mapperek, stb..) vagy implementáljuk a ***javax.ws.rs.core.Application*** osztályt, megadjuk a helyét a ***javax.ws.rs.Application*** paraméterrel, majd a konfigurációt többi részét az osztályon belül definiáljuk.

Els?re nézzük az a web.xml-es konfigurációt. **<init-param>** szekciókkal kell megadni a Jersey paramétereit:

- jersey.config.server.provider.packages: meg kell adni azt a java csomagot, ahol a webservice implementációk és az exception mapper-ek vannak. Mi itt azt a csomagot adjuk meg, ahol a service implementációk vannak.
- jersey.config.server.provider.classnames: fel lehet sorolni konkrét class megadásokkal további implementációkat. Mi itt az Exception mapper-eket adjuk meg.

A servlet-mapping szekcióban els?re megadjuk hogy a Jeresey servlet a /**rest**/ útvonalon fog hallgatózni. Aztán adjuk meg a JSF servletet, ami minden másra illeszkedni fog.

```
...

<servlet>
<servlet-name>Faces Servlet</servlet-name>
<servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>


<servlet>
<servlet-name>Jersey Service</servlet-name>

<servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
<init-param>
<param-name>jersey.config.server.provider.packages</param-name>
<param-value>hu.adam.loginHelper.jaxrs.service</param-value>
```

```
</init-param>

<init-param>
<param-name>jersey.config.server.provider.classnames</param-name>
<param-value>
                           hu.adam.loginHelper.jaxrs.mapper.AppExceptionMapper;
                           hu.adam.loginHelper.jaxrs.mapper.ErrorMessage;
                           hu.adam.loginHelper.jaxrs.mapper.WebServiceExceptionMapper
                         </param-value>
</init-param>

<load-on-startup>1</load-on-startup>
</servlet>


<servlet-mapping>
<servlet-name>Jersey Service</servlet-name>
<url-pattern>/rest/*</url-pattern>
</servlet-mapping>


<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>/faces/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>*.jsf</url-pattern>
</servlet-mapping>
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>*.faces</url-pattern>
</servlet-mapping>
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
```

Másik alternatíva lenne ha JAX-RS konfigurációt a javax.ws.rs.core.Application osztály implementációjában definiálnánk. Ekkor a web.xml-ben csak az implementációs osztályt kell megadni: ha a web.xml-ben az

```
<servlet>
<servlet-name>Jersey Service</servlet-name>
<servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
<init-param>
<param-name>javax.ws.rs.Application</param-name>
<param-value>hu.adam.MyApplication</param-value>
</init-param>
```

Majd a tovább konfigurációkat az implementációs osztályban definiáljuk:

```
import java.util.HashSet;
import java.util.Set;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/rest/")
public class MyApplication extends Application {

 public Set<Class<?>> getClasses() {
        Set<Class<?>> s = new HashSet<Class<?>>();
                //Webservices
        s.add(LoginService.class);

                //Mappers
        s.add(AppExceptionMapper.class);
        s.add(GenericExceptionMapper.class);
        return s;
    }
}
```

Service osztály definiálása

A service osztályokat nagyon egyszer? definiálni, az alábbi osztály loginByEmail metódusa a POST:<app root>/rest/login/loginbyemail URL-en lesz elérhet?. A HTTP body-ban a LoginByEmalLoginHelperRequest osztály JSON reprezentációját fogja várni, amit automatikusan mappelni fog a JAX-RS egy LoginByEmalLoginHelperRequest objektum példányba. A metódus a TokenResponse típusú objektummal tér vissza, mait a JAX-RS automatikusan JSON -ra fog mappelni anélkül, hogy vagy a request vagy a response objektumba bármilyen annotációt el kéne helyezni. Bonyolult, vagy nem egyértelm? struktúrák esetén szükség lehet rá, hogy annotációkkal támogassuk a JSON<->JAVA mapper-t, lásd lentebb.

```
import javax.ws.rs.POST;
import javax.ws.rs.Path;


@Path("/login/")
@Consumes({ "application/json" })
@Produces({ "application/json" })
public class LoginService {

@POST
@Path("/loginbyemail")
public TokenResponse loginByEmail(LoginByEmalLoginHelperRequest loginByEmalRequest) throws WebServiceException {

return new TokenResponse();
}

        @GET
@Path("/book/{isbn}")
public Book getBook(@PathParam("isbn") String id, @QueryParam("title") String title) {

Book oneBook = new Book();
oneBook.setIsbn("ISBN11111");
```

```
oneBook.setTitle("Mastering Jax-RS");

return oneBook;

}

}
```

A második metódus (getBook) a *GET:<app root>/rest/book/* URL-en érhet? el. Vár egy PATH paramétert a /book/ után, valamint egy title nev? QUERY paramétert, tehát a teljes request valami ilyesmi lesz: *GET:<app root>/rest/book/12345?title=adam*

Path paraméterek kezelése

Query paraméterek kezelése

Header paraméterek kezelése

```
@GET
@Path("/book/")
public Book getBook(@HeaderParam("Authorization") String token) {
            ...

}
```

Response manuális összeállítása

A @GET/POST/PUT annotációkkal ellátott metódusoknál a visszatérési objektum típus meghatározására két lehet?ségünk van.

- Ha rest metódusnak egy POJO a visszatérési értéke (vagy egy abból képzett típus) akkor sikeres futás estében, a metódus visszatérése után a JAX-RS ebb?l automatikusan REST választ fog generálni, a POJO-t JSON/XML-re fogja konvertálni, és be is állítja a HTTP státuszt, nekünk ezzel ebben az esetben semmi dolgunk. Viszont több okból is meg van kötve a kezünk: Ha a metódus sikeresen elfut, akkor a JAX-RS ezt mindig sikeres futásnak fogja tekinteni, és a HTTP státuszt mindig 200-ra fogja állítani, nem tudunk beavatkozni a válasz elkészítésébe.

```
@GET
@Path("/contract")
public Contract getContract() {
    Contract contract = new Contract();
    return contract;
}
```

- Ha a service metódusnak nem egy POJO a visszatérési értéke, hanem a javax.ws.rs.core.Response, akkor teljes kontrolunk van a válasz összeállításában, viszont mindent mind sikeres, mind sikertelen ágon nekünk kell kézzel a válaszba belerakni a megfelel? response JAVA objektumot és beállítani a HTTP státuszt.

https://dennis-xlc.gitbooks.io/restful-java-with-jax-rs-2-0-en/cn/part1/chapter7/complex_responses.html Alább láthatunk egy példát a teljesen manuális response összeállítására. Ha a header-t és azon belül a sütiket is állítani akarjuk, akkor muszáj ezt a megközelítést használni.

```
@GET
@Path("/contract2")
public Response getContract() {

    Contract contract = new Contract();

    ResponseBuilder builder = Response.ok(contract);
    builder.language("en").header("Some-Header", "some value").cookie(new NewCookie("adamCooke", "value"));

    return builder.build();
}
```

A státuszt kétféle képen határozhatjuk meg A Resonse osztálynak többféle metódusa van, ami beállítja a státuszt: ok=200, created=204 ..., vagy mi állítjuk be kézzel: .status(int)

```
public Response getContract() {
    ...
    return Response.status(Status.OK).type(MediaType.APPLICATION_JSON).build();
}
```

Hibakezelés

Áttekintés

Ez egy sarkalatos pontja a REST interfész implementációnak, ugyanis a kliensnek minden esetben vissza kell adni egy szabványos REST választ, ahol hiba esetén lehet?leg megmondjuk, hogy mi történt, tehát minden hibát le kell valahogy kezelni. Úgy fogjuk megoldani, hogy hiba esetén egyedi response objektumot dobjuk, hogy hiba esetén egy saját Exception-t dobunk, amit egy Exception provider-el feldolgozunk, és egyedi hiba objektumot készítünk bel?le a válaszba.

Ahhoz hogy a hibaágakat is kezelni tudjuk, létre kell hozzunk egyedi Exception osztályokat, és hozzájuk tartozó response mapper-eket (Provider), amik az Exception eldobása estén megkapják a vezérlést, és az Exception alapján össze tudják állítani a választ. Ehhez három osztályt kell létrehozzunk:

1. ErrorResponse osztály, egy egységesített struktúra, amit minden nem 200-as válaszban vár a kliens. (pl hibakódok, szöveges leírás)
2. Saját Exception implementáció (WebServiceException), amibe minden olyan infónak szerepelnie kell, ami alapján az ErrorResponse osztály elkészíthet?.

3. Exception provider osztály, ami megkapja a vezérlést ha WebServiceException dobódott, és a benne található információk alapján példányosítja a ErrorResponse osztályt, majd beleteszi a REST válaszba, amit a JAX-RS JSON formátumra fog hozni.

Jax-RS provider típusok

A JAX-RS-ben 3 típusa létezik a provider-ekenek, melyekkel három különféle feladatot láthatunk el:

- Entity provider

https://docs.huihoo.com/jersey/2.13/message-body-workers.html
The Entity provider API contains 2 interfaces. One for handling inbound entity representation-to-Java de-serialization - **MessageBodyReader<T>** and the other one for handling the outbound entity Java-to-representation serialization - **MessageBodyWriter<T>**. A MessageBodyReader<T>, as the name suggests, is an extension that supports reading the message body representation from an input stream and converting the data into an instance of a specific Java type. A MessageBodyWriter<T> is then responsible for converting a message payload from an instance of a specific Java type into a specific representation format that is sent over the wire to the other party as part of an HTTP message exchange. Both of these providers can be used to provide message payload serialization and de-serialization support on the server as well as the client side.

- Context provider

https://www.logicbig.com/tutorials/java-ee-tutorial/jax-rs/context-resolver.html
Context providers supply a context object to resource classes or to other providers.A context provider class must implement the ContextResolver<T>interface.

- Exception provider

These providers map a checked or runtime exception to an instance of Response. They implement ExceptionMapper<T>. Számunkra ez a lényeges. Itt fogjuk az exception-t

Megvalósítás

Els?ként definiáljuk azt az osztályt ami alapján a NEM 200-as státusz esetén a válasz objektumot létre akarjuk hozni. Létrehoztunk egy Facory metódust is a JSON<->JAVA konverzióra, amit **@JsonCreator** annotációval láttunk el (a metódusnak az 'of' nevet adtuk). A **@JsonProperty**-val megadtuk az egyez mez?k JSON nevét és Jáva típusát.

```
import com.fasterxml.jackson.annotation.JsonCreator;
import com.fasterxml.jackson.annotation.JsonProperty;

public class ErrorMessageResponse implements Serializable {

private String correlationId;

private String message;

        @JsonCreator
        public static ErrorMessageResponse of
                    (@JsonProperty("correlationId") String correlationId,
                     @JsonProperty("messages") String message) {
        return new ErrorMessageResponse(correlationId, message);
        }

        //Getters and setters and constructor...
}
```

A HTTP válaszba JSON alakban ez így fog kinézni:

```
{"correlationId":"XXX", "message":"111"}
```

Hozzunk létre egy egyedi Exception implementációt, amit minden kezelt hiba estén mi fogunk dobni a webservice metódusok belsejében. Ezt az exception-t fogja elkapni a mapper osztályunk, és alakítja majd át ErrorMessageResponse objektumra. Az exception-t származtassuk le az IllegalArgumentException-böl, és a konstruktorban töltsük ki az ?s konstruktorát is.

```
public class WebServiceException extends IllegalArgumentException {
        //HTTP status
private int status;

private String correlationId;

private String responseMessage;


public WebServiceException(String correlationId, String responseMessage, int status, String message, Throwable cause) {
super(message, cause);
            ...
        }

        //getters...
}
```

És végezetül nézzük meg az Exception provider osztályt aminek implementálni kell a ExceptionMapper<T> interfészt és a toResponse metódust, amivel az Exception alapján el?állítjuk a válasz objektomt.

```
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;
import javax.ws.rs.ext.Provider;
import hu.adam.loginHelper.exception.WebServiceException;

@Provider
```

```
public class WebServiceExceptionMapper implements ExceptionMapper<WebServiceException> {
@Override
public Response toResponse(WebServiceException exception) {

                ErrorMessageResponse emr = new ErrorMessageResponse(exception.getCorrelationId(), exception.getResonseMessage());

return Response.status(exception.getStatus())
.entity(emr)
.type(MediaType.APPLICATION_JSON).
build();
}
}
```

Használata:

```
@GET
@Path("/contract")
public Contract getContract() {
try {
Contract contract = new Contract();
} catch (Exception e) {
throw new WebServiceException("111", "service-error",404, e.getMessage(), e);

}
return contract;
}
```

A válaszban ez így fog megjelenni:

```
response code: HTTP 404
response boy: {"correlationId":"111", "message":"service-error"}
```

Cookie kezelés

https://www.logicbig.com/tutorials/java-ee-tutorial/jax-rs/cookie-param.html

JAX-RS kliens

https://dennis-xlc.gitbooks.io/restful-java-with-jax-rs-2-0-en/cn/part1/chapter8/building_and_invoking_requests.html

Kliens bemutatása

Maven dependecia:

```
<dependency>
<groupId>org.glassfish.jersey.core</groupId>
<artifactId>jersey-client</artifactId>
<version>2.28</version>
</dependency>
```

A JAX-RS kliens használata magáért beszél, elég intuitív. A ClientBuilder-el csinálunk egy globális kliens példányt, majd a klienssel készítünk sorban egy target-et, hozzáadunk path-t, request-et, metódust stb... A post(), put() metódusok két paramétert várnak, az els? a request body objektum (ha van ilyen az interfészen), a második a válasz típusa. A get() csak egy paramétert vár, a válasz típusát. A JSON<->Java mappelést a JAX-RS automatikusan el fogja végezni.

Request és Response megadása

Nézzünk egy példát ahol a kliens meghívja a POST:login interfészt, a request body JSON stukturáját a LoginRequest objektum írja le, a választ pedig a LoginResponse.

```
public class LoginResponse {
   private String user;
   private String pass;
   //getters and setters ...
}

public class LoginResponse {
   private String userId;
   private String token;
   //getters and setters ...
}
```

Ekkor a request így fog kinézni:

```
$ curl -XPOST 'localhost:8080/rest/login?' -H 'Content-Type: application/json' -d'
{ "user":"adam",  "pass":"1234" }
```

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;


    private static Client client = ClientBuilder.newClient();

    LoginRequst loginRequest = new LoginRequest();

    LoginResponse loginResponse =  client.target(baseUrl).path("login").
                            request(MediaType.APPLICATION_JSON).post(Entity.json(loginRequest), LoginResponse.class);
```

A válaszban az alábbi JSON fog visszaérkezni:

```
{ "userId":"11111", "token":"XXXXXX"}
```

Amit a JAX-RS mappelni fog egy LoginResponse objektumba.

Templét használata a request-ben

A request URL-ben használhatunk PATH és request paramétereket is, amikhez a JAX-RS kliens behelyettesít? metódusokat biztosít. A PAHT paramétereket a resolveTemplate() metódussal, míg a QUERY paramétereket a queryParam() metódussal helyettesíthetjük be:

```
WebTarget target = client.target("http://commerce.com/customers/{id}")
                         .resolveTemplate("id", "123")
                         .queryParam("verbose", true);
```

Listák kezelése a válaszban

Ha az interfész válaszában egy POJO listát kapunk vissza, akkor a válasz típusának a megadását az alábbiak szerint kell megadni: ?

```
Client client = ClientBuilder.newClient();


List<Book> books = client.target(url)
.request(MediaType.APPLICATION_JSON).get(new GenericType<List<Book>>(){});
```

Hibakezelés

A hibák kezelésére három megközelítést fogunk alkalmazni, amik egybevágnak a server implementációnál tárgyalt megközelítésekkel.

1. Exception kezelés: A get(), put(), post().. metódusoknál továbbra is megadjuk a válasz objektum típusát. 200 -as válasz esetében a JAX-RS automatikusan mappelni fogja a választ a megadott objektumba, így ha nem volt hiba, egy Java objektumban megkapjuk a választ. Ha bármilyen hiba dobódik az interfész hívása közben, akkor a JAX-RS kivétel típusából tudhatjuk, hogy milyen hiba történt. A kivételb?l ki tudjuk szedni a válasz body-ban esetlegesen küldött objektumot.
2. A get(), post().. HTTP metódusokban nem mondjuk meg a válasz típusát. Ekkor a kliens a HTTP státusztól függetlenül egy **javax.ws.rs.core.Response** objektummal fog visszatérni ha volt hiba hanem. Ekkor a Response-ból nekünk kell kiszedni a válasz objektumot a **readEntity(Class<T> entityType)** metódussal és a HTTP státuszt a **getStatus()** metódussal.
3. A harmadik megközelítés egy változata a másodiknak. Ahelyett, hogy a readEntity() hívással a JAX-RS -e bíznánk a parszolást, a JSON/XML String reprezentációját kérjük el a válaszból, és azt manuálisan parszoljuk.

Exception kezelés

https://dennis-xlc.gitbooks.io/restful-java-with-jax-rs-2-0-en/cn/part1/chapter7/exception_handling.html
Talán ez a legtisztább megközelítés, megadjuk a HTTP metódusban a válasz objektum típusát, amit sikeres futás esetén (HTTP 200) meg fogunk kapni a megadott JAVA objektumban out-of-the-box.

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.core.Response;

    private static Client client = ClientBuilder.newClient();

    LoginRequst loginRequest = new LoginRequest();
    try {
            LoginRequst loginRequest =  client.target(baseUrl).path("login").
                                        request(MediaType.APPLICATION_JSON).post(Entity.json(loginRequest), LoginRequst.class);

// 400-as hibák
} catch (ClientErrorException e) {

ErrorMessageResponse errorMessageResponse = e.getResponse().readEntity(ErrorMessageResponse.class);
                int status = e.getResponse().getStatus();

// 500-as hibák
} catch (ServerErrorException e) {
                ...
// 300-as redirekt hibák
} catch (RedirectionException e) {
                ...
}
```

Base Exception	Status code range	Description
ClientErrorException	4XX	Client side error
ServerErrorException	5XX	Server side error
RedirectionException	3XX	Redirect

Exception	Status code	Description
BadRequestException	400	Malformed message
NotAuthorizedException	401	Authentication failure
ForbiddenException	403	Not permitted to access
NotFoundException	404	Couldn?t find resource
NotAllowedException	405	HTTP method not supported
NotAcceptableException	406	Client media type requested not supported
NotSupportedException	415	Client posted media type not supported
InternalServerErrorException	500	General server error

ServiceUnavailableException 503 Server is temporarily unavailable or busy

Response objektum használata

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.core.Response;

    private static Client client = ClientBuilder.newClient();


    LoginRequst loginRequest = new LoginRequest();

    Response response =  client.target(baseUrl).path("login").
                            request(MediaType.APPLICATION_JSON).post(Entity.json(loginRequest));

    int status = e.getResponse().getStatus();

    if (status < 300) {
        LoginResponse loginResponse = response.readEntity(LoginResponse.class);

    } else {
        ErrorMessageResponse errorMessageResponse = response.readEntity(ErrorMessageResponse.class);
    }
```

Response as String

```
<dependency>
                <groupId>com.fasterxml.jackson.core</groupId>
                <artifactId>jackson-core</artifactId>
                <version>2.9.8</version>
            </dependency>
            <dependency>
                <groupId>com.fasterxml.jackson.core</groupId>
                <artifactId>jackson-databind</artifactId>
                <version>2.9.8</version>
            </dependency>
            <dependency>
                <groupId>com.fasterxml.jackson.core</groupId>
                <artifactId>jackson-annotations</artifactId>
                <version>2.9.8</version>
            </dependency>

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.core.Response;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;

    private static Client client = ClientBuilder.newClient();
    private static ObjectMapper objectMapper = new ObjectMapper();

    LoginRequst loginRequest = new LoginRequest();

    Response response =  client.target(baseUrl).path("login").
                            request(MediaType.APPLICATION_JSON).post(Entity.json(loginRequest));

    String responseString = response.readEntity(String.class);
    int status = e.getResponse().getStatus();

    LoginResponse loginResponse = objectMapper.readValue(responseString,LoginResponse.class);
```