**Exception Handling & Exception Mapper:**

1. Purpose Exception Handling

2. Purpose Exception Handling using Http Response Status codes

3. RESTful Business Requirement Document for developers and Clients

4. javax.ws.rs.WebApplicationException

|-Approach-1: ThrowingUserExceptionNormally

|-Approach-2: SendExceptionDetailsUsingResponse

|-Purpose of WebApplicationException

|-Approach-3: WebApplicationException

5. Practical difference between WebServices and RESTful services

6. Exception Mapping or Mapping default exceptions

7. Sub exception classes of WebApplicationException

8. JAX-RS exception hierarchy or Complete Exception Flow in JAX-RS API and Overriding Error Details

|-Internal working Flow

|-JAX-RS Complete Exception Flow project Diagram

9. RESTful as DB-REST-Wrapper

|-Advantages of REST wrappers on top of the DB

|-Use Cases for REST Wrappers

10. Wrapping Up

**1. Purpose Exception Handling:**

Whenever a Resource throws an Exception then we need to communicate with the client with proper message so that he can understand what is went/happening at the server so that he can communicate or resend the req back for further course of action.

**2. Purpose Exception Handling using Http Response Status codes:**

As we are dealing with integrated tech the client may not be java rather client can be anything so that if we send java Exception the other people cannot understand that is the reason we need Http Status coded which can work with any language bcz every can use Http protocol so that they cannot understand easily. So we need to build the Exception info as part of the Response obj using Status codes so that client can identify the status code and can get the failure or success info from the server.

In Web Services we can use soap: faults to build the error response back to the consumer but in REST we communicate back to the client using status codes.

In Web Services and RESTful services also faults or status will send back as part of the response only stating that reason for the failure.

Errors can be reported to a client in JAX-RS 2-ways

1. Either by creating and returning the appropriate Response object or

2. By throwing an exception.

Our application code is allowed to throw any checked (classes extending java.lang.Exception) or unchecked (classes extending java.lang.RuntimeException) exceptions we want.

Thrown exceptions are handled by the JAX-RS runtime if we have registered an exception mapper. Exception mappers can convert an exception to an HTTP response. If the thrown exception is not handled by a mapper, it is propagated and handled by the container (i.e., servlet) bcz JAX-RS is running within Web-Container/Servlet-Container.

JAX-RS also provides the javax.ws.rs.WebApplicationException. This can be thrown by application code and automatically processed by JAX-RS without having to write an explicit mapper. Let’s look at how to use the WebApplicationException first. We’ll then examine how to write our own specific exception mappers.

**3. RESTful Business Requirement Document for developers and Clients:**

Describes about the business requirements and contains

What will be the Query params,

What will be the Path params,

What will be the Request-Body.

What will be the Request format and its elements and every element has to be described in details like which one has to valid, which one should not be valid.

The Response body also will be specified what are all the elements to be send as part of the response body in detail and each element description will be provided.

There will be 2-Documentaions

1-for Resource development guide or developers guide and

1-for client to access the REST resource explain about xml elements.

Describing about each and every operation they will give documentation with method name, URL, elements which are mandatory which are optional and description each element, status code etc.

For example:

See documentation which pdf name mention here Business Document\_10\_1691.pdf see page: 24

Some other examples:

Example:

beid description

Status code table:

Result-code description

000 Ok req accepeted

001 FieldFormatNotSupported

005 Internal Server error

006 Internal Server error

008 Internal error

009 UnAutherizedError

004 AlreadyRegisted

045 AccountNameAllReadyExist

046 InvalidCardType

047 RequiredFieldsMissing

How do you start creating a RESTful Web Services?

How do u get the requirement and who will define it?

What are all the aspects that has define in the documentation?

**4. javax.ws.rs.WebApplicationException:**

**Approach-1: Throwing User-Exception Normally:**

@Path("/card")

public class CardAgent {

// Applying for Credit Card

@POST

@Consumes(MediaType.APPLICATION\_XML)

@Produces(MediaType.APPLICATION\_XML)

@Path("/apply/{bank}/{uuid}")

public Response applyCard(InputStream is,

@PathParam("bank") String bankName,

@PathParam("uuid") String uuid) {

// validate logic

if (bankName.equals("icici") && uuid.equals("C1")) {

throw new CardAlreadyIssuedException("Already User has been issued the card");

}

return Response.ok().build();

}

}

Access the application:

http://localhost:8083/3.1ThrowingUserExceptionNormally/rest/card/apply/icici/C1

Select POST

Content-Type: application/xml

Send some dummy xml to test

<card></card>

Response:

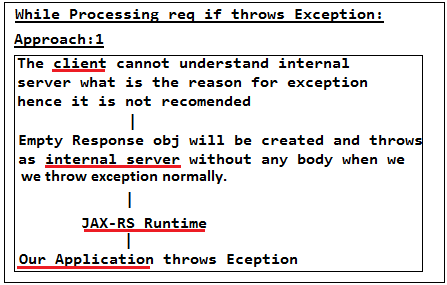
500: Internal Server Error

Stacktrace as error details in body which cannot be understand by the client.

Observe at server console:

com.wae.exception.CardAlreadyIssuedException: Already User has been issued the card.

That means if we throw user Exception normally JAX-RS Runtime will sends back to the client as 505 Internal Server Error which client cannot understand.



**Approach-2: Send Exception Details Using Response:**

@Path("/card")

public class CardAgent {

// Applying for Credit Card

@POST

@Consumes(MediaType.APPLICATION\_XML)

@Produces(MediaType.APPLICATION\_XML)

@Path("/apply/{bank}/{uuid}")

public Response applyCard(InputStream is,

@PathParam("bank") String bankName,

@PathParam("uuid") String uuid) {

try {

// call the service layer so that service layer will throw this exception actually

if (bankName.equals("icici") && uuid.equals("C1")) {

throw new CardAlreadyIssuedException("Already User has been issued the card");

}

} catch (CardAlreadyIssuedException e) {

// Actually in entity() method we need to return JAXB obj which

// contains error details as obj

return Response.status(409)

.entity("<error><error-code>409</error-code><message>Card Already Issued</message></error>").build();

// In this Approach code readability will not be there bcz exception is looks like response so we cannot code properly.

}

return Response.ok().build();

}

}

Access the application:

http://localhost:8083/3.2SendExceptionDetailsUsingResponse/rest/card/apply/icici/C1

Select POST

Content-Type: application/xml

Send some dummy xml to test

<card></card>

Response:

Status code : 409 Conflict (Which we given)

Error details as Response:

<error>

<error-code>409</error-code>

<message>Card Already Issued</message>

</error>

Observe at server console:

Nothing will be there in the server console bcz we modelled exception details as Response so that it send Exception as error details using Response obj.

This approach gives response with appropriate status code that we given and error details to the client but it will not gives readability to the developers whether it is exception response or normal response. Hence it not recommend to use. That is the reason JAX-RS has provided WebApplicationException.

**Purpose of WebApplicationException:**

Instead of throwing our exception normally or returning status codes and error details as normal responses directly, JAX-RS has provided one exception called as WebApplicationException which we can throw directly by populating with proper response code, with proper error message using Response. So that for the developer code readability will be more bcz we are throwing exception with proper error details using Response.

The JAX-RS API has a built in unchecked exception called as WebApplicationException that applications can throw. We can pre-initialized this exception with either a Response or a particular status code by using methods that JAX-RS has provided which are given below.

public class WebApplicationException extends RuntimeException {

public WebApplicationException() {...}

public WebApplicationException(Response response) {...}

public WebApplicationException(int status) {...}

public WebApplicationException(Response.Status status) {...}

public WebApplicationException(Throwable cause) {...}

public WebApplicationException(Throwable cause,

Response response) {...}

public WebApplicationException(Throwable cause, int status) {...}

public WebApplicationException(Throwable cause,

Response.Status status) {...}

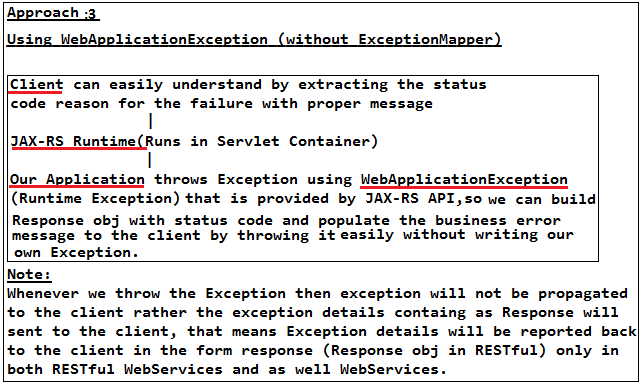
public Response getResponse() {...}

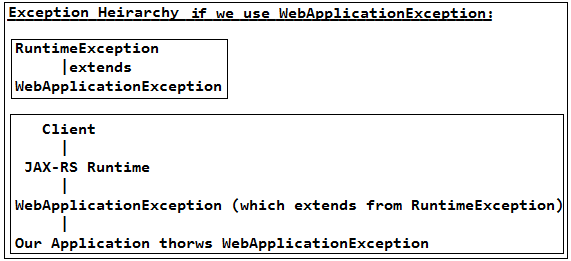
}

When JAX-RS sees that a WebApplicationException has been thrown by application code, it catches the exception and calls its getResponse() method to obtain a Response to send back to the client. If the application has initialized the WebApplicationException with a status code or Response object, that code or Response will be used to create the actual HTTP response. Otherwise, the WebApplicationException will return a status code of 500, "Internal Server Error" to the client.

**Approach-3: WebApplicationException**

For example, let’s say we have a web service that allows clients to apply for credit cards details represented in XML:





@Path("/card")

class CardAgent {

// Applying for Credit Card

@POST

@Consumes(MediaType.APPLICATION\_XML)

@Produces(MediaType.APPLICATION\_XML)

@Path("/apply/{bank}/{uuid}")

public Response applyCard(InputStream is,

@PathParam("bank") String bankName,

@PathParam("uuid") String uuid) {

// To check various possibilities, comment one by one case

// Case: 1 (Directly throw WebApplicationException)

if (bankName.equals("icici") && uuid.equals("C1")) {

// we can close the resources directly here without writing

// try-finally combination to avoid lengthy code bcz we know here we

// terminating the program so we can close here itself.

throw new WebApplicationException(Response.status(Status.CONFLICT)

.entity("<error><error-code>409</error-code><message>Card Already Issued(Using Case: 1)</message></error>").build());

}

// Case: 2 (throw WebApplicationException inside our

// User-Checked-Exception by conversion mechanism)

/\*\*

\* Note: We can throw directly CardAlreadyIssuedException but if we

\* throw this JAX-RS Runtime will displays default error stack trace but

\* we don't want that default stack trace rather we need to build error

\* Response that is the reason we need to throw WebApplicationException

\* so that JAX-RS will handles bcz he knows how to handle

\* WebApplicationException which contains/allows us to build error

\* details as in Response obj.

\*/

/\*\*

\* If we wanted to Wrap/convert one exception to another exception then

\* only we need here try catch otherwise not required to take try-catch

\* bcz JAX-RS will handles the exception if we wrap our exception to

\* WebApplicationException. If we don't want to wrap then not required

\* try-catch rather we can throw directly and declare to throws bcz it

\* is Checked-Exception.

\*/

try {

// Here we are throwing exception to observe the o/p but in reality

// service layer validator classes will throws exception

if (bankName.equals("icici") && uuid.equals("C1")) {

throw new CardAlreadyIssuedException("Already User has been issued the card");

}

} catch (CardAlreadyIssuedException e) {

// Actually in entity() method we need to return JAXB obj which

// contains error details as obj

// Status.CONFLICT indicates status code as 409 already exists

throw new WebApplicationException(Response.status(Status.CONFLICT)

.entity("<error><error-code>409</error-code><message>Card Already Issued (Using Case: 2) </message></error>").build());

}

// Case: 3 (throw WebApplicationException inside our

// User-Un-Checked-Exception by conversion mechanism)

/\*\*

\* If we wanted to Wrap/convert one exception to another exception then

\* only we need here try-catch otherwise not required to take try-catch

\* bcz JAX-RS will handles the exception if we wrap our exception to

\* WebApplicationException. If we don't want to wrap then not required

\* try-catch rather we can throw directly.

\*/

try {

// Here we are throwing exception to observe the o/p but in reality

// service layer validator classes will throws exception

if (bankName.equals("icici") && uuid.equals("C1")) {

throw new CardAlreadyIssuedRuntimeException("Already User has been issued the card");

}

} catch (CardAlreadyIssuedRuntimeException e) {

// Actually in entity() method we need to return JAXB obj which

// contains error details as obj

throw new WebApplicationException(Response.status(Status.CONFLICT)

.entity("<error><error-code>409</error-code><message>Card Already Issued(Using Case:3)</message></error>").build());

}

return Response.ok().build();

}

}

Access the application:

Case: 1

http://localhost:8083/3.3WebApplException/rest/card/apply/icici/C1

Select POST

Content-Type: application/xml

Send the some dummy xml to test

<card></card>

Response:

Status code : 409 Conflict (Which we given)

Observe Error details as Response:

<error>

<error-code>409</error-code>

<message>Card Already Issued(Using Case: 1)</message>

</error>

Observe at server console:

Nothing will be there in the server console bcz we modelled exception details as Response so that it send Exception as error details using Response obj and if we wanted to see the exception then we can take try-catch to printStackTrace() and to log then again throw it but not required unless we wanted to see the to do logging.

Case: 2

http://localhost:8083/3.3WebApplException/rest/card/apply/icici/C1

Select POST

Content-Type: application/xml

Send the some dummy xml to test

<card></card>

Response:

Status code : 409 Conflict (Which we given)

Observe Error details as Response:

<error>

<error-code>409</error-code>

<message>Card Already Issued(Using Case: 1)</message>

</error>

Case: 3

http://localhost:8083/3.3WebApplException/rest/card/apply/icici/C1

Select POST

Content-Type: application/xml

Send the some dummy xml to test

<card></card>

Response:

Status code : 409 Conflict (Which we given)

Observe Error details as Response:

<error>

<error-code>409</error-code>

<message>Card Already Issued(Using Case: 1)</message>

</error>

Observation:

This WebApplication approach (which contains 3-cases) will gives more readability about the code bcz we throwing exception which contains response as error details, but in previous approach error details also as like normal response so it will not gives readability to the developers whether it is exception response or normal response. But if we use WebApplication exception which allows us to build errors as response so that we can model a error details as response so that readability will be more bcz we are throwing exception (but not returning just as response) which contains error details as response.

**5. Practical difference between Web Services and RESTful services:**

1. We cannot model a REST Resource with 2-objects like Order and Address bcz body will contains only 1-XML so as we cannot sent multiple XML's as part of request body we cannot take multiple obj's as params but we can take Order and Address both in a single obj called as PurchaseOrder so that we can get both obj's in a single XML.

@Path("/order")

class OrderResource {

@POST

@Path(("/new")

@Consumes(MediaType.APPLICATION\_XML)

@Produces(MediaType.APPLICATION\_XML)

public Response newOrder(PurchaseOrder purchaseOrder) {

}

}

<purchaseOrder>

<order>

<orderId>929292<orderId>

<productName>LCD TV<productName>

<order>

<address>

....

</address>

< /purchaseOrder>

2. In Web Services we can use soap: faults to build the error response back to the consumer but in REST we communicate back to the client using status codes.

🡪Can we model a REST Resource which will takes 2-obj's as parameters?

Ans:

No, We cannot model REST Resource with obj's as parameters bcz we cannot send 2-xml's in a single body. So if we want 2-obj's then wrap those obj's in a single obj and model it. Similarly we cannot model using 2-Inputstream's or 2-char[] etc as 2-parameters rather we need to take only one param.

🡪What is the practical difference between Web Services and RESTful services that you have observe while working?

Ans:

In REST we can never model a Resource taking 2-parameters but in Web Services it can be possible.

🡪How can we communicate back to the Consumer or Client in case of WebServices and RESTful WebServices?

Ans:

In Web Services using <soap:faults> and in case of RESTful using Status codes.

**6. Exception Mapping or Mapping default exceptions:**

@Path("/card")

public class CardAgent {

// We have Bank Account but we don't have credit card but we are trying to get the credit card details with wrong cardNo which throws exception

@GET

@Consumes(MediaType.TEXT\_PLAIN)

@Produces(MediaType.APPLICATION\_XML)

@Path("/get/{card}")

public Response applyCard(@PathParam("card") String cardNo) {

try {

// call the service layer so that service layer will throw this exception actually

if (cardNo.equals("C2")) {

throw new CardNotFoundException();

}

} catch (CardNotFoundException e) {

throw new WebApplicationException(Response.status(Status.NOT\_FOUND)

.entity("<error><error-code>400</error-code><message>With Entered cardNo there is no CardDetails to display so plz apply for the Credit Card </message></error>").build());

}

return Response.ok().build();

}

}

@Path("/card-payer")

public class CardPayer {

// We have Bank Account but we don't have credit card but we are trying to pay the money for online shopping throw online via with wrong credit cardNo.

@GET

@Consumes(MediaType.TEXT\_PLAIN)

@Produces(MediaType.APPLICATION\_XML)

@Path("/pay/{card}")

public Response payAmount(@PathParam("card") String cardNo) {

try {

// service layer will checks the valid card or not and throws exception if not valid, if valid card then process the payment

if (cardNo.equals("C2")) {

throw new CardNotFoundException();

}

} catch (CardNotFoundException e) {

throw new WebApplicationException(Response.status(Status.NOT\_FOUND)

.entity("<error><error-code>400</error-code><message>With Entered cardNo there is no CardDetails to display so plz apply for the Credit Card </message></error>").build());

}

return Response.ok().build();

}

}

Here we have 2-requirements where we need to throw CardNotFoundException in the 2-Resource classes so we need to populate the Response with proper status codes in 2-Resource classes with same error details which is an duplication of building error response in 2-Resource classes.

So whenever we have an exception which we can use in multiple places we need to write try and catch in each and every Resource class and populate with same error details as Response which is duplication handling the exception, So intruder to avoid this duplication of populating Response with Status codes and error details in each and every Resource we can write a mapper and throw the Exception so that JAX-RS will takes care of identifying the Exception Mapper and populates Response with error details that we did in the mapper so that we are free form duplication of populating same error responses.

Similarly many applications have to deal with a multitude of exceptions thrown from application code and third-party frameworks. Relying on the underlying servlet container to handle the exception doesn’t give us much flexibility. Catching and then wrapping all these exceptions within WebApplicationException would become quite tedious. So avoid this alternatively, we can implement and register instances of javax.ws.rs.ext.ExceptionMapper that means we can write our ExceptionMapper by using ExceptionMapper that been provided by the JAX-RS and register with this so that these objects know how to map a thrown application exception to a Response object.

public interface ExceptionMapper<E extends Throwable> {

Response toResponse(E exception);

}

The default error handling for JAX-RS is that we can write an ExceptionMapper for these scenarios. For example, if we want to send back a different response to the client when JAXRS cannot find an @Produces match for an Accept header, you can write an ExceptionMapper for NotAcceptableException. This gives you complete control on how errors are handled by your application.

Similarly for example, one exception that is commonly thrown in Java Persistence API (JPA)–based database applications is javax.persistence.EntityNotFoundException. It is thrown when JPA cannot find a particular object in the database. Instead of writing code to handle this exception explicitly, we could write an ExceptionMapper to handle this exception for us. Let’s do that.

Let’s write our exception Mapper for EntityNotFoundException as follows.

@Provider

public class EntityNotFoundMapper implements

ExceptionMapper<EntityNotFoundException> {

public Response toResponse(EntityNotFoundException e) {

return Response.status(Response.Status.NOT\_FOUND).build();

}

}

Similarly we can write our Exception as CardNotFoundException and we can map our Exception with ExceptionMapper so that whenever we throw the CardNotFoundException JAX-RS Runtime will build the error response and takes that response and sendback to the client so we no need to duplicate the error with status codes in all the Resource classes, rather we need to just throw exception from our Resource class.

@Provider

public class CardNotFoundExceptionMapper implements

ExceptionMapper<CardNotFoundException> {

public Response toResponse(CardNotFoundException e) {

return Response.status(Response.Status.NOT\_FOUND).build();

}

}

Our ExceptionMapper implementation must be annotated with the @Provider annotation. This tells the JAX-RS runtime that it is a component. The class implementing the ExceptionMapper interface must provide the parameterized type of the ExceptionMapper. JAX-RS uses this generic type information to match up thrown exceptions to ExceptionMappers. Finally, the toResponse() method receives the thrown exception and creates a Response object that will be used to build the HTTP response.

JAX-RS supports exception inheritance as well. When an exception is thrown, JAX-RS will first try to find an ExceptionMapper for that exception’s type. If it cannot find one, it will look for a mapper that can handle the exception’s superclass. It will continue this process until there are no more super classes to match against.

Finally, ExceptionMappers are registered with the JAX-RS runtime using the deployment APIs using @Provider annotation.

So by using ExceptionMapper we can rewrite the above Resource class as follows without duplicating the error details in each and every Resource class. So if we use Exception Mapper we no need to write try and catch rather simply we can throw the exception so it will be received by the JAX-RS Runtime through the exception propagation/delegation principle so it looks for the exception mapper and sends details for the failure from the Mapper. If mapper is not there then it will sends default error messages, but sending default error messages is not recommended hence we need to use either WebApplicationException or ExceptionMapper and if we want to use same exception in multiple places then use ExceptionMapper otherwise we can use WebApplicationException.

Write Mappers:

@Provider

public class CardNotFoundExceptionMapper implements

ExceptionMapper<CardNotFoundException> {

@Override

public Response toResponse(CardNotFoundException cnfe) {

// NOT\_FOUND indicates 400 status code

return Response.status(Status.NOT\_FOUND)

.entity("<error><error-code>400</error-code><message>Entered card Not registered to get the details plz apply for new Card</message></error>").build();

}

}

@Provider

public class CardNotFoundRuntimeExceptionMapper implements

ExceptionMapper<CardNotFoundRuntimeException> {

@Override

public Response toResponse(CardNotFoundRuntimeException rte) {

// NOT\_FOUND indicates 400 status code

return Response.status(Status.NOT\_FOUND)

.entity("<error><error-code>400</error-code><message>Entered card Not registerd plz apply for new Debit Card</message></error>").build();

}

}

@Provider

public class WebApplicationExceptionMapper implements

ExceptionMapper<WebApplicationException> {

@Override

public Response toResponse(WebApplicationException rte) {

// NOT\_FOUND indicates 400 status code

return Response.status(Status.NOT\_FOUND)

.entity("<error><error-code>400</error-code><message>Entered card Not registerd plz apply for new Credit Card</message></error>").build();

}

}

@Path("/card")

public class CardAgent {

// We have Bank Account but we don't have credit card but we are trying to

// get the credit card

// details with wrong cardNo which throws exception

@GET

@Consumes(MediaType.TEXT\_PLAIN)

@Produces(MediaType.APPLICATION\_XML)

@Path("/get/{card}")

public Response applyCreditCard(@PathParam("card") String cardNo)

throws CardNotFoundException {

// call the service layer so that service layer will throw this

// exception actually

CardAgentDao dao = new CardAgentDao();

String details = null;

// Here we no need to do anything bcz exception will automatically

// propagates to the caller

details = dao.getCardDetails(cardNo);

details = "<cardNo>" + cardNo + "</cardNo>";

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

}

}

@Path("/card-payer")

public class CardPayer {

// We have Bank Account but we don't have credit/debit card but we are

// trying to pay the money for online

// shopping throw online via with wrong credit/debit cardNo

@GET

@Consumes(MediaType.TEXT\_PLAIN)

@Produces(MediaType.APPLICATION\_XML)

@Path("/credit/{card}")

public Response payAmountViaCreditCard(@PathParam("card") String cardNo) {

CardPayerDao dao = new CardPayerDao();

boolean response = dao.pay(cardNo);

// Approach-1

if (response == false) {

throw new WebApplicationException();

}

String payement = "<payment>success</payment>";

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

}

@GET

@Consumes(MediaType.TEXT\_PLAIN)

@Produces(MediaType.APPLICATION\_XML)

@Path("/debit/{card}")

public Response payAmountViaDebitCard(@PathParam("card") String cardNo) {

CardPayerDao dao = new CardPayerDao();

boolean response = dao.pay(cardNo);

if (response == false) {

throw new CardNotFoundRuntimeException(

"CardNotFound: Entered card not eligible for payment Plz apply for new Debit Card");

}

String payement = "<payment>success</payment>";

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

}

}

Access the application:

http://localhost:8083/3.4ExceptionMapperForCheckedUncheckedWebApplExceptions/rest/card/get/C2

Select GET

Content-Type: text/plain

Response:

<error>

<error-code>400</error-code>

<message>Entered card Not registerd to get the details plz apply for new Card</message>

</error>

http://localhost:8083/3.4ExceptionMapperForCheckedUncheckedWebApplExceptions/rest/card-payer/credit/C2

Select GET

Content-Type: text/plain

Response:

<error>

<error-code>400</error-code>

<message>Entered card Not registerd plz apply for new Credit Card</message>

</error>

http://localhost:8083/3.4ExceptionMapperForCheckedUncheckedWebApplExceptions/rest/card-payer/debit/C2

Select GET

Content-Type: text/plain

Response:

<error>

<error-code>400</error-code>

<message>Entered card Not registered plz apply for new Debit Card</message>

</error>

**7. Sub exception classes of WebApplicationException:**

JAX-RS 2.0 has added a nice exception hierarchy for various HTTP error conditions. So, instead of creating an instance of WebApplicationException and initializing it with a specific status code, we can use one of these exceptions instead. We can change our previous example to use javax.ws.rs.NotFoundException:

@Path("/customers")

public class CustomerResource {

@GET

@Path("{id}")

@Produces("application/xml")

public Customer getCustomer(@PathParam("id") int id) {

Customer cust = findCustomer(id);

if (cust == null) {

throw new NotFoundException());

}

return cust;

}

}

Like the other exceptions in the exception hierarchy, NotFoundException inherits from WebApplicationException. If we looked at the code, we can see that in its constructor it is initializing the status code to be 404.

The below given Table lists some other exceptions we can use for error conditions that are under the javax.ws.rs package.

Table given below is JAX-RS exception hierarchy.

**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

BadRequestException is used when the client sends something to the server that the server cannot interpret. The JAX-RS runtime will actually throw this exception in certain scenarios. The most obvious is when a PUT or POST request has submitted malformed XML or JSON that the MessageBodyReader fails to parse. JAX-RS will also throw this exception if it fails to convert a header or cookie value to the desired type.

For example:

@HeaderParam("Custom-Header") int header;

@CookieParam("myCookie") int cookie;

If the HTTP request’s Custom-Header value or the myCookie value cannot be parsed into an integer, BadRequestException is thrown.

NotAuthorizedException is used when you want to write your own authentication protocols. The 401 HTTP response code this exception represents requires you to send back a challenge header called WWW-Authenticate. This header is used to tell the client how it should authenticate with the server.

NotAuthorizedException has a few convenience constructors that make it easier to build this header automatically:

public NotAuthorizedException(Object challenge, Object... moreChallenges) {}

For example, if we wanted to tell the client that OAuth Bearer tokens are required for authentication, We would throw this exception:

throw new NotAuthorizedException("Bearer");

The client would receive this HTTP response:

HTTP/1.1 401 Not Authorized

WWW-Authenticate: Bearer

**ForbiddenException** is generally used when the client making the invocation does not have permission to access the resource it is invoking on. In Java EE land, this is usually because the authenticated client does not have the specific role mapping required.

**NotFoundException** is used when you want to tell the client that the resource it is requesting does not exist. There are also some error conditions where the JAX-RS runtime will throw this exception automatically. If the JAX-RS runtime fails to inject into an @PathParam, @QueryParam, or @MatrixParam, it will throw this exception. Like in the conditions discussed for BadRequestException, this can happen if you are trying to convert to a type the parameter value isn’t meant for.

**NotAllowedException** is used when the HTTP method the client is trying to invoke isn’t supported by the resource the client is accessing. The JAX-RS runtime will automatically throw this exception if there isn’t a JAX-RS method that matches the invoked HTTP method. RESTful Java with JAX - RS 2. 0 (Second Edition).

**NotAcceptableException** is used when the client is requesting a specific format through the Accept header. The JAX-RS runtime will automatically throw this exception if there is not a JAX-RS method with an @Produces annotation that is compatible with the client’s Accept header.

**NotSupportedException** is used when a client is posting a representation that the server does not understand. The JAX-RS runtime will automatically throw this exception if there is no JAX-RS method with an @Consumes annotation that matches the Content-Type of the posted entity.

**InternalServerErrorException** is a general-purpose error that is thrown by the server. For applications, you would throw this exception if you’ve reached an error condition that doesn’t really fit with the other HTTP error codes. The JAX-RS runtime throws this exception if a MessageBodyWriter fails or if there is an exception thrown from an ExceptionMapper.

**ServiceUnavailableException** is used when the server is temporarily unavailable or busy. In most cases, it is OK for the client to retry the request at a later time. The HTTP 503 status code is often sent with a Retry-After header. This header is a suggestion to the client when it might be OK to retry the request. Its value is in seconds or a formatted date string.

ServiceUnavailableException has a few convenience constructors to help with initializing this header:

public ServiceUnavailableException(Long retryAfter) {}

public ServiceUnavailableException(Date retryAfter) {}

Example:

@Path("/payer")

class CardPayer {

// We have Bank Account but we don't have credit card but we are trying to

// pay the money for online

// shopping throw online via with wrong credit cardNo

@GET

@Consumes(MediaType.TEXT\_PLAIN)

@Produces(MediaType.APPLICATION\_XML)

@Path("/card/{card}")

public Response getCardDetails(@PathParam("card") String cardNo) {

PayerDao dao = new PayerDao();

boolean response = dao.pay(cardNo);

// Comment one by one to test multiple approaches

// Approach-1

if (response == false) {

/\*\* NotFoundException has predefined status-code as 404. Note that we

\* didn't written Mapper bcz it will have predefined status codes

\* defined by JAX-RS API, if we want we can write mapper as well.

\*/

throw new NotFoundException();

}

// Approach-2

if (response == false) {

throw new NotFoundException(Response.status(Status.NOT\_FOUND)

.entity("<error>Entered Card Not Registered with us so plz apply for new Card</error>")

.build());

}

String payement = "<payment>success</payment>";

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

}

@PUT

@Consumes(MediaType.TEXT\_PLAIN)

@Produces(MediaType.APPLICATION\_XML)

@Path("/online/{userName}/{password}")

public Response payAmountOnlineBanking(

@PathParam("userName") String userName,

@PathParam("password") String password) {

PayerDao dao = new PayerDao();

boolean response = dao.payOnline(userName, password);

// Approach-1

if (response == false) {

// InternalServerErrorException has predefined status-code as 500

throw new InternalServerErrorException();

}

// Approach-2

if (response == false) {

// ServiceUnavailableException has predefined status-code as 503

// throw new ServiceUnavailableException(); // (or)

throw new ServiceUnavailableException(

Response.status(Status.SERVICE\_UNAVAILABLE)

.entity("<error>Currently this payment service is not available plz try after 24-Hours</error>").build());

}

// Approach-3

if (response == false) {

throw new ServiceUnavailableException("<error>Service is Unavailable </error>",24L);

}

String payement = "<payment>success</payment>";

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

}

}

Access the application:

Approach: 1

http://localhost:8083/3.5SubExceptionClassesOfWebApplExceptionAndMappers/rest/payer/card/C2

Select GET

Content-Type: text/plain

Response Status code:

Status code- 404 NotFound (We thrown the exception as NotFoundException that is the reason we are getting 404)

Approach: 2

http://localhost:8083/3.5SubExceptionClassesOfWebApplExceptionAndMappers/rest/payer/card/C2

Select GET

Content-Type: text/plain

Response Status code:

Status code- 404 NotFound (We thrown the exception as NotFoundException that is the reason we are getting 404)

Response Body:

<error>Entered Card Not Registered with us so plz apply for new Card</error>

Approach: 1

http://localhost:8083/3.5SubExceptionClassesOfWebApplExceptionAndMappers/rest/payer/online/dbreddy/dbreddy

Select PUT

Content-Type: text/plain

Response Status code:

Status Code-500 Internal Server (Bcz we thrown InternalServerException)

Approach: 2

http://localhost:8083/3.5SubExceptionClassesOfWebApplExceptionAndMappers/rest/payer/online/dbreddy/dbreddy

Select PUT

Content-Type: text/plain

Response Status code-503 Service Unavailable (bcz we thrown ServiceUnavailableException)

Response Body:

<error>Currently this payment service is not available plz try after 24-Hours</error>

Approach: 3

http://localhost:8083/3.5SubExceptionClassesOfWebApplExceptionAndMappers/rest/payer/online/dbreddy/dbreddy

Select PUT

Content-Type: text/plain

Response Status code-503 Service Unavailable (bcz we thrown ServiceUnavailableException)

Response Headers

Retry-After: 24

**8. JAX-RS exception hierarchy or Complete Exception Flow in JAX-RS API and Overriding Error Details:**

Our application code is allowed to throw any checked (classes extending java.lang.Exception) or unchecked (classes extending java.lang.RuntimeException) exceptions we want.

Thrown exceptions are handled by the JAX-RS runtime if we have registered an exception mapper. Exception mappers can convert an exception to an HTTP response. If the thrown exception is not handled by a mapper, it is propagated and handled by the container (i.e., servlet) bcz JAX-RS is running within Web-Container/Servlet-Container.

JAX-RS also provides the javax.ws.rs.WebApplicationException. This can be thrown by application code and automatically processed by JAX-RS without having to write an explicit mapper (If we want we can write mapper for WebApplicationException as well and if we want we can send our error response by throwing WebApplicationException with our error details so that it will not uses Default ExceptionMapper).

Our ExceptionMapper implementation must be annotated with the @Provider annotation. This tells the JAX-RS runtime that it is a component. The class implementing the ExceptionMapper interface must provide the parameterized type of the ExceptionMapper. JAX-RS uses this generic type information to match up thrown exceptions to ExceptionMappers. Finally, the toResponse() method receives the thrown exception and creates a Response object that will be used to build the HTTP response.

JAX-RS supports exception inheritance as well. When an exception is thrown, JAX-RS will first try to find an ExceptionMapper for that exception’s type. If it cannot find one, it will look for a mapper that can handle the exception’s superclass. It will continue this process until there are no more super classes to match against.

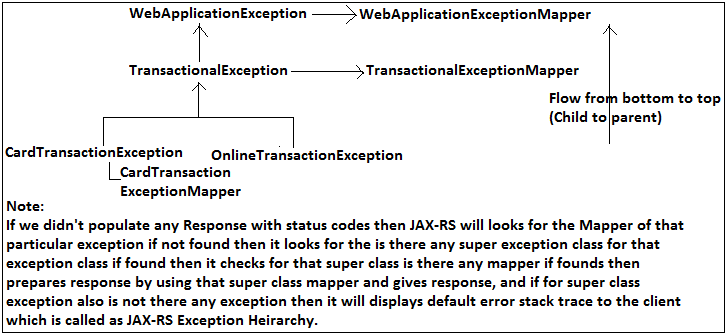
**Internal working Flow:**

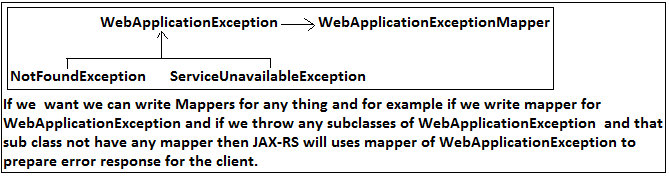
If we throw WebApplicationException with populated response then it will directly prepares response and gives to the client. If we directly throw WebApplicationException then it looks for is there any mapper for WebApplicationException or not if founds then uses that mapper to prepare response by calling toResponse() on that mapper.

If we throw any sub classes of WebApplicationException exception like NotFoundException or ServiceUnavailableException (these will have predefined status codes bcz these are implements from ClientErrorException which has logic to populate status codes pre-definably) and if we populate any exception body then it prepares response and gives to the client. If we didn't populate any data as part of the body and if we throw directly then it looks for the mapper of that WebApplicationException subclass if not founds then it looks for the mapper of WebApplicationException if founds then populates response body from that WebApplicationException mappe, if mapper is not found then prepares response with that subclass exception ststus code bcz it have pre-build status codes and body with error stack trace which is called as Built-in JAX-RS Exception Hierarchy or Built-in Parent-Child Exception Hierarchy.

If we write any user exceptions also same as above. For example we written user exception and we didn't populate any Response status codes and details then JAX-RS will looks for the Mapper of that particular exception if not found then it looks for is there any super exception class for that exception class if found then it checks for that super class is there any mapper or not if mapper founds then prepares response by using that super class mapper and gives response, and if for that super class exception also mapper is not available then it will displays default error stack trace to the client which is called as User defined JAX-RS Exception Hierarchy or user defined Parent-Child Exception Hierarchy.

Let us consider following exception hierarchy





Here we have 2-mappers one for WebApplicationException as WebApplicationExceptionMapper and one for TransactionalException as TransactionalExceptionMapper and TransactionalException is a child of WebApplicationException and inturn TransactionalException has 2-child exceptions like CardTransactionException and OnlineTransactionException then if we throw OnlineTransactionException direclty then it will looks for mapper for itself if not found then it looks for mapper of TransactionalException but not mapper of WebApplicationException bcz nearest parent is for OnlineTransactionException is TransactionalException hence TransactionalExceptionMapper will be used by the JAX-RS to build error response by calling toResponse() from the mapper, if TransactionalExceptionMapper is not there then it looks for WebApplicationExceptionMapper which is called as JAX-RS Exception hierarchy.

**Conclusions (Note):**

1. Always it is recommended to define User Exception as UCE's using RuntimeException and in RESTful to define Unchecked Exceptions it highly recommended to use WebApplicationException(UCE) so that we easily implement the Exceptions bcz WebApplicationException is UCE and it has more constructors to define so we can model exceptions easily if we use this.

2. If we want we can write Mappers for anything and for example if we write mapper for WebApplicationException and if we throw any subclasses of WebApplicationException and that sub class not have any mapper then JAX-RS will uses mapper of WebApplicationException to prepare error response for the client.

3. If we want we can write Mappers for any exceptions.

4. Overriding Mapper exception details with our error response details:

If we written mapper but we don't wanted to use Mapper to populate the response for that specific exception then we can populate complete response in that particular exception so that JAX-RS will not use for that one bcz we already populated details using Response so it will not calls Mapper even it is available which is called exception error details overriding.

public class TransactionalException extends WebApplicationException {

// write constructors

}

public class CardTransactionException extends TransactionalException {

// write constructors

}

public class OnlineTransactionException extends TransactionalException {

// write constructors

}

@Provider

public class WebApplicationExceptionMapper implements

ExceptionMapper<WebApplicationException> {

}

@Provider

public class TransactionalExceptionMapper implements

ExceptionMapper<TransactionalException> {

}

@Provider

public class CardTransactionExceptionMapper implements

ExceptionMapper<CardTransactionException> {

}

@Path("/payer")

class CardPayer {

// We have Bank Account but we don't have credit card but we are trying to

// pay the money for online

// shopping throw online via with wrong credit cardNo

// Resource Method-1

// http://localhost:8083/3.6JAX-RSExceptionHeirarchy/rest/payer/card/C2

@PUT

@Consumes(MediaType.TEXT\_PLAIN)

@Produces(MediaType.APPLICATION\_XML)

@Path("/card/{card}")

public Response payViaCard(@PathParam("card") String cardNo) {

PayerDao dao = new PayerDao();

boolean response = dao.pay(cardNo);

if (response == false) {

// Comment one to test it

// Case-1

// throw new CardTransactionException();

// Case-2

throw new CardTransactionException(Response.status(Status.NOT\_FOUND)

.entity("<error>Entered Card in not registered with us for payment plz apply for new Card</error>").build());

}

String payement = "<payment>success</payment>";

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

}

// Resource Method-2

@PUT

@Consumes(MediaType.TEXT\_PLAIN)

@Produces(MediaType.APPLICATION\_XML)

@Path("/online/{userName}/{password}")

public Response payViaOnlineBanking(

@PathParam("userName") String userName,

@PathParam("password") String password) {

PayerDao dao = new PayerDao();

boolean response = dao.payOnline(userName, password);

if (response == false) {

throw new OnlineTransactionException();

}

String payement = "<payment>success</payment>";

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

}

// Resource Method-3

// http://localhost:8083/3.6JAX-RSExceptionHeirarchy/rest/payer/wallet/W101

@PUT

@Consumes(MediaType.TEXT\_PLAIN)

@Produces(MediaType.APPLICATION\_XML)

@Path("/wallet/{walletNo}")

public Response paytmWallet(@PathParam("walletNo") String walletNo) {

PayerDao dao = new PayerDao();

boolean response = dao.walletPayment(walletNo);

if (response == false) {

throw new NotFoundException(Response.status(Status.NOT\_FOUND)

.entity("<error>Invalid Wallet Id plz check walletId</error>")

.build());

}

String payement = "<payment>success</payment>";

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

}

}

Access the application:

Resource Method-1:

Case: 1 (Comment case: 2 in programme)

http://localhost:8083/3.6JAX-RSExceptionHeirarchyAndOverridingErrorDetails/rest/payer/card/C2

Select PUT

Content-Type: text/plain

Response:

Response Status code: 503 (Service Unavailable)

Response Body:

<error>(CardTransactionExceptionMapper) Try After 24-hours</error>

Case: 2 (Comment case: 1 in programme)

http://localhost:8083/3.6JAX-RSExceptionHeirarchyAndOverridingErrorDetails/rest/payer/card/C2

Select PUT

Content-Type: text/plain

Response:

Response Status code: 404 (We given)

<error>Entered Card in not registered with us for payment plz apply for new Card</error>

Resource Method-2:

http://localhost:8083/3.6JAX-RSExceptionHeirarchyAndOverridingErrorDetails

/rest/payer/online/dbreddy/dbreddy

Select PUT

Content-Type: text/plain

Status Code-503 (Service Unavailable)

Content-Type: text/plain

Response:

<error>Try After 24-hours</error>

Resource Method-3

http://localhost:8083/3.6JAX-RSExceptionHeirarchyAndOverridingErrorDetails/rest/payer/wallet/W101

Select PUT

Content-Type: text/plain

Response:

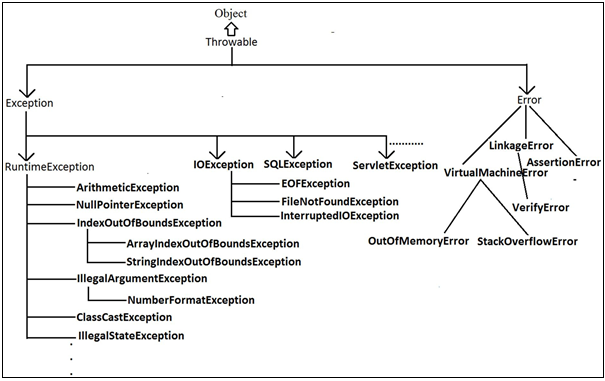
Status Code-404 (NotFound)

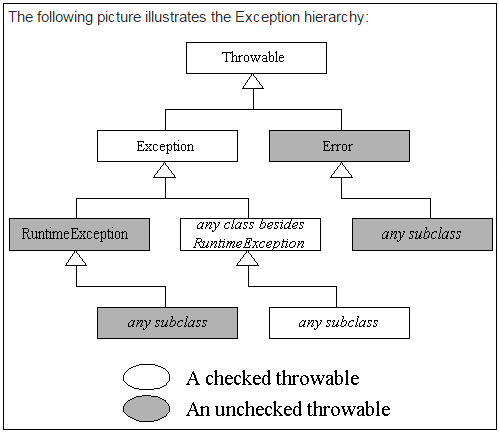
Response Body:

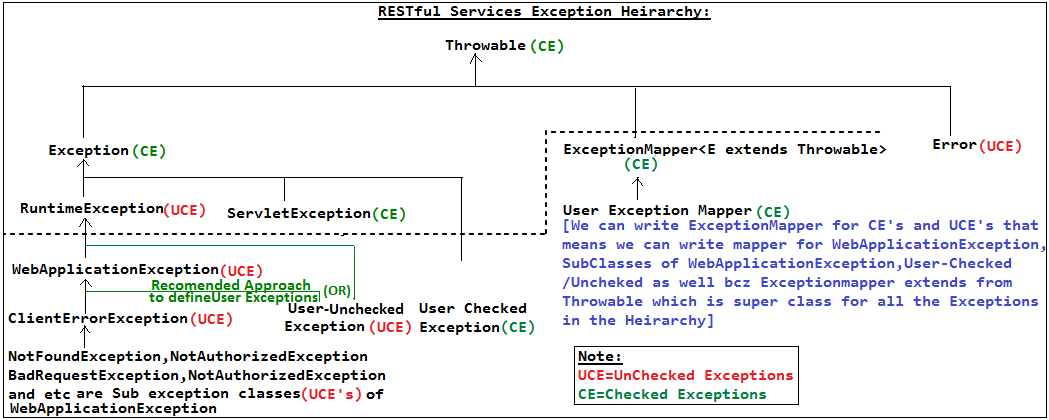
<error>Invalid Wallet Id plz check walletId</error>

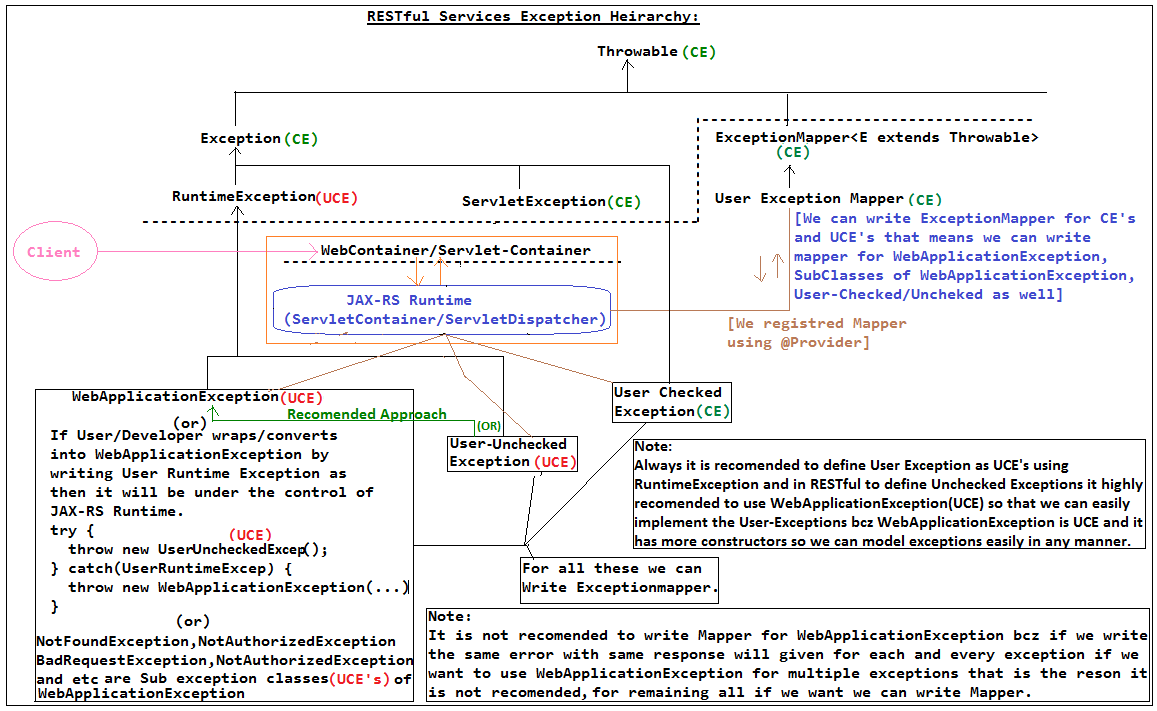
From this Resource Method-3 response we can observe that we can override error details for any exception we want even though the mapper is available which is called as "exception error details overriding".

**JAX-RS Complete Exception Flow project Diagram:**









🡪How do you propagate the Exception or how do manage the Exceptions in case of RESTful services?

Ans: When our Resource is encountering an Exception we just will not propagte that exception back to the client rather we need to build our Response obj populating with status codes to let the client to identify whether response getting as part of the body is success or failure with an proper error message to communicate back easily. To achieve this we are relaying on the Http Status codes bcz client can be anything (java/.net) so that they can get this failure from the status codes in an interoperable manner.

**9. RESTful as DB-REST-Wrapper**

|-Advantages of REST wrappers on top of the DB

|-Use Cases

**RESTful as DB-REST-Wrapper:**

For example 3-Appl wants to use data to access difficult to understand schema that's where we can even use REST API there so that all the applications can read the data from the REST Resource so that even though schema changes all the 3-appl no need to worry bcz they are working with REST API so that no effect on the applications.

That means REST will used as wrappers on top of the DB's so that applications no need to worry about schema and other things bcz REST taking care everything so if we want any data or to insert the data we will talk to the REST only.

We may have a doubt if we wanted to talk to the REST we need to connect over the network and in case of DB's also we need to connect to the network only bcz in real application DB will be some other system so any how we are connecting to the DB to perform the operations rather than connecting to the DB if we connect to the REST that is there on top of the DB which is called as "DB-REST-Wrapper" so that it will takes care of connecting to the DB and sends back the data back to the applications.

The REST and DB's both must will runs in the same system or intranet systems bcz REST is wrapper on top of the DB, other we need to more appl which will connects REST to DB which will kills the application performance. But if we connect to the REST it will connects DB easily bcz both are in same system so that as like normal with 1-network connection only we will get the data from the DB using REST hence there will not be any performance issues as well.

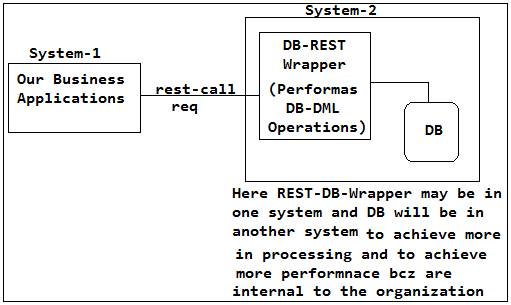
**Advantages of REST wrappers on top of the DB:**

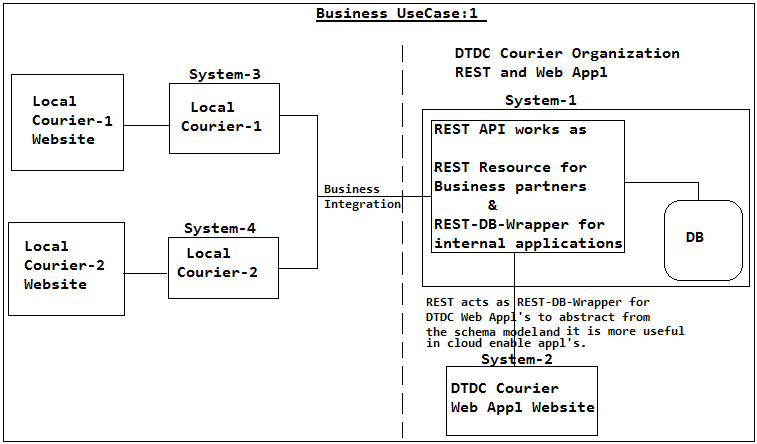
1. REST will abstract the schema model from the application point of view. so that any no.of applications can communicate with same schema model if they have same type of business applications for multiple applications or for per application bcz it free from DB logic schema.

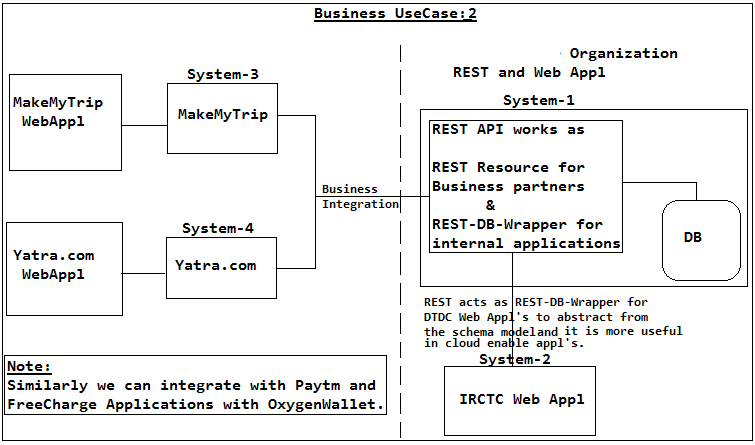
2. The cloud integrations means REST Wrappers only to connect with DB.

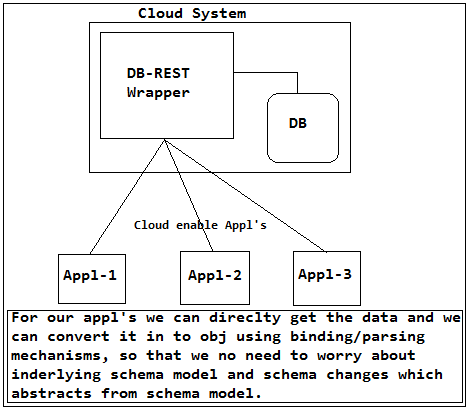
3. Every application or applications build REST to connect with DB bcz anyhow actual application is going to communicate with DB over the network and tighly coupled to the schema but if we go for REST-DB-Wrapper we will connect to the REST which will in turn connects to the DB bcz both are running in same system so with same one network connection itself we can get the data from DB and advantage is our application free for changes in schema model of the DB and persistence related logic of frame work logic.

**Use Cases for REST Wrappers:**









**10. Wrapping Up:**

In this chapter, we learned that JAX-RS has default response codes for both success and error conditions. For more complex responses, your JAX-RS resource methods can return javax.ws.rs.core.Response objects. JAX-RS has a few exception utilities. We can throw instances of javax.ws.rs.WebApplicationException or let the underlying servlet container handle the exception. Or, you can write an ExceptionMapper that can map a particular exception to an HTTP response.