Restful Web Service

Table of Contents

[1. Introduction 5](#_Toc517647798)

[2. RESTFUL API Design guide lines 5](#_Toc517647799)

[2.1 Should have resource based address 5](#_Toc517647800)

[2.2 Triggers based on HTTP Methods and URI 5](#_Toc517647801)

[2.3 Well Defined Metadata 5](#_Toc517647802)

[3. Resource URI designing 6](#_Toc517647803)

[3.1 Designing Resource Based URI’s 6](#_Toc517647804)

[4. Collection URI’s 6](#_Toc517647805)

[4.1. Instance Resource URI 7](#_Toc517647806)

[4.2. Collection resource URI 7](#_Toc517647807)

[5. Filtering and Pagination 7](#_Toc517647808)

[6. HTTP Methods 7](#_Toc517647809)

[7. Idempotent Methods 8](#_Toc517647810)

[8. REST Response 9](#_Toc517647811)

[9. HATEOAS – Hyper Media as the Engine of Application State 10](#_Toc517647812)

[10. Richardson Maturity Model 10](#_Toc517647813)

[11. JAX RS 11](#_Toc517647814)

[12. Environment Setup 11](#_Toc517647815)

[13. Understanding application structure 12](#_Toc517647816)

[14. Creating Resource 13](#_Toc517647817)

[15. Returning XML response 15](#_Toc517647818)

[16. HTTP methods list 17](#_Toc517647819)

[17. Building Service Stubs 17](#_Toc517647820)

[18. Accessing Path Parameters 20](#_Toc517647821)

[19. Returning JSON Responses 21](#_Toc517647822)

[20. Implementing POST methods 22](#_Toc517647823)

[21. Implementing Profile Resource 23](#_Toc517647824)

[22. Pagination and Filtering 25](#_Toc517647825)

[23. Param Annotation 27](#_Toc517647826)

[23.1. Matrix Param 27](#_Toc517647827)

[23.2. Header Param 27](#_Toc517647828)

[23.3. Cookie Param 28](#_Toc517647829)

[24. Using Context and Bean param annotation 28](#_Toc517647830)

[25. Implementing Sub Resources 30](#_Toc517647831)

[26. Sending Status code and Headers 34](#_Toc517647832)

[27. Handling Exceptions 34](#_Toc517647833)

[28. Using Web Application Exception 36](#_Toc517647834)

[29. HATEOAS 37](#_Toc517647835)

[30. HATEOAS Continuation 39](#_Toc517647836)

[31. Content Negotiation 40](#_Toc517647837)

# Introduction

The RESTFUL web service is the acronym for “*Representational State Transfer*”. In Brief the calling system sends out an HTTP request for which the RESTFUL web service returns data to the calling system which houses the implementation to render a view based on the data received.

Basically RESTFUL web service can transmit data in terms of XML, JSON (Java Script Object Notation), text and HTML or CSS.

***Note***: by definition a *web service* is an application that is available for programmatic use.

# RESTFUL API Design guide lines

REST API is based on the fundamental principles of HTTP hence while designing any RESFUL API the following a set of guide line makes the application must user friendly.

## Should have resource based address

Let’s take an example of URI that is used to get weather information.

URI: weatherapp.com/weatherlookup.do?zipcode=12345

The above mentioned URI clearly states that the method do is being called with the parameter zipcode of value “12345” which is not an effective resource based address rather it is an action based URI.

Example of resource based URI’s:

1. Weatherapp.com/zipcode/12345
2. Weatherapp.com/country/brazil

The first URI indicates as if the value is being retrieved but the fact is that implementation does a look up to get the value and the second URI does a weather lookup by country name.

## Triggers based on HTTP Methods and URI

The various method of HTTP include (GET, PUT, DELETE, POST), so the implementation of the RESTFUL web service should include business service methods based on the URI and the HTTP method through which the URI is being accessed.

## Well Defined Metadata

The Meta data refers to the response header and the status code; the following are the commonly used status code to represent the outcome of the web service call.

1. Code 200 – Successful Outcome
2. Code 550 – Error Scenario (including business validation)
3. Code 404 - If the requested resource is not available or does not exist.

The format in which the data is about to be exchanged will be defined by the “Component-type”. Basically it represents the format in which the data is requested by the calling system. Following are the few of those values.

1. Text/xml
2. Application/JSON (Java Script Object notation).

If a given APIP of the RESTFUL web service can provide data in multiple formats say text as well as JSON, Then based on the “Component-Type” the calling system can perform “Content Negotiation” specifying the type of the content expected from the web service.

# Resource URI designing

Let’s consider a small Messenger application that has the following entities.

1. Post on messages.
2. Comment on Messages.
3. Like and Share on Messages.
4. User Profile (with the option to create and update)

## Designing Resource Based URI’s

URI for accessing processing profile: */profiles/{profileId}*

URI for accessing messages : /messages/{messageId}

It can be observed that the entity name is a noun and plural which denote many.

In the event if we are to deign URI for comments it can be done as “*/comments/{id}”* but under few scenario the relation that exists between the resource has to be considered say in our case one message can have many comments hence the URI can be designed as *“/messages/{messageId}/comments/{commentsId}”*.

Similarly for likes: */messages/{messageId}/likes/{likeId}*

For shares: */messages/{messageId}/Share/{shareId}*

# Collection URI’s

There are basically 2 types of URI in REST Web service.

## Instance Resource URI

These URI often refer to specific object or entity.

* *Example 1:* An URI to access message with id “2”.

URI : /messages/{2}

* *Example 2:* URI to access 2nd comment of message having id 2.

URI: /messages/{2}/comments/{2}

## Collection resource URI

These are URI’s used to access a generic object or entity. It can also be observed that the name of such resource is a plural.

* *Example 1:* URI to access all the messages

/messages

* *Example 2:* All the comments for the message with id 2

/messages/{2}/comments

# Filtering and Pagination

In the event say a client application requests to retrieve all the messages the amount of data might be of very large volume. In such scenario’s we have to implement features that allows client to perform pagination or filtering. i.e it enables query parameters.

* *Example 1:* If the client wants to retrieve message starting from id 30 followed by the next 10 messages.

URI: /messages?offset=30&limit=10

* *Example 2:* Client initiates a request to get the entire set of messages during the year 2014 starting with message id 20 followed by the next 30 messages.

URI: /messages?year=2014&offset=20&limit=30

# HTTP Methods

In a RESFUL web service it is advisable to define controller methods to trigger, based on both the HTTP method and the respective URI’s here are a couple of example for common scenarios.

The most common HTTP methods are GET, PUT, POST, DELETE.

/getProducts.do?id=10 /deleteOrders.do?id=10

Through resource based URI the above mentioned URI will be defined as follows.

***GET Request*** /products/10 ***DELETE Request*** /Orders/10

* To Retrieve message for the id 10

***GET Request*** /messages/10

* To Update message with id 10

***PUT Request*** /messages/10

The PUT method will deliver the content to be updated through the message body to the server application and yes we can use POST as well but it is much preferred to us PUT over POST to update message or content.

* To Delete message with id 10

***DELETE Request*** /messages/10

* To create a new message – During the creation of the message the client would not have the message id as the implementation to generate the id will be contained within the server side application. In order to create message we would use the ***POST*** method to the collection resource URI “messages” as “/messages/{message}”.

|  |  |  |
| --- | --- | --- |
| ***Http Method*** | ***URI*** | ***Function*** |
| GET Request | /messages | Retrieve all the messages |
| DELETE Request | /messges/10/comments | Delete all the comments for the message with id 10 |
| GET Request | /messages/10/comments | Retrieve all the comments for the message with id 10 |
| POST Request | /messages/10/comments | Create a new list of comments overriding the existing oe for message with id 10 |

# Idempotent Methods

The basic methods of HTTP are post, delete, get and put. These methods are broadly classified into repeatable and non-repeatable methods.

***Repeatable Methods (Idempotent):*** When multiple calls of such method are made they don’t have any harmful impact on the server side.

Repeatable Methods in HTTP: Get, Put, Delete

For example if the delete method is called upon a message multiple times “/message/1” the first call will delete the message but the further calls have not impact on the system as the message does not exist.

Let’s consider a GET request, irrespective of the number of calls the end system juts retrieves and returns the data back to the calling system. In such cases during the first fetch the data will be cached and any further calls would get the response from the cache.

*Format:* GET /messages/10

***Non Repeatable Methods (Non Idempotent):*** Multiple calls of such method have a harmful impact on the server side. Say for example duplicate records get created.

That is if there are multiple post on the same message the business layer will end creating a new ID for every post which has an impact on the system resource. Hence in real time scenario when we submit any form say multiple times the browser prompts user to validate if the second submission is legit.

Non Repeatable Methods: Multiple calls of these method

# REST Response

A RESTFUL web service can return multiple responses based on the requirement of the calling system (JSON or XML).

When a response is sent from a RESTFUL web service the response header contains the metadata about the response such as date, message length, and content type.

The content type indicates the format of the response; the calling system can read the header and choose the appropriate parser to read the message response.

As it is a programmatic call to a web service it is advisable to make use of the status code to represent the outcome. There are majorly 5 categories of the messages which are determined by the first digit of the status code.

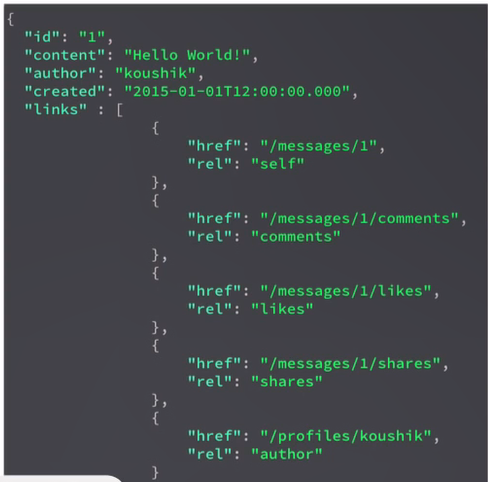
* 1\*\* - Informational Messages.
* 2\*\*- Successful outcome
* 3\*\* - Redirection Messages.
* 4\*\*- Client side error.
* 5\*\* -Server Side Error.

EG: 404 Page Not found – If client is accessing a resource that does not exist.

# HATEOAS – Hyper Media as the Engine of Application State

Whenever a request is being sent to the RESTFUL web service the response sent back to the calling system will have the related URI’s.

For example: A Message call say, GET /messages/1 would return the following response:



***HATEOAS*** – is a way to provide the links to resource URI in the API response so that the client does not have to deal with URI construction.

In the absence of the above mentioned context the client basically have to build the URI’s in order to communicate with the RESTFUL web service and just to make it convenient we embed the related resource URI’s I the response of the API call.

# Richardson Maturity Model

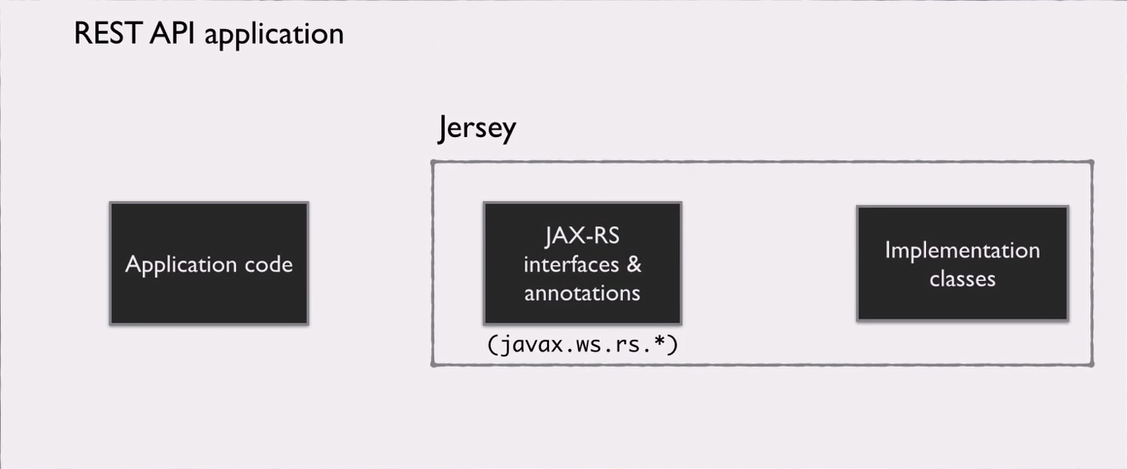
The Richardson maturity model can be used to determine if any given application is completely or partially restful. Here are the 4 levels of the model.

1. ***Level 0***: if the web service does the communication by means of POX (Plain Old XML) the application can be defined under level zero which means the web service is not restful.
2. ***Level 1***: if the web service implements resource based URI’s. In other words there exists a URI for all the available resource of a web service.
3. ***Level 2***: With the implementation using HTTP methods such as (GET, POST, DELET, PUT) and the use of right idempotent concepts the application falls under level 2.
4. ***Level 3***: Making use of HATEOAS, that is providing the list of URI’s in the response to an API call makes an application to be categorised level 3.

# JAX RS

JAX RS is the standard of specification which contains interfaces and annotations to create a RESTFUL web service. There are various libraries that implement the JAX RS interface like JERSEY, REST EASY.

The implementations are interchangeable and will not have any compatibility issue as all the libraries implement the same JAX RS and also every library comes with a copy of the JAXRS interface.



# Environment Setup

The JERSEY library is provided as part of the maven which is a project management utility which takes care of Deploying the application, dependency and life cycle management.

When we create a new MAVEN project we can choose an archetype which is basically a base version of the intended project upon which we can start building the required changes. Here is the archetype used in the further topics.

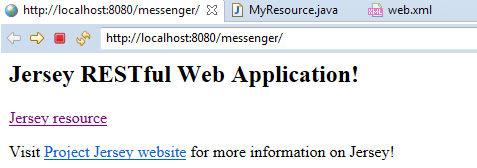


Group Id: org.glassfish.jersey.archetypes

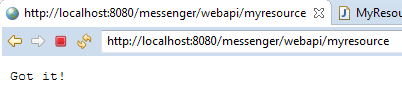
Artefact Id: jersey-quickstart-webapp

# Understanding application structure

Here is the default page that gets rendered when the maven project which was derived from the archetype is deployed on the server.



*The link Jersey resource leads to the following output.*

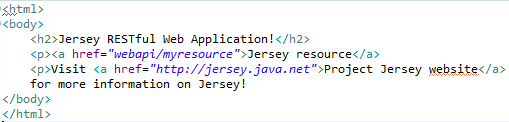


Working mechanism of the above mentioned view – When the URI “http:localhost//8080/messenger” is called upon, the execution lands up on the web.xml file which contains the URL mapping mechanism which invokes the servlet associated to the URL and that in turn invokes the respective class which loads the default file “index.jsp”.

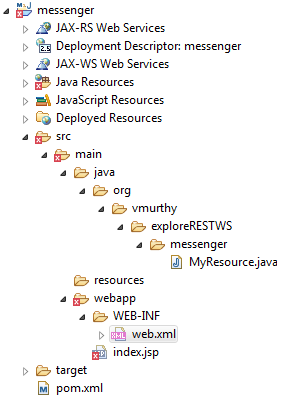
File Name: Web.xml



File Name: index.jsp



Picture: Project folder Structure



# Creating Resource

Whenever a request is received JERSEY needs to be informed of the class to which the request has to be passed on to. In order to do so the configuration is done in the web.xml file. The “INIT-PARAM”

From the web.xml file JERSEY identifies the class that has to triggered when the URI **“/webapi/\***”. Any other extension to the above mentioned URI is being called say “**webapi/messages**”, JERSEY needs to be configured on the respective target class to which the request has to be routed. This is done by mentioning the package that incorporates all the controller classes. At the start of the application JERSEY scans the predefined package and frames the routing information for any given request.

INIT PARAM Snippet :

<init-param>

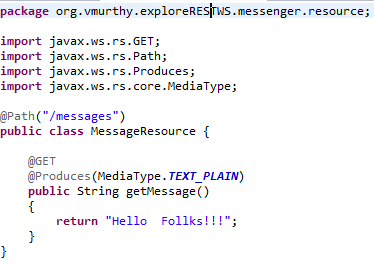
<param-name>jersey.config.server.provider.packages</param-name>

<param-value>org.vmurthy.exploreRESTWS.messenger</param-value>

</init-param>

The controller class should be incorporated in the package mentioned in the “init-param” for jersey to perform the mapping of the request to the controller class.

Below is the snapshot of the class which creates a new resource ***URI***: <http://localhost:8080/messenger/webapi/messages>.



Description of Annotations used:

1. @Path defines the URI against which the class has to be triggered.
2. @GET represents the http method against which the method will be executed. Similarly we can have a method to be triggered for post request with the annotation @POST.
3. @Produces – indicates the type of object returned by any given method. This is defined by the attribute “MediaType.PLAIN\_TEXT”.

# Returning XML response

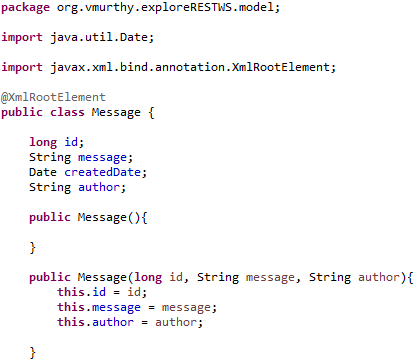
The above mentioned example will be enhanced with a model and service class wherein the Model class will define the structure of the message and the service defines the method through which the list of message will be retrieved.

It is mandatory to have a no argument constructor in any given user defined modem class.

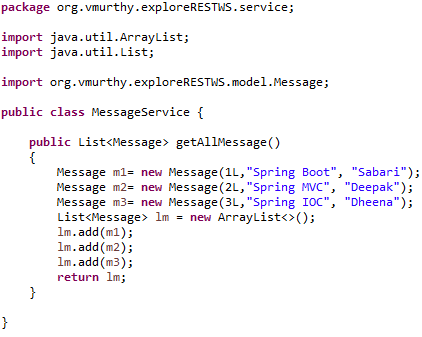
In order to return the response in XML format the @Produces annotation has to be updated with the value “**MediaType.APPLICATION\_XML**”. By doing so we are requesting JERSY to convert the list of message object to XML format.

Since JERSEY does not have the functionality to perform the conversion by itself it internally calls the JAXB to perform the conversion which requires the model class to be marked with the “@***XmlRootElement***” annotation.

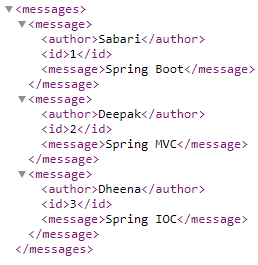
File 1: *Message.java*



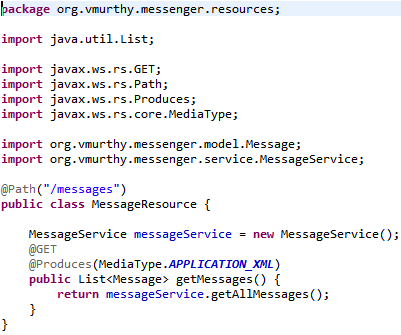
File 2: *MessageService.java*



Output:



File 3: MessageResource.java



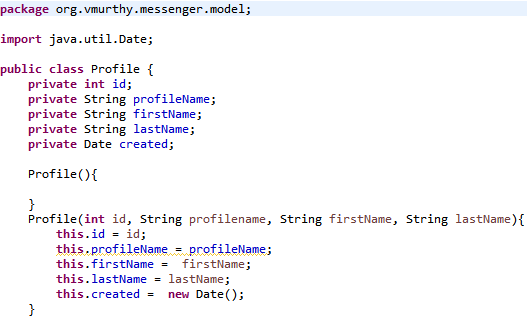
# HTTP methods list

1. GET
2. POST
3. PUT
4. PATCH
5. DELETE
6. COPY
7. HEAD
8. OPTIONS
9. LINK
10. UNLINK
11. PURGE
12. LOCK
13. UNLOCK
14. VIEW

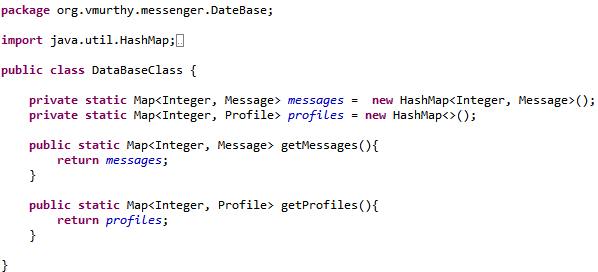
# Building Service Stubs

Restructuring the MessageService class to provide more features like update, delete, insert and retrieve operations. Since the application will not be interacting with any database, we will be using in memory objects to store and retrieve objects (implementation is provided in Database Class).

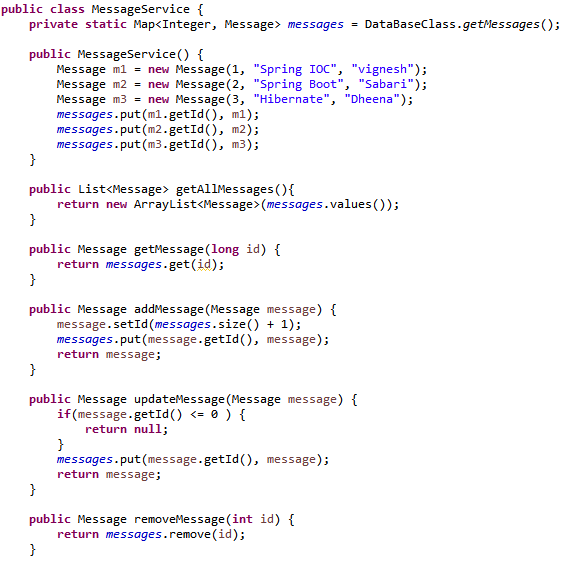
File 1: Profile.java



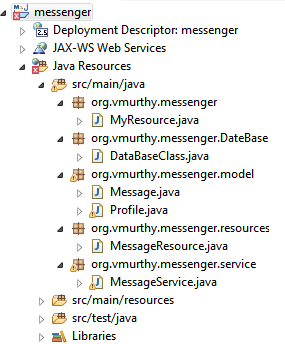
File 2: DataBaseClass.java



File 3: MessageService.java



Project Structure:



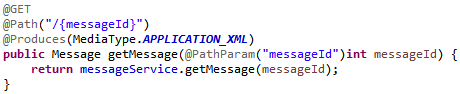
# Accessing Path Parameters

The MessageResource class will be enhanced to retrieve a message based on a message id. For example, the call to the below mentioned URL with id as say 1 should return the message with the respective id.

URL: http://localhost:8080/messenger/webapi/messages/**id**

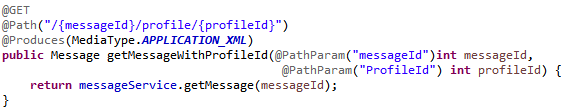
The annotation @PathParam(“ParameterName”) annotation does the mapping of the value from the incoming URL to the parameter of the method and also the conversion of the path parameter to the type of the method parameter is implicit.

Example 1: URL: <http://localhost:8080/messenger/webapi/messages/1>



Example 2: URL with multiple path parameters

URL: <http://localhost:8080/messenger/webapi/messages/1/profile/2>



File 1: MessageReource.java



Note: The following syntax is also allowed @PathParam(“*RegularExpression*”)

# Returning JSON Responses

To modify the return type of the web methods to JSON the @Produces annotation should be initialized with value as MediaType.APPLICATION\_JSON. This change just instructs jersey about the return type and the process of converting the message object to JSON should be done by an external entity.

The MessageBodyWriter does the conversion of object to JSON and the MessageBoidyReader does the conversion of JSON to object. But both MesssageBodyReader and MessageBodyWriter are interfaces and there is various implementation for the same. In the following example, we make use of “moxy” do perform the conversion.

For incorporating “MOXY” to the project a dependency needs to be added in the pom.xml file.

File 1: pom.xml



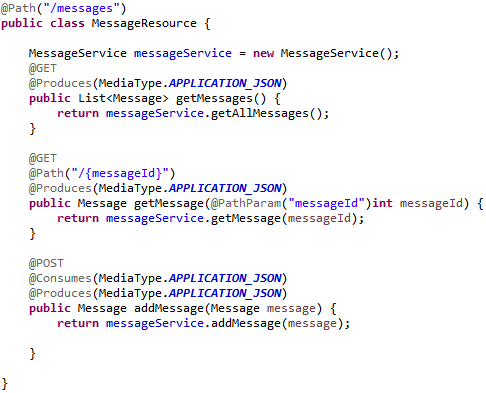
File 2: MessageResource.java



# Implementing POST methods

The implementation is a straight forward approach with the @POST, @Consumes(MediaType.APPLICATION\_JSON),@Produces(MediaType.APPLICATION\_JSON), the message body that is being sent gets formatted to a “Message” object implicitly through the “*MessageBodyReader*” implementation. the programmer need to make sure that the corresponding web method has a parameter to consume the incoming object.

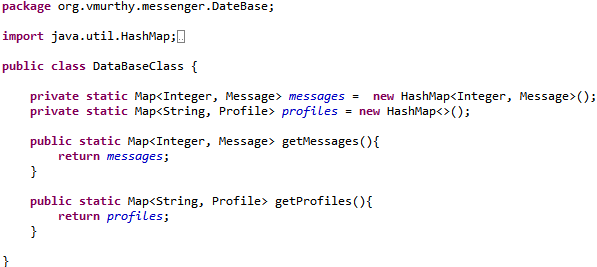
File 1: MessageReource.java



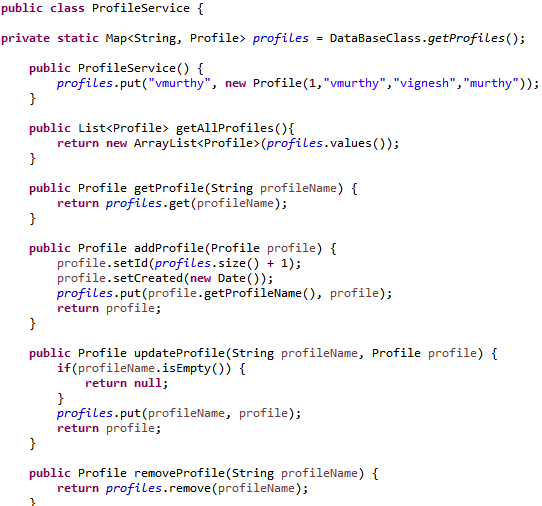
# Implementing Profile Resource

A new stream of features will be provided to create, update, delete and retrieve all the profiles from the in-memory objects.

File 1: DataBaseClass.java



File 2: ProfileService.java



File 3: ProfileResource.java



# Pagination and Filtering

Pagination is the process of defining the size of the result being presented in each page. For example, the user may be looking for the first 20 records starting from index 5.

URI for Pagination: http:localhost:8080/messenger/webapi/messages?start=5&size=20

Message Signature: getAllMessages(@QueryParam(“year”)int year)

Filtering is a way of defining condition based on which the result needs to be listed. Say, all the messages from the year 2018.

URI for Filtering: http:locahost:8080/messenger/webapi/messages?year=2018

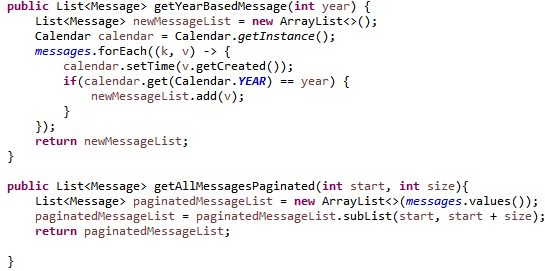
Message Signature: getAllMessage(@QueryParam(“start”)int start, @QueryParam(“size”) int size)

Even though we have the same URI path (“/messages”) being called with different sets of query parameter, there can be only one web method to handle all the call to the respective path. However, the method can accept any number of query parameter. The initialization of which depends upon the query parameter being passed on the URI and the call the appropriate stream of execution should be well defined by the programmer since the method remains the same for various query parameters.

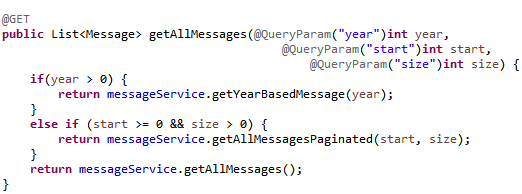
In the below mentioned example the method “*getAllmessages*” is imeplemented with functionality to handle the following scenarios.

1. Get all the messages by date.
2. Get a defined (size is passed by the user) list of messages.
3. Get all the messages.

File 1: MessagerService.java



File 2: MessageResource.java



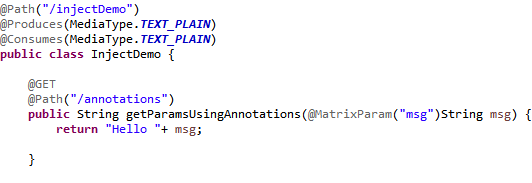
# Param Annotation

## Matrix Param

This can be considered as another version of the query parameter. But the only difference is that instead of a question mark between the resource and the query parameter, in the case of matrix parameter the would be a semicolon.

Example URI: http://localhost:8080/messenger/webapi/injectDemo/**annotations;msg**=vignesh

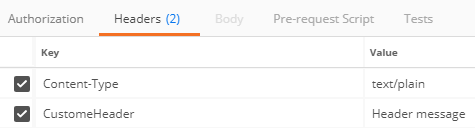
File 1: InjectDemo.java



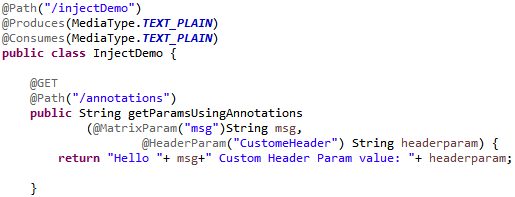
## Header Param

The header param annotation is used to access the values of the header it might be the existing header attributes or custom header values.

Example URI: http://localhost:8080/messenger/webapi/injectDemo/annotations;msg=vignesh



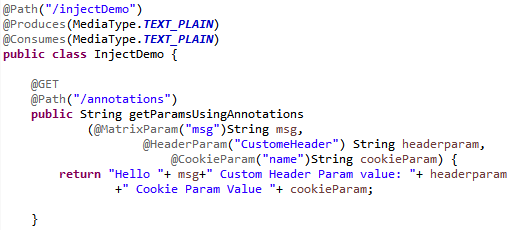
File 1: InjectDemo.java



## Cookie Param

This parameter is used to access the cookie value from the URI.

File 1: InjectDemo.java



# Using Context and Bean param annotation

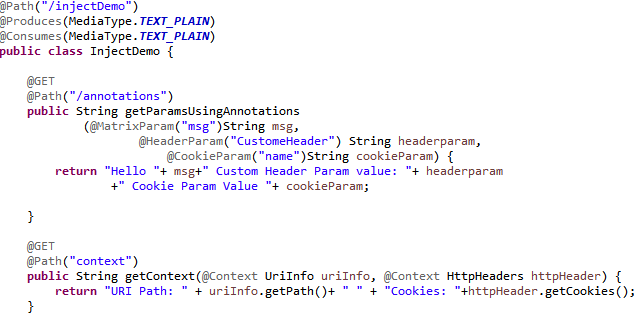
The main intention behind the context and bean param is that instead accessing say the header information bit by bit like, @Cookie, @Matrix which is a complete entity contained in the header.

We can directly access the complete header using the @Context annotation but only few object types are compatible with the annotation.

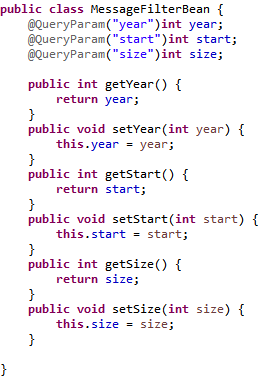
1. UriInfo
2. Httpheader.

As for the Bean Param, if class has 10 query param, writing all the query param in the method signature makes it complex. Under such circumstance, we can define a class that has all such variables and use an object of that class in the method where it is required in the form of an argument.

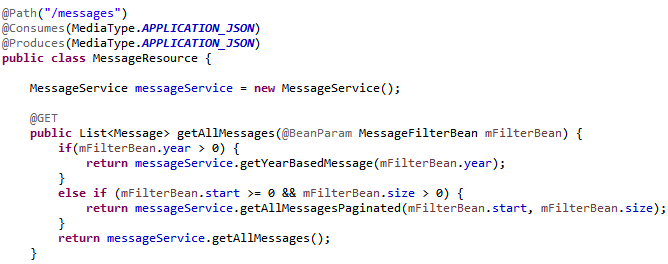
File 1: InjectDemo.java



File 2: MessageFilterBean.java



File 3: MessageResource.java

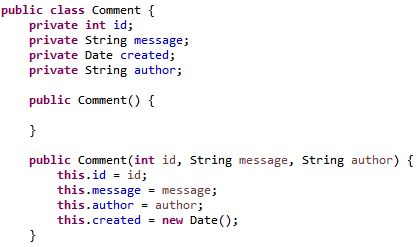


# Implementing Sub Resources

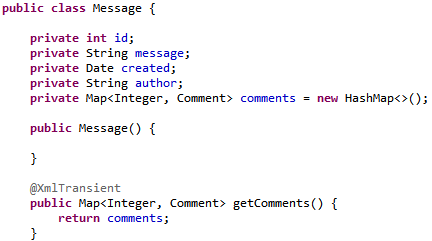
As per the project that is being developed, every given message will be associated to N number of comments and the URI will begin with path related to messages followed by the path to comments. But it is not required that the handler for comments need to be implemented within the class that handles URI invocation for messages.

In Such scenario, we can delegate the control to the comment resource handler class by implementing a method that returns an object of the type “Comment”.

File 1: Comment.java



File 2: Modified Message.java class (added field to accommodate a list of comment)

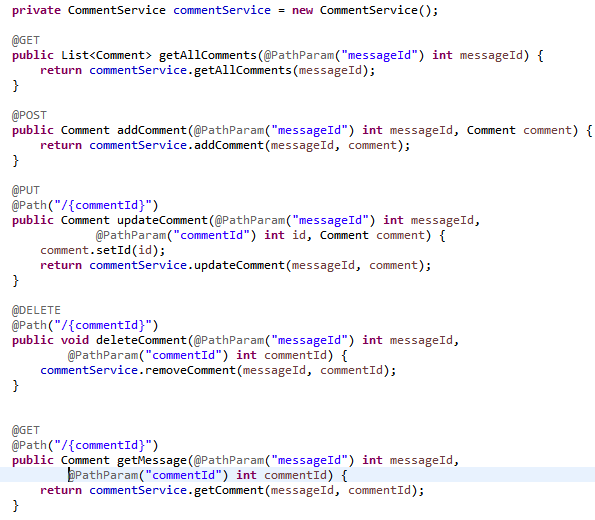


The XmlTransient annotation indicates that the comments object is to be omitted while sending the message object when a rest call is received.

File 3: CommentService.java

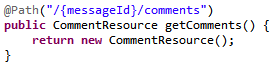


File 4: MessageResource.java



Note: From the above mentioned class it can be observed that the path mentioned as the root is only “/comments”. But the comment class will be able to access the parameters in the path preceding to “/comments”, provided it is the path which triggered current the flow.

File 5: MessageResource.java (Modified to delegate the control to CommentResource.java class)



# Sending Status code and Headers

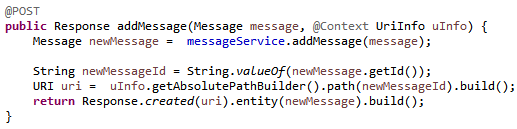
All this time in the above mentioned project under discussion for any given operation the object itself was being returned which prevents us from sending information on the header segment and also to notify the client on the newly created URI or the next relevant URI’s.

In order to be able to send such information the implementation of the web method should be changed to return a “Response” object of the type “*javax.ws.rs.core.Response*”. following which the response object has in butyl method that allows us to define the header values.

For example, rather than setting the response code to some value, if we use the “created” method of the response class to mention the newly created URI to the client the status code also gets initialized to the appropriate value.

The segments of the response is built using the builder design pattern to compose the original object.

File 1: *MessageResource.java*



# Handling Exceptions

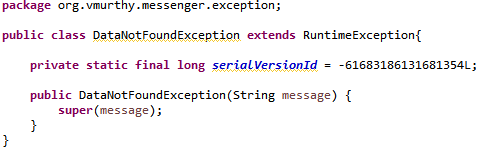
Whenever an exception occurs in a web service the exception is propagated to the calling hierarchy and the chain continues unless a handler is found. In the case where there is no handler, the exception reaches the container and the server generates an error for the same.

Example: If a client initiates a “*getMessage*”API on a message that does not exist, we would generate at present the response is made null, The present functionality will be modified to generate DataNotFoundException.

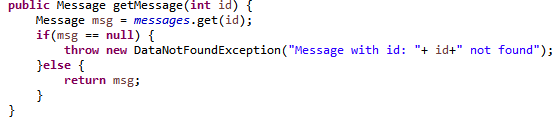
So, when the method “*getMessage”* is called with an id where there is no messages associated to it the method would generate a “*DataNotFoundException*”

For Handling any exception in a restful web service, we will have to devise a exception mapper class by implementing the “Exceptionmapper” interface and register the implementing class to JAX-RS to make sure it is aware of the existence of the exception mapper class.

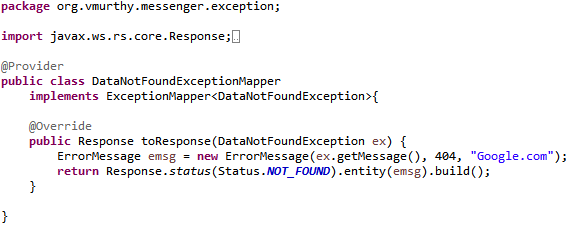
File 1: DataNotFoundException.java



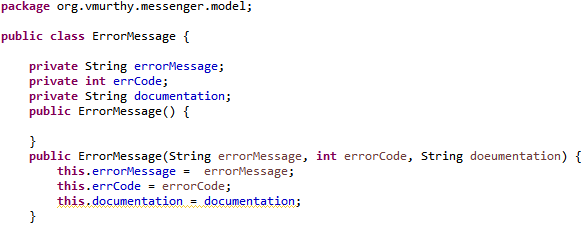
File 2: MessageService.java



File 3: DataNotFoundExceptionMapper.java

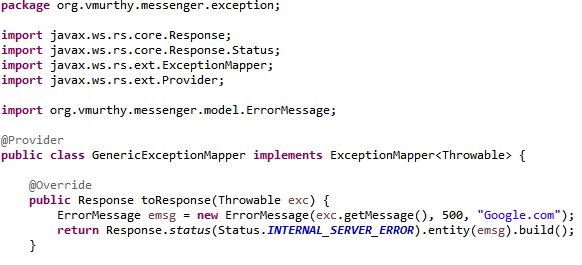


File 4: ErrorMessage.java



Whenever an exception is generated in the “getMessage” method contained in the MessageService class the exception gets propagated to MessageResource and then from there to the container where it performs a lookup in the JAX-RS registry to see if there are any exception mapper for the exception and once it finds it, the exception gets wrapped into an model object and the object is presented to the client.

File : Generic Exception Handler



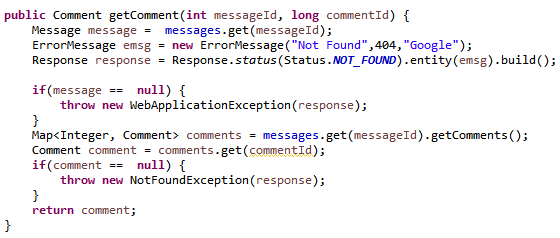
If there is any exception other than the DatnotFoundException, it will be handled by the generic exception mapper as it implements “*ExceptionMapper*” of type “Throwable”.

# Using Web Application Exception

The JAX-RS itself provides Exception mapper for few exception that are within the hierarchy of WebApplicationException. So, whenever an exception occurs for few types we can frame a customer response object and pass that as an argument to the WebApplicationException class and generate WebApplicationException which will be handled by the JAX-RS.

There are few other Exception that fall under the hierarchy of the WebApplicationException which can also be generated in similar fashion the below mentioned example demonstrates both WebApplicationException and NotFoundException (Resource not found scenario).

File 1: CommentService.java



But using of web application exception is discouraged since it requires the implementation to be done in the business layer or in the presentation layer rather than having it as a separate entity.

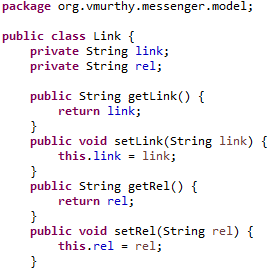
In Summary, there are three ways to handle exception.

1. Through custom Exception mappers.
2. Using Web Application Exception.
3. By sub classes of Web Application Exceptions

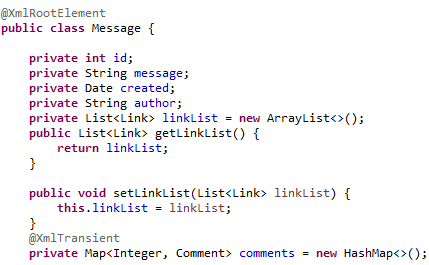
# HATEOAS

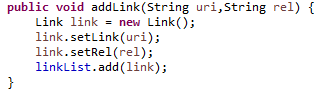
***Hyper Text as the Engine for Application*** state is a design by which we provide a navigable link for the client through the response of a given call. That is while sending the response for a given resource we also embed the links to the relevant resources in the response which also includes the link to the resource itself (Self link).

*File 1*: Link.java

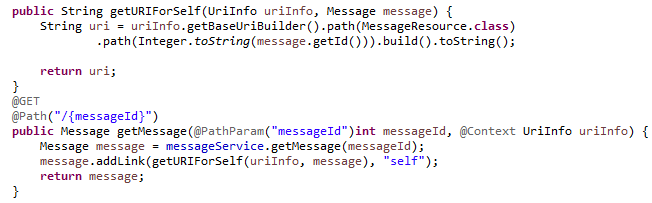


*File 2*: Message.java





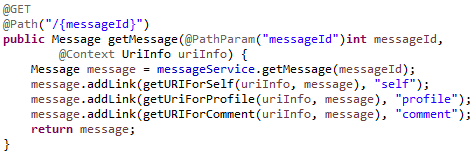
*File 3*: MessageResource.java



# HATEOAS Continuation

Since “Comments” is a sub resource it cannot be built in similar to all other resources. The below mentioned in the logic in the method “*getCommentUri*” details the implementation.





# Content Negotiation

It is a process by which the client and the rest web service agrees upon the format of the message being exchanged. The content negotiation happens by means of two properties contained in the header.

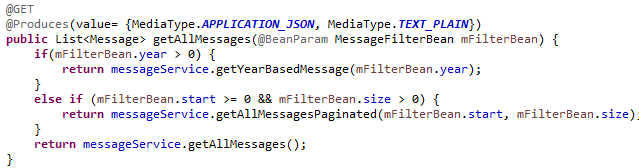
1. Accept: this is a property that defines the forma of the message that is expected by the client from the rest web service.
2. Content Type: Defines the format of message being sent by the client to the web service.

The reason we have these features is the allow us to write defined methods that can process resource call for XML format and JSON format individually. However it is also possible for one method to accept multiple data format and product multiple data format.

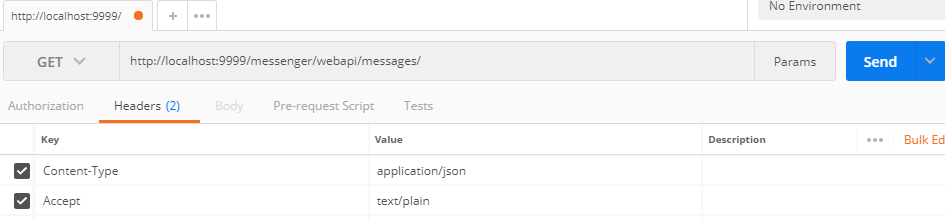
File 1: MessageResource.java



File 1: MessageResource.java (One method which can product JSON and plain text format result)



Header Fields:



Any given method can have the combination of parameters that performs their invocation.

1. Resource URI
2. Http Request Method Type.
3. @produces
4. @Consumes