

Table of Contents

1	Introduction	4
1.1	Web services in a nutshell	4
1.2	Web services characteristics	4
1.3	Representational state transfer	6
2	REST and HTTP	7
2.1	Understanding HTTP	7
2.2	Resource locations	7
2.3	HTTP methods	8
2.4	Metadata	8
2.5	Content types	9
2.6	Summary	9
3	Designing Resource URIs	10
3.1	URIs, retro style	10
3.2	Resource relations	12
3.3	Summary	13
4	RESTful URI types	14
4.1	URI Types	14
4.2	Filtering collections	15
4.3	Summary	15
5	HTTP Methods	16
5.1	Getting a message	17
5.2	Updating a message	17
5.3	Deleting a message	17
5.4	Creating a new message	18
5.5	Collection URI scenarios	18
5.6	Summary	18
6	Idempotence In HTTP Methods	20
6.1	PUT vs POST Methods	20
6.2	Definition of idempotence	21
7	REST Response	23
7.1	From requests to responses	23
7.2	Formats	23
7.3	Message Headers	24
7.4	Status codes	25
7.4.1	1XX Codes - Informational	25
7.4.2	2XX Codes - Success	25
7.4.3	3XX Codes - Redirection	26

7.4.4	4XX Codes - Client error	26
7.4.5	5XX Codes - Server error	27
7.5	Scenarios	27
8	REST Response	28
8.1	Hyperlinking	28
8.2	Scenario	29
8.3	Link relations	30
9	The Richardson Maturity Model	33
10	What is JAX-RS	36
11	Setting up	37
12	Understanding the Application Structure	38
13	Creating resource	39
14	Return XML response	40
15	PostMan as client	41
16	Accessing Path Param.....	42
17	Sample message Service.....	43
18	Return JSON response	45
19	Implementing POST Method	45
20	Implementing Update and Delete Method	46
20.1	Implementing PUT method for update.....	46
20.2	Implementing DELETE method	46
21	Pagination and Filtering (@QueryParam)	48
22	Param Annotations	49
22.1	MatrixParam Annotations	49
22.2	HeaderParam Annotations	49
22.3	CookieParam Annotations	49
22.4	BeanParam Annotations	49
22.5	Context Annotations	50
23	Implementing Sub Resources.....	51
24	Sending Status codes and Location Headers	52
25	Exception handling.....	53
25.1	Using WebApplicationException	54
26	HATEOAS	55

27 Bootstrap JAX-RS Application	57
28 Resource Life Cycle	58
29 Param Annotations and Member Variables.....	59
30 Param Converters.....	60
30.1 Custom Converters	60
31 MessageBodyReaders and MessageBodyWriters	62
31.1 MessageBody vs Param	62
31.2 Implementing MessageBodyWriters	63
31.3 Implementing Custom Media Type	64
32 JAX-RS Client	65
32.1 JAX-RS Client code.....	65
32.2 Creating Invocations.....	66
32.3 Handling Generics.....	67
33 Rest API Authentication.....	68
33.1 Implementing Filters	68
33.2 Basic Authorization.....	68
34 Filter and Interceptors	70
35 Conclusion	72

1 Introduction

1.1 Web services in a nutshell

Web services are services that are exposed on the internet for programmatic access. They are online APIs that you can call from your code.

- To call an API when writing Java code, you add the jars or classes in your class path. Everything executes on a single machine.
- In the case of web services, you have different pieces of code deployed in different machines and calling methods of each other over the network.

You must have seen different Facebook or Twitter apps. Or games that can post to your Facebook wall. Even though the games are not designed by Facebook. How can they do that? They do that by calling online APIs. Companies like Facebook and Twitter publish web services that let other developers call them from their code. So, other application developers can write code to consume these services and do things on Facebook or Twitter itself.

In a way, they are similar to web pages. For example, Twitter has a web site at twitter.com. When you access it, you get HTML response that lets you read and write tweets. They have some HTML elements for data as well as styling with CSS. That's because they are meant for humans to read and consume. But, Twitter also has this different URL called api.twitter.com that does a lot of the same things as twitter.com, but it behaves a bit differently.

For instance, it does not have HTML and CSS. Any data it returns is in XML or JSON format. And there are specific URLs for different operations. This is what the developers can use from their code, to read data and write data to Twitter. Since it deals with just barebones data, they can just parse the data and build up their objects and data structures. There's no need for all the fancy HTML and CSS.

RESTful web services are a type of web services that are modern, light-weight, and use a lot of the concepts behind HTTP, the technology that drives the web. There is another type of web services that you can choose to write your services in, and that's called SOAP web services.

1.2 Web services characteristics

When we talk about web services, there are a few characteristics about them that we have to keep in mind.

First, they are web services, so the exchange of data happens over the web. Over HTTP. A client sends an HTTP request, and the server returns back an HTTP response, similar to web sites.

But like we discussed, instead of the response being complete web pages, **only the data is returned, because the client is just a program**, and not a human.

The client could then have its own logic to present the data to the users in a presentable format, but the exchange between the web service client and web service server is usually just bare-bones data.

The next characteristic of web services is the protocol used. Now what's a protocol? When a web service client makes a request to a web service endpoint, they are usually messages transmitted from one machine to another. These messages need to be in a format, a language, that both the client and the server can understand.

This language, or protocol, is standardized in some web service types.

For SOAP web services is a type of web services, where the protocol always has to follow the standard called SOAP. All SOAP web services uses this protocol to communicate with clients. It used to stand for Simple Object Access Protocol. But that name is now discontinued and we are stuck with just the acronym. There is a specific format, which is XML, and there are specific rules which detail how that XML should be. The client and the server needs to talk to each other using this SOAP protocol only!

- What's the protocol for REST? Well, there is none! Yes, a REST client can send messages in XML. Or it can send messages in JSON format! Or text format! There are no rules. As long as the client and server understand each other, everybody is happy.
- Secondly, let's look at how the communication happens. We saw that since RESTful web services are web services, the request and response messages are almost always exchanged over HTTP. But in HTTP, there are different methods available. You would have heard of GET, POST PUT and so on. Which methods to you need to use? What's the standard for REST?
- Well, the answer is, there is none. Messages can be exchanged in any (or all) HTTP methods. There are guidelines and best practices that tells you what methods need to be used when designing the service, depending on what the request is, but there is no rule as such.

Let's look at the next characteristic: service definition. When you are coding and you need to call a method of a library class, you need to know certain things. The name of the class, the method name, the input arguments and so on. When you call web services, a similar concept applies. You need to know what the method does, what the input arguments are and what the return type is. You need to know the service definition.

In the SOAP web services world, every web service provider publishes a **formal document called WSDL** that contains all the details that any client would ever need to know about the web service. This is again in XML, and it needs have a specific structure. It contains details about the methods available, the input and output types and so on. Every client has all the details they'd need.

What's the formal document that specifies the service definition details for REST web services? I'm pretty sure you can guess the answer to this one. Yes, you are right. There is none!

Most Restful web services come with an informal README document written not in XML but in readable English. Many of them don't even have documentation. In fact, the best RESTful web services would not even need any documentation.

You are probably wondering if this is really true. Is it really the case? Does the RESTful web services have any rules at all? There are no rules. In contrast, SOAP web services have strict rules for each of these characteristics we've seen.

The reason for this kind of a difference is simple. All SOAP web services follow this thing called the SOAP specification. This specification is a set of rules that dictate what a SOAP web service should be. This was designed by a committee, and it is still maintained by the committee. The specification lays out all the rules, including the rules we just discussed. If a web service doesn't follow even one of these rules, it is, by definition, not a SOAP web service. As simple as that.

RESTful web services, on the other hand, does not have any specification! It is a concept. An idea. There is no specification, and no committee to tell you what's right and what's wrong.

The term REST was first introduced by a **guy named Roy Fielding** in his doctoral thesis back in the year 2000. REST stands for REpresentational State Transfer, and it is not really about web services at all!

1.3 Representational state transfer

Representational state transfer is actually ***an architecture style***. Say, you are working on the architecture of a new application. There are certain decisions you'll need to make. Certain criteria you need to think about.

REST consists of a coordinated set of these criteria and constraints that you can use to guide you in the application architecture. It is a set of guidelines. It is a style of architecture.

You can use this style for any application. However, if you apply this style and these guidelines when architecting a web service, you have drum roll Restful web services. There in lies the difference.

Unlike SOAP web services, you don't have any strict rulebooks to follow when it comes to Restful web services. You can have a spectrum.

It is common to hear people say some web service is completely Restful, and some other web service is not fully RESTful. Really! And the goal when building RESTful web services is to make it as RESTful as practically possible.

2 REST and HTTP

The concepts of REST are very closely linked with HTTP. HTTP, as you probably know, is everywhere on the internet. Every time you load a web page, you make an HTTP request and you get HTML content in the response.

REST is inspired by a lot of the concepts of HTTP. Roy Fielding, the one who coined the term, is one of the principal authors of the HTTP specification. So, it's no surprise that the ideas behind REST make good use of the ideas and concepts behind HTTP. So, to understand REST, you really need to have some basic understanding of HTTP itself. Notice that I said HTTP specification. Yes, specification means rules. What defines HTTP is clearly laid out in the specification, so unlike REST, there is no vagueness about it.

2.1 Understanding HTTP

HTTP stands for HyperText Transfer Protocol. Like we've already seen, you can think of a protocol as a language or mechanism for communication. So, HTTP is a way to exchange and communicate information online.

The stuff you exchange and transfer in HTTP is called hypertext. (Hence the name). Hypertext is a structured form of text that has one interesting property: it contains logical links to other text. These links are called hyperlinks.

A common and popular way to write hypertext is using a language called HyperText Markup Language, or HTML.

2.2 Resource locations

Just like web pages, REST APIs have URLs and addresses too. That way they are similar to web sites. One major difference is, since they are not meant to be read directly by humans, **the response usually contains just the core data.**

For example, if you need to look up a weather for a place on a weather website, you'll get a response with HTML showing the weather in a readable format. The HTML returned might also have other HTML elements, some CSS for styling, the site banner, some ads on the side and so on. This is because this response is meant to be read by a human. ***But a REST API response for a weather service probably has just the weather data in XML or JSON.***

Since APIs have addresses, ***an API designer or web service developer needs to decide what the addresses should be.***

The practice in RESTful APIs is to have resource based addresses. In the case of a weather website, the URI to look up the weather at a zip code 12345 could be something like

</weatherapp.com/weatherLookup.do?zipcode=12345>.

This is a perfectly valid URI, and it is common to see addresses like this too. *But this address is not resource based. I would say this is more action based.* This tells you that there is something called weatherLookup.do that takes the zip code as parameter.

Resource based addresses, on the other hand, indicate just the resource and they are independent of the server side implementation.

For instance, a RESTful API for weather could have the address `/weatherapp/zipcode/12345`.

It's almost as if you are not making the server do any action, but rather just look up and get something that already exists.

So, weather for zip code 56789 is at `/weatherapp/zipcode/56789`. And weather forecast for a country could be designed to be at a location like `/weatherapp/countries/countryname`.

2.3 HTTP methods

Now that you have decided what the address is, how do you interact with it? *HTTP has what are called methods or verbs that you can use to interact with URLs.*

We must be familiar with GET and POST methods. A GET method lets you get information from the server. And POST is used when you want to submit information to the server. **They work well with resource based URIs** that we just saw. So a GET request to URI `/weatherapp/zip/12345` will get you the weather at that location.

There is another method called PUT that you wouldn't normally use in HTML forms. PUT also lets you submit data to the server, but it is a bit different from POST, and we'll learn about this later. There's also a DELETE method that lets you specify that you want something removed.

A good RESTful web service API makes good use of these HTTP methods. Not all requests are done through POST like a SOAP web service would do. The method that a developer chooses for each API action depends on the action that is performed and the intended use.

2.4 Metadata

Ok, so let's say we tell the client what the address is and what HTTP method to use to call it. When they make the call, what is the response that we'll send back? Well, obviously we need to send the response they want.

A GET request for the weather URL would have the weather information in the response body. But HTTP also defines status codes and response headers which lets the server send back extra information or metadata that might be useful to the client.

One useful piece of information that every response has is the status code. It's a number that shows up in the very first line of the response. It indicates if the response was successful or if there was an error. If a HTTP response is successful, a 200 status code is returned. If there is an error on the server while

processing the request, the server sends back status code 500. If you are trying to access something that does not exist or the server is unable to find, the popular error code 404 is returned.

If you are accessing a website, you'll probably get some HTML that explains the problem. For example, for a 404 error, you get a page that shows the "page not found" message, probably along with links to the home page to help the user.

But in the case of RESTful web services, you cannot send readable messages because the client is a piece of code! This is why sending the right status code is very important.

2.5 Content types

Finally, let's look at the format of the messages. Let's say you submit some data to the server as a POST request.

There is no specification that strictly enforces what the format of the data should be. It could be XML, JSON or some other format.

How can the server even identify what kind of data is sent? Similarly, ***how does the client know what data format is returned by the server? The answer is again a header value called Content-Type.***

Like I mentioned, the headers contain a lot of metadata, and one of the metadata values it can contain is the format of the message. There are standard predefined content type values, like `text/xml` for XML content or `application/json` for JSON content. A message that's sent with the right content type is easily readable by the server and the client. What's really interesting is that the same API can send back data in multiple different formats, and the actual format it chooses depends on what the client wants. This happens by a process called content negotiation, which is another powerful feature that you can use when developing RESTful web services.

2.6 Summary

This was a broad overview of some of the important points about RESTful web services and how they've been influenced by HTTP. When you design a RESTful API:

- You need to have resource-based URIs. Every resource or entity should be identifiable by a single URI.
- You need to choose the right HTTP methods for different actions and operations for the API.
- The response needs to return the right HTTP status codes
- All requests and responses need to have the right Content-Type header set so that the format of the messages are well understood by everyone.

3 Designing Resource URIs

In this section, we understand how RESTful URIs are designed, understanding REST API design is more important than learning how to actually implement them.

The app we are building in this course is gonna be called Messenger. It's a social media application that lets people post messages as status updates. They can also write comments on other messages or like other messages ala Facebook. Users also have user profile information that they can create and update.

A very simple application with a very simple ER diagram.

- You have a USER table with user information.
- MESSAGES table that contains all messages anyone ever posted, each row referring to a USER who posted it.
- COMMENTS and LIKES tables which refer to the message that's being commented or liked. And the user who has entered the comment or hit 'like'.

We want to design RESTful "resource based" URIs. How do we do that?

If this were to be a web application and in Struts framework, If It takes in a message ID to display messages. The URI to get post ID 10 could be something like this:

`/MyApp/getMessages.do?id=10`

Or it could be: `/MyApp/retrieveMessages.action?postId=10`

However, when writing REST APIs, the consumers have to be aware of the URIs. This is because, the consumer of your RESTful API is a developer who has to write code to make HTTP calls to the URI.

What would really help is have a common URI convention for entities like this. That's where the RESTful concept of resource URIs come in.

Before I start explaining the best practices for forming these URIs, I should tell you, that's what this is: best practice. Like we've seen before, there is no right or wrong way to create URIs.

But if you are writing a REST API, it's better you follow these best practices to keep both you, and the API client from going completely insane.

3.1 URIs, retro style

To access the pages of the site, you would enter the URL that consists of the path to the page ending with the page name. Every page has a specific URI that uniquely identifies that HTML page. There is no ambiguity there. This is exactly the concept behind resource based URIs.

Every thing or entity has a URL that's unique and standard.

Best way to design RESTful URIs is to think of them as static pages. Take the profile pages on a site like Facebook for example. If you had to design profile pages as static HTML, think of how you'd create them. You would create one HTML for every profile. So, if my profile name is naren, the name of the page would be naren.html.

Let's say there are 4 users for this website: naren, raj, sid and jane. So, I have 4 static HTML pages, the names of the files being the names of the profile. Now that I have a bunch of these profile HTML files on my site, I'll group all the profile pages in a profiles folder. **So, the path to my profile page would be something like:**

`/profiles/naren.html`

Drop the .html extension, and you have the RESTful URI.

`/profiles/naren`

Making it generic, the URI for any profile page is:

`/profiles/{profileName}`

Think of resources and create a unique URI for them. Let's look at some more examples. How about posts or messages? Let's say every message has an ID. Then you could design URIs like this:

`/messages/{messageId}`

So, the URI

`/messages/1` shows you message ID 1 and

`/messages/10` shows message ID 10 and so on.

Notice two things with the URIs.

First, the URI contains nouns and not verbs. Things in the system like documents, persons, products or accounts are resources. You don't have URIs like `getMessages` or `fetchMessages`. It's just messages. When you are designing RESTful URIs, keep an eye out for any verbs in your URI. There typically shouldn't be any. Just nouns. And typically the nouns are the resource names themselves, like posts or profiles.

Secondly, notice the resource name is plural in both cases. It's not `/message/{messageId}`. Again, this is a good practice, because it makes it clear there are multiple message under `/messages`, not just one.

So, here's the very first step to designing a REST API for any system.

Identify the things or nouns or entities in your system. They are your resources. Then assign resource URIs for each resource.

The advantage of using such resource based URIs is that they are really not dependent of the framework you use. So, no .do or .action in the URIs. And no ?id= query params. These details are of no significance to your clients,

Let's look at some other nouns in our application are comments, likes and shares. Each one can be a resource. Let's start with comments.

What would be a good restful URI for a comment with ID 20? Well, it could be **/comments/20**. That's correct. But there is one more thing you can do here.

3.2 Resource relations

When designing URIs for resources, you'll often encounter **some resources that are dependent on each other**. Take the example of messages and comments. Someone posts a message and then other people comment on it.

A message can have multiple comments, and each comment has its own IDs. **A message has a one-to-many relationship with comments**. We've designed the URI for posts to be /messages/{messageId}. Could the URI for comments be /comments/{commentId}?

Well, it could, but that treats messages and comments as two independent resources, and not acknowledge the relationship between them.

Say we have two messages, message 1 and message 2. Message 1 has comments ID 1, 2 and 3. Message 2 has comments 4 and 5.

If I were designing this as static HTML pages, **we wouldn't want to create one comments folder and put all comments pages in it!** *we would lose the relationship information that comments have to the messages*. To convey that relationship, I could create a folder for each message and put all comments pages related to that message in that folder.

So, the URI for comment 2 is: /messages/1/comments/2

Notice how the message ID is a part of the URI, which is then followed by comments and the comment ID.

The generic URI for a comment is: /messages/{messageId}/comments/{commentId}

Is this URI better than /comments/{commentId}? Well, it depends. This makes it clear that the comment belongs to a particular message, so the relationship between resources is well established. But on the other hand, to get to a comment, you need to know the comment ID as well as the message ID. You know the URI for message ID 20. But what's the URI for comment ID 300? You'd need to know what the message ID is to access any comment. So, it depends on what you expect your client to know when they need to access this.

This structure can be applied to other related resources too. For example, message can be shared, and each share has a unique ID. So, the URI for likes could be:

/messages/{messageId}/likes/{likeId}

A share could be:

/messages/{messageId}/shares/{shareId}

So, typically, when there is a one-to-many relationships, you could choose to have the "one" side of the relationship to be the root resource, and then the resource on the "many" side follow that.

3.3 Summary

So, what are the resources in our system? We have profiles, messages, comments, likes and shares. We have identified resource URIs for each. We'll treat **profiles and messages as first level entities**, and **comments, likes and shares as second level entities** in relation to messages.

Now you might wonder why messages are not related to profiles. Why are they both first level entities? Messages are posted by someone who has a profile. There is a one to many relationship between profile and messages! So, couldn't you have message URIs like this?

/profiles/{profileId}/messages/{messageId}

You could! In this case, I decided to have messages independent of profiles because I felt they weren't as tightly coupled together as messages and comments are. But this is something that you should decide when designing the API for your system.

I hope the concept of resource based URIs makes sense. One other important consideration is a concept of collection URIs.

4 RESTful URI types

We can think of RESTful URIs as belonging to two types: instance resource URIs and collection resource URIs. Let's understand what they mean.

4.1 URI Types

We've designed URIs for messages and comments in the previous section. To recap, a message is accessible at: `/messages/{messageId}`

And comments are accessible at: `/messages/{messageId}/comments/{commentId}`

These URIs are great when you want to look up a particular message or a particular comment. **These are called instance resource URIs. A single instance of a message or a comment is accessible by the instance resource.**

Instance resource URIs typically have a unique ID of that resource to identify which instance you are interested in.

What about if you want all messages?

The answer is simple. Just access `/messages`

That's it! Again, this is analogous to a simple static site with HTML pages. Accessing a directory gives you all the contents in that directory.

So, think of `/messages` as the top level directory for all messages and accessing that URI gives a list of all messages.

Similarly, if you need all comments made for message 2, the URI is: `/messages/2/comments`. That's the directory for all comments for message 2.

These URIs are different from the URIs we saw in the previous section. They do not represent a particular resource, **but rather, a collection or a list of resources. So, they are called collection URIs.** These URIs pull up a collection of instance resources.

This also explains why the resource names are in plural. They help the client understand that they are working with a collection of resources with these URIs.

- `/messages` returns all messages
- `/profiles` returns all profiles
- `/messages/{messageId}/comments` returns all comments
- `messageId /messages/{messageId}/likes` returns all likes
- `messageId /messages/{messageId}/shares` returns all shares for `messageId`

This shows you the advantage of nesting related resources like messages and comments, or messages and likes in this way. But this also brings up a problem. What if you need a list of all comments irrespective of which message they are associated with.

4.2 Filtering collections

When you have collection URIs like this, you'd want a way to filter the result, Getting all comments for a message is mostly not going to be a big list. That's going to result in a lot of data, and the client want to design to paginate or filter the results. **One way to do that is using query params**

. We have so far avoided query params in our URIs, but filtering and pagination is a good scenario to use them. **One standard practice to provide pagination is to have two query params: starting point and page size.**

For example, consider this URI:

`/messages?offset=30&limit=10`

This URI fetches messages starting from message number 30 and returns the next 10 messages. The offset and limit params correspond to the start and page size values.

The client who uses your API might have their own logic for displaying pagination controls on their UI. But they would use these two params to make calls to the RESTful service to get chunks of data in pages.

You can also choose to implement **other kinds of filters using query params**. Take an example of retrieving messages based on date. Let's say you want to provide ability for clients to retrieve messages posted in a given year. You could have them send request like this:

`/messages?year=2014`

This returns all messages made in the year 2014. This is of course, something you can use together with pagination like this:

`/messages?year=2014&offset=50&limit=25`

4.3 Summary

There are two types of REST URIs. One is instance resource URIs that identify a specific resource, and the other is collection URI which represent a collection of resources. Collection URIs usually end in plurals, like messages, comments, products and so on, and are typically a portion of the resource URI. An instance resource URI identify a specific resource below a collection resource URI.

And finally, you can implement query params as a way of achieving pagination and filtering when accessing collection URIs.

5 HTTP Methods

We saw that a URI like `getMessages.do?id=10` is action-based and not RESTful. We chose the URI `/messages/10` to replace it. This is resource based and RESTful and all that good stuff. But here's a problem.

Look at the URI `getMessages.do?id=10`. It has 3 pieces of information.

- It refers to messages
- It refers to ID 10
- It gets that information

Now look at the URI `/messages/10`. There's definitely #1 and #2. But the URI does not have the information in point 3.

You could argue that accessing that URI gets the data for message id 10. But what about other operations? We have looked at just getting data so far. You could have URIs that submit data.

For example: `/submitMessage.do?id=10` lets you submit something to the message ID 10. What would be the REST URI equivalent for that?

Well, equivalent resource based REST URI is: `/messages/10`

Yes, that's right! It's the same URI.

How about a **delete operation**? What would be the equivalent of a URI like `/deleteMessage.do?id=10`?

The equivalent is **still `/messages/10`!**

How is that possible? How can the same URI do all these different operations? And how does it know when to do what?

The answer is HTTP methods. If you want your API client to perform different operations for message 10, just have them use different HTTP methods to the same URI `/messages/10`!

The most common HTTP methods are:

1. GET
2. POST
3. PUT
4. DELETE

There are some other methods that are rarely used like HEAD and OPTIONS. But for the most part, we'll focus on the 4 common HTTP methods.

These HTTP methods have specific meanings, and you typically use the right method for the right operation. Also, when using HTTP, like when you are browsing a web page, you are actually using these methods automatically. When you type in a URL in the address bar of your browser, the browser automatically issues a GET request for that URL. When you submit a form, the browser probably uses a POST request to do so. The idea is to use the right methods depending on the operation. Choosing methods for a RESTful API also follows the same idea.

So, for example, in the REST world, you do not make a call to `getProducts`. You make a GET request to the `products` resource URI. You do not call `deleteOrder`. You make a DELETE request to the order resource URI. The URI tells you what entity or resource is being operated upon and the method tells you what the operation is.

5.1 Getting a message

This should be obvious now. The URI is the resource URI that you need to get. the HTTP method is GET, since you are literally getting the information

Example: GET `/messages/20` returns the message ID 20.

5.2 Updating a message

Let's say our social media application lets you make changes to messages you've already submitted.

What's the right method for submitting an updated message. Is it the POST method? This one is not all that obvious. You can use POST, and I know some people do, **but the standard practice is to use the PUT method. Why PUT and not POST? There is a very important difference between two.** But for now, remember that we'll be using PUT to update or change any resource.

Of course, you'll want the resource to be updated with something. The new content. That content needs to be sent in the body of the PUT request.

Example: PUT `/messages/20` (with the request body containing the new message content) replaces the message ID 20 with the content in the request body.

5.3 Deleting a message

This one should be easy. Yes, there is a DELETE method, and that's what you should use here. When you want your clients to delete a resource, just have them issue a DELETE request to the resource URI. No request body required.

Example: DELETE `/messages/20` deletes message ID 20.

5.4 Creating a new message

This one is interesting. Before we talk about the method to use, think about what URI to call to create a new message. To get, change or delete a message, you used the message resource URI `/messages/{messageId}`. How about creating a new message? The message hasn't been created yet, so there is no message ID! And the ID is typically managed by the application, by finding the next unused ID, so the client usually has no idea what the ID will be.

This is why, requests for creating a new resource is always issued to the collection URI for that resource. **To create a new message, the request is made to `/messages`. To create a new profile, a request is made to `/profiles`.** The application receives this request and creates a new resource and assigns an ID to it.

With that settled, what will be the HTTP method? For creating resources, the practice is to use a POST method. The POST body will should contain the content for creating the resource.

Once the resource is created, the service will need to let the client know what the ID is. Because, unless the client knows the ID, there is no way for it to do anything with the newly created resource. So, in the response for the POST request, the web service sends back the ID of the newly created resource.

Example: POST `/messages` (with the request body containing the new message) creates a new message with the content in the request body. Response to this request contains the message ID that was created.

5.5 Collection URI scenarios

You'll appreciate the elegance and overall awesomeness of REST API practices when you realize how seamlessly these concepts translate to collection URIs.

Imagine what happens when you make a GET request to a collection URI like `/messages`. Yes, you get all messages! Other HTTP methods work the same way too. Here are some examples:

- DELETE on `/messages/10/comments` deletes all comments associated with message 10.
- POST on `/messages/10/comments` creates a new comment for message ID 10 with the request body containing the new comment information.
- PUT on `/messages/20/comments` replaces the list of comments for message ID 20 with a new list of comments in the PUT body. (This kind of API is not commonly used though)
- DELETE on `/messages` deletes all messages (Again, not commonly used, and not something you'd want to implement, I think!)

5.6 Summary

There you go. We have identified standard HTTP methods for all the CRUD operations. Whenever you need to provide APIs for creating, deleting, fetching or updating a resource, you know what methods to choose for them.

This, of course, doesn't address all the operations. Very rarely do web applications perform just CRUD operations. What if you need to provide an API for archiving a message? Or run a server-side job? How can we map GET, PUT, POST and DELETE methods to those miscellaneous operations?

The fact is, these methods do not really map to CRUD operations. It does look like that based on what we've seen in this tutorial, but there is more to it. We will talk about this in a later tutorial, so hold on to that thought for a bit!

6 Idempotence In HTTP Methods

Idempotence. Yes, that's a word. And it's an important property of HTTP methods according to the specifications.

6.1 PUT vs POST Methods

When I was learning about RESTful web services, one thing that confused me was the difference between PUT and POST. Like we've already seen, you use PUT when you want to update an existing resource, and POST when you want to create a new resource. But if you search online, you will very likely find a lot of resources that contradict each other. Some are plain wrong, while others tell you the right thing to do, but do not explain why. I'll try to explain this difference and hopefully, it'll be clear to you by the time you are done with this tutorial.

Method classification

There are two ways in which we can classify these 4 popular HTTP methods: GET, PUT, POST and DELETE. The GET method is a read-only method; it lets you read information. **But the methods PUT, POST and DELETE are write methods; they change something on the server. They either create, update or delete, but they all cause something to change on the server.**

So, it is safe to assume that you can make a GET request as many times as you want without having any impact on the server. You should never have a GET method do things like updates or deletes.

For example: GET on /messages/20/delete. Never do this!

Nothing changes when you do a GET, so it's safe to make multiple requests and not worry about the side effect.

But how about PUT, POST and DELETE? Since they are methods that write to the server, **you obviously cannot make those calls multiple times! Or can you?**

Just because an operation is not read-only, it doesn't automatically mean that it cannot be duplicated.

Take an example of a Java assignment statement. Assume count is an integer variable

```
count = 100;
```

This is definitely not a read-only operation. This statement writes a value 100 to the variable count. However, if you were to repeat this operation three times, **lines 2 and 3 do not really do anything**. Well, maybe they do write the value to the variable, but for all practical purposes, they do not have any effect.

```
count = 100;
```

```
count = 100;
```

```
count = 100;
```

This nature of some operations that let them be repeatable is important in HTTP methods. Like we saw, **GET is clearly a repeatable operation**, because it is read-only.

Let's take DELETE. Say you make a DELETE request to /messages/10. This deletes message ID 10. Say you make the same call again. Well, message 10 is already deleted. So nothing happens. While it isn't really required or desirable to make multiple DELETE calls to the same resource, you can see that it is at least not a problem. There are no unwanted side effects if you were to make a duplicate call by mistake.

how about PUT. Say you make a PUT request to /messages/20 with some message text in the request body. This is going to replace whatever message ID 20 was with this new message text that's being sent in the request body.

Say you make the exact same call again. Message ID 20 is again replaced with the exact same message text again. Make the same request the third time, and the result is the same. Guess what? **Even a PUT is safe when it comes to making multiple calls.** If you were to accidentally repeat a PUT request, well, don't worry about it. The final saved message remains the same after every request.

The problem, however, is with the POST request. If you were to make a POST request to /messages, you create a new message. Say you forgot you made a POST request, and you issued the request again, and now you've actually created a duplicate message. **Repeat that call, and you get another message!** **So every time a POST request is made, something new happens.** This is clearly not a safe method to make multiple calls with. Every duplicate call changes things by creating a new resource. **It's definitely not a good idea to make multiple POST calls, unless you actually need multiple resources.**

So, we have another way to classify HTTP methods into two types.

- One set of methods, including GET, PUT and DELETE, are safe for make repeated calls without worrying about the impact. They may not all be read-only. But they do not cause side-effects if called multiple times.
- And the other category, consisting of POST which you have to be very careful with, and make only as many calls as you need. The methods in the first set are called idempotent methods. GET, PUT and DELETE are idempotent. POST is non-idempotent.

6.2 Definition of idempotence

Idempotence is the property of certain operations in mathematics and computer science, that can be applied multiple times without changing the result beyond the initial application.

The HTTP specification requires GET, PUT and DELETE methods to always be idempotent. If a client makes a request with one of these methods, they do not have to worry about making duplicate requests.

But if they are making a POST request, they cannot safely make duplicate requests without any side effects.

- Which is why resource creation should be a POST method. Because resource creation requests are not idempotent. Which is because multiple requests to create resources results in multiple resources.
- But updating a resource, like we saw, can be called multiple times safely. Which is why update requests ideally use the **HTTP PUT method, which is supposed to be idempotent as per the specification.**

The only way you can safely use PUT for creating a new resource is in scenarios where the client specifies the new ID of the resource being created.

In which case, the client sends the request for creating a resource to the actual instance resource URL that includes the ID. If you were to implement this, then resource creation request is idempotent. Think about it. If you repeat the request, since it has the ID in it, it doesn't create a new resource. Perhaps, the resource with the ID is re-created or updated. This is the only scenario where you can use PUT for creating resources. But in most cases, when you have the server creating IDs and you issue a create resource request to the collection URI, you'd want to use POST.

Like I've mentioned before, these methods have standard meanings.

The fact that this is a standard means that if you ignore it when implementing your APIs, you'll confuse your clients or cause their code to function improperly.

Also, a common thing that many APIs do is cache some of their GET responses. When a client makes a GET request, it also updates the cache, and another GET request to that same resource URI within a certain period of time will be served directly from the cache, thereby increasing performance.

This works only because GET doesn't change anything on the server, so it is cacheable. You can definitely build an **API that creates new resources when your clients call GET, but if you do that, you'll not have many clients using your API for long!**

Clients can build safeguards to make sure duplicate requests do not happen.

Take the example of a browser refresh button. Every browser has a refresh or reload button that does a very simple function: resend the last HTTP request that was made by the browser. If the last request happens to be an idempotent request like a GET, the browser just goes ahead and resends the request when you hit refresh. But if it was a POST, like after you've submitted a form, if you hit refresh, the browser warns you with a message that says something like **"You've already submitted this data before. Are you sure you wish to resubmit?"**.

This is simply the browser protecting you from making a duplicate non-idempotent request. So, it pays to generally use the right HTTP method for the right operation.

7 REST Response

7.1 From requests to responses

When a request comes in, what should the REST web service respond with? Knowing what the client will get back from the server is an important part of the API, because the client needs to write code to handle the response.

If it were a web application, we know the response is usually an HTML page. With styling, formatting and also, of course, the actual data in a presentable format.

But when it comes to RESTful web services, you don't need to do all the styling and formatting anymore. You just need to send the actual data. How do you send it?

We discussed about various standard formats that responses can be sent in, like XML and JSON. JSON has been growing in popularity, because it is much more compact and less verbose when compared to XML, especially when large data is involved.

Also, more often than not, a client to a RESTful API is client side Javascript code, **and sending back data in JSON means it can easily convert it to a Javascript object.** Considering these advantages, we'll choose JSON as response for our social media application in this course.

However, note that you do not typically need to settle for just one format. You can write APIs to support multiple response formats, and we will implement one such API endpoint later in this course to illustrate that.

7.2 Formats

Let's say our Message entity class has these four member variables: the id, the text of the message, when it was created and who created it.

```
public class MessageEntity {  
    private long id;  
    private String message;  
    private Date created;  
    private String author;  
    ...  
}
```

When a GET request is made for a specific message, say message ID 10, the JSON that you would return would look something like this:

```
{  
    "id": "10",  
    "message": "Hello world",  
    "created": "2014-06-01T18:06:36.902",  
}
```

```
"author": "Naren"  
}
```

But the response doesn't have to be JSON. You could return XML as well, if the client asks for it in XML format. We haven't yet covered how a client can ask for a specific format, Here is a possible XML response for the same message ID 10.

```
<messageEntity>  
  <id>10</id>  
  <message>Hello world</message>  
  <created>2014-06-01T18:06:36.902</created>  
  <author>koushik</author>  
</messageEntity>
```

Clearly the JSON response and the XML response are different. But they represent the same resource: message ID 10. So, in other words **both these responses are different representations of the same resource.**

This is a very important thing to remember. When you make REST API calls you are sending or receiving representations of the resource.

Different representations could have different formats, even though the underlying resource is the same. This is actually how REST gets its name. Representational State Transfer. You are transferring the representational state.

When you make REST API calls you are sending or receiving representations of the resource.

7.3 Message Headers

It's great that a REST web service can return data in XML or JSON. But that brings up a problem. **How does the client know what format the response is in?** The client can of course request data in a particular format, but there's no guarantee that the service responds in that format. Say, a client request asks for XML. But if the REST service knows only JSON, it does return JSON ignoring the client's preference for XML. How does the client know the format then?

The answer is using HTTP headers. The HTTP protocol has a concept of request and response headers. Every HTTP request or response has a body, which is the message itself, and certain header values that contain metadata about the message.

The header data could be stuff like the content length and date. **One such possible header is Content-Type.** The response could contain the Content-Type header with the value for JSON or XML. There are special values for JSON and XML, and type of content is being sent back as a response header. The client can then examine this header value and then parse the response body content accordingly.

7.4 Status codes

Think about error messages in a web application. When something goes wrong, the application typically returns a page with an error message, maybe in bold red text. Even if it isn't in red, the message itself would give the user an idea that it's an error.

But in the case of REST APIs, since the consumer is not a human, we need to provide some set of codes to the consumer to help them identify error scenarios.

HTTP specification requires the very first line of any response to be a status line. This line will have a numerical code and a short phrase explaining what the code means. This is not just for errors. Every HTTP response needs to have this line.

If the response is successful, the very first line of the response will be: 200 OK

Let's take the familiar 404 error code. If a request is made on a URI, for example /messages/101 and there is *no message available with ID 101*, the first line of the response should be: 404 Not Found

Again, the code 404 is for the client code to read and act. The phrase Not Found is an aid to the programmer, in case they forget what the code means.

There are a bunch of codes that are important for us to remember and use when developing a REST API. The error codes start from 100 and go up to 599. Not all of them are valid error codes though, so you don't have 500 different possible error codes.

There are 5 classes of status codes and the first digit indicates what class the code belongs to: 1 to 5.

7.4.1 1XX Codes - Informational

The codes starting with 1XX are informational, like acknowledgement responses. We'll not be using this set of codes in this course.

7.4.2 2XX Codes - Success

The codes starting with 2XX are success codes. This indicates that the server received the request from the client and processed it successfully. Some examples:

200 OK

Indicates successful response. You'd return this for any request that you can successfully respond to.

201 Created

Indicates successful resource creation. Say you get a POST request for a collection URI like /messages and you successfully create a new message. You could return 200 OK to indicate success, but a better response code would be 201 Created.

204 No Content

Sometimes the server receives requests that need it to do something, but it doesn't need to return any content back. Like DELETE requests, for example. In this case, you could either return 200 OK with no response content. Or return 204 No Content, which makes it obvious that the server really intends to send nothing back.

7.4.3 3XX Codes - Redirection

The server sends these codes to ask the client to do further action to complete the request. For example, it could be a redirect, asking the client to send the request somewhere else.

302 Found and 307 Temporary Redirect

One of these two error codes are returned by the server if it wants the client to request elsewhere. It's a redirect.

304 Not Modified

When a client tries to get a resource that it has already got before, the server can send this status code to say "I've already given you this resource a little while back, and nothing has changed since then."

7.4.4 4XX Codes - Clienterror

These error codes are returned if the client makes an error in the request. The request syntax could have been incorrect, or the client is requesting something that it's not supposed to see.

400 Bad Request

This is a client error. The server is not able to understand the request

401 Unauthorized

The request needs the client to sign in or authorize themselves.

403 Forbidden

The client may have authorized, but they are still not allowed to make the request. (Maybe they don't have the right access rights).

404 Not Found

Not found

415 Unsupported Media Type

The client is speaking in a language that the server cannot understand

7.4.5 5XX Codes - Server error

The 4XX codes are when the client screws up when sending the request. **The 5XX codes are when the server screws up when sending the response.** It's basically the server saying, Ok, I got your request, and it looked like a valid one, but something went wrong when I tried to process it.

500 Internal Server Error

This is a generic error code. The server gets a request. The resource exists (or you'd send a 404 instead) but something went wrong when processing the request. In such cases, the standard practice is to send the error code 500, along with error details in the body of the request.

There are a bunch of other codes, but these are the important ones to remember.

These error codes are for you, as a web service developer to use. The clients know what it means when they see one of these error codes.

So, it's up to you to send the right error codes when these events happen. For example, let's say you get a runtime exception when processing a request. You need to send back error code 500.

7.5 Scenarios

Let's look at the same CRUD use cases we saw in the previous tutorial, and identify what the status codes should be for the message resource

Operation	URI	Method	Success / Failure	Status code
Get message	/messages/{messageId}	GET	Success	200
			Not found	404
			Failure	500
Delete message	/messages/{messageId}	DELETE	Success	200 or 204
			Not found	404
			Failure	500
Edit message	/messages/{messageId}	PUT	Success	200
			Wrong format / data	400 or 415
			Not found	404
			Failure	500
Create message	/messages	POST	Success	201
			Wrong format / data	400 or 415
			Failure	500

Responses for other resources would mostly follow the same pattern. Again, this is just a small subset of the HTTP status codes.

8 REST Response

This is actually an acronym. HATEOAS. It stands for Hypermedia as the Engine of Application State.

8.1 Hyperlinking

There's no service definition specification for REST APIs. There's no formal document that really documents the API. Most REST APIs have "help" pages that explain what the API URIs are and what operations are supported.

We visit websites online all the time. You don't need documentation to use web sites. You go to the home page, and you'll find links to other pages. You click on one such link, and you'll get that page, with more links.

This is basically the advantage of using HTTP. Remember that HTTP is HyperText Transfer Protocol. hypertext is text that has links to other text. These links, which are called hyperlinks, are what's really handy to navigate your way through any site.

Let's think about the response we return in our REST API. What if we implement the same concept there too? Let's say you receive a GET request from a client for a message ID. **We return the message information in JSON or XML, yes. But what you could also do is send links to comment resource URIs. And likes and shares resource URIs.**

It's the server saying "Hey client, I know you asked for message ID 20. Here's the contents of message #20. I'm also throwing in collection resource URIs for comments, shares and likes. If you want to get a list of all the comments for message ID 20, this is the URI to use. Oh, and here's the profile resource URI for the author of the message, if you want to get the profile information of the author of this message".

So, the web service is being super-helpful to the client by providing all these links in the response. Similar to hyperlinks in web sites. Whether the client wants to use it or not doesn't matter. But if they want it, it's there. And just like that, you've eliminated the need for documentation for all these APIs.

The client developer just picks up the value of the right URIs from a previous response and makes subsequent calls to those URIs.

If you do this, you don't let the client programmer have to know and hard-code the URIs in order to interact with the resources and the application state. You basically let the hypertext you send in the response drive the client's interaction with the application state.

So, you could say that hypertext, or hypermedia as it is sometimes called, **is being the driver or engine of application state. Hypermedia as the Engine of Application State. HATEOAS.**

8.2 Scenario

Let's walk through a scenario so that this concept becomes clearer. Let's start with the `/messages` collection URI. Accessing `/messages` should give you a list of messages in the system. Let's say a message representation has the following fields:

- Message ID
- Message Content
- Message Author
- Posted Date

Four simple properties. A JSON representation for a sample message would look something like this:

```
{
  "id": "01",
  "content": "Hello World!",
  "author": "naren",
  "postedDate": "03-01-2014"
}
```

Now when you access the `/messages` collection URI, you'd basically get a collection of such message resources. To keep it simple, let's say there are just 3 messages in the system. Accessing `/messages` would give something similar to this:

```
[
  {
    "id": "1",
    "content": "Hello World!",
    "author": "naren",
    "postedDate": "03-01-2014"
  },
  {
    "id": "2",
    "content": "Yo!",
    "author": "sid",
    "postedDate": "04-01-2014"
  },
  {
    "id": "3",
    "content": "What's up?",
    "author": "jane",
    "postedDate": "04-02-2014"
  }
]
```

Now that the client has the list of messages, let's say they want the details of the first message - message ID 1. We've already designed the resource URI for message to be `/messages/{messageId}`.

So, to get the URI, they'll have to take the value of the ID field of the message they are interested in, and append it to the string `/messages/` and there they have the resource URI.

As a API service implementer, why not send that to the client yourself? Since we are sending the message resource details anyway, why not just construct the URI fully and send it to the client?

Consider a sample response for a single message like this:

```
{
  "id": "1",
  "content": "Hello World!",
  "author": "naren",
  "postedDate": "03-01-2014",
  "href": "/messages/1"
}
```

If this were to be the kind of response for every message in `/messages`, then the client wouldn't really have to do any URI construction. The resource URI is one of the properties of the resource. If you were to design your API so that every resource has the instance resource URI to itself, it makes it really convenient for the client to use it.

8.3 Link relations

We are on our way to implementing some HATEOAS concepts. Let's look at the concept of links, and how you can apply them to the resources in the Messenger API.

We've looked at adding the resource URI to every resource. So, a profile resource, or a comment resource, well, pretty much every resource could have a `href` attribute that has the value of the instance resource URI. Below are the example links.

```
{
  "id": "1",
  "content": "Hello World!",
  "author": "naren",
  "postedDate": "03-01-2014",
  "href": "/messages/1",
  "comments-href": "/messages/1/comments",
  "likes-href": "/messages/1/likes",
  "shares-href": "/messages/1/shares",
  "profile-href": "/profiles/naren",
  "comment-post-href": "/messages/1/comments"
}
```

If you do this, the client doesn't need to remember the URIs, yes, **but they now have to remember the property names for these URIs and you basically have the same problem.**

There needs to be a better way to manage these links. And there is! **You can use the rel attribute.**

If you've used the anchor tags when writing HTML, you might have encountered this rel attribute before. It's basically an attribute that you can add to any link to specify the relationship between the current document and the linked document.

The most common example of rel is in stylesheet links. You'd have seen stylesheet links in HTML head tags like this:

```
<link rel="stylesheet" href="path/to/some.css"/>
```

Here href provides the actual URL being linked, and the rel attribute describes the relation of that link to the main document. Here the relation is that the link is a stylesheet of the main document.

We can use the **rel attribute** to add extra information in the links in our REST response. Here's the original href response modified with the rel attribute addition:

```
{
  "id": "1",
  "content": "Hello World!",
  "author": "koushik",
  "postedDate": "03-01-2014",
  "links": [
    {
      "href": "/messages/1",
      "rel": "self"
    }
  ]
}
```

What's different here is that we've introduced this new property called links which is an array. This is going to contain all the links that you'd want to embed in the response. However, you add the rel attribute to make it clear what the link points to. Notice the rel value self which indicates that the link in the resource points to itself.

This could be extended by adding new links and assigning the appropriate rel values for each:

```
{
  "id": "1",
  "content": "Hello World!",
  "author": "koushik",
```

```

    "postedDate": "03-01-2014",
    "links": [
      {
        "href": "/messages/1",
        "rel": "self"
      },
      {
        "href": "/messages/1/comments",
        "rel": "comments"
      },
      {
        "href": "/messages/1/likes",
        "rel": "likes"
      },
      {
        "href": "/messages/1/shares",
        "rel": "shares"
      },
      {
        "href": "/profiles/koushik",
        "rel": "author"
      }
    ]
  }

```

Now the client doesn't need to remember the link property values. They just have to find the link with the right rel value for the resource they want and then look up the href value from that link.

A couple of things to note here. While the concept of having the URIs in the response to achieve HATEOAS is something that's well understood and mostly agreed upon by all, the way to do this could vary differently among implementations.

The format of JSON that I've outlined here is just one of the multitude of ways you could structure links. Again, there's no right or wrong. You can choose to tweak how you want to present the links in the JSON response of your API depending on your preference. Secondly, the rel attribute is a part of the HTTP specification, so there are only certain standard values that are allowed for it.

This link lists the available values. And obviously, the rel values here like "comments" and "likes" are not valid. But we'll still use it.

In summary, HATEOAS is a way to provide links to resources in the API response, so that the client doesn't have to deal with URI construction and business flow.

9 The Richardson Maturity Model

Below is the API documentation summary of what we have so far. I hope the choices and the design approach for this API is clear to you now.

Messages:

Operation	URI	Method	Success / Failure	Status code
Get message	/messages/{messageId}	GET	Success	200
			Not found	404
			Failure	500
Delete message	/messages/{messageId}	DELETE	Success	200 or 204
			Not found	404
			Failure	500
Edit message	/messages/{messageId}	PUT	Success	200
			Wrong format / data	400 or 415
			Not found	404
			Failure	500
Create message	/messages	POST	Success	201
			Wrong format / data	400 or 415
			Failure	500

Messages:

Operation	URI	Method	Success / Failure	Status code
Get profile	/profiles/{profileName}	GET	Success	200
			Not found	404
			Failure	500
Delete profile	/profiles/{profileName}	DELETE	Success	200 or 204
			Not found	404
			Failure	500
Edit profile	/profiles/{profileName}	PUT	Success	200
			Wrong format / data	400 or 415
			Not found	404
			Failure	500
Create profile	/profiles	POST	Success	201
			Wrong format / data	400 or 415
			Failure	500

Comments (and similarly Likes and Shares):

Operation	URI	Method	Success / Failure	Status code
Get comment	/messages/{messageId}/comments/{commentId}	GET	Success	200
			Not found	404

Operation	URI	Method	Success / Failure	Status code
			Failure	500
Delete comment	/messages/{messageId}/comments/{commentId}	DELETE	Success	200 or 204
			Not found	404
			Failure	500
Edit comment	/messages/{messageId}/comments/{commentId}	PUT	Success	200
			Wrong format / data	400 or 415
			Not found	404
			Failure	500
Create comment	/messages/{messageId}/comments	POST	Success	201
			Wrong format / data	400 or 415
			Failure	500

Now that we have designed the API this way, let's look at what this means. Are we in a position to say this API is "fully RESTful"? Remember, in the first section I mentioned that this isn't a yes or no question, and that there is a spectrum of anywhere from "not fully RESTful" to "almost RESTful" to "not RESTful at all".

These terms are hard to work with. How do you know how RESTful an API is? Well, there is one way to know, and that's using a model developed by Leonard Richardson. It's called the Richardson Maturity Model, and it breaks down all the concepts we've discussed into 3 levels. Every REST API belongs to one of these 3 levels.

The model also defines a Level 0 which is not a RESTful API. It is not necessary that every API score highly as per this model. But it helps to understand this model when designing any RESTful API so that you at least know where you stand. And try to make it better if possible.

Let's start with Level 0. I hope you are familiar with some of the basics of a SOAP web service. The way a SOAP web service generally works is that there is a URL called the endpoint where the service is exposed. One URL. That URL receives all requests from the client. If you were to write the Messenger API as a SOAP web service, you'd probably have one URI at /messenger. This URL receives all requests. How does it know what to do? How does the client tell it to do different stuff, like look up messages or delete a comment? Well, that happens in the message that's sent to this common URL. The message contains both the operation that needs to be performed, and the data that's needed for that operation. For example, the XML below could create a new message:

```
<create-message>
  <message-content>Hello World!</message-content>
  <message-author>naren</message-author>
```

```
</create-message>
```

And a delete comment request (sent to the same URL) could look like this.

```
<delete-comment>  
  <message-id>30</message-id>  
  <comment-id>2</comment-id>  
</delete-comment>
```

Notice that the operation that needs to be performed is a part of the request that's sent. This is how the same URL can be used for different operations. In fact, the same HTTP method can be used for each operation, because, all the details are in the request body. In fact, that's what SOAP does. The requests are always POST, with the POST body containing all the information.

This is Level 0 in the Richardson Maturity Model. This is often called the The swamp of POX. This refers to the common use of Plain Old XML (or POX) to define everything that an operation needs. No HTTP concepts are leveraged for communicating information between the server and client.

This design approach is obviously not something we want to do in this course. If you were to refine this model to introduce the concept of resource URIs, you will reach level 1 in the RMM.

This is the starting level for RESTful APIs. The earlier level isn't even considered REST. We designed resource URIs for messages (/messages), profiles (/profiles) and so on. If you did just this, and nothing else, you stand at level 1. Now you have message requests going to one URI and all comments requests going to another URI. There would still be information about the operation in the requests, because the message URI needs to handle adding deleting or updating messages.

If you take the next step and use different HTTP methods for these different operations, then you've reached Level 2 in RMM. An API on Level 2 uses standard HTTP methods like GET, POST, PUT and DELETE to do different operations, on the resource URI. The URI specifies what resource is operated upon, and the HTTP method specifies what the operation is.

There also needs to be better use of HTTP status codes, and the right use of idempotent and non-idempotent methods for an API to be at Level 2.

Finally, Level 3 is when you implement HATEOAS. That is, the responses have links that control the application state for the client. The client doesn't need to be aware of the different API URIs. All the URIs that the client would need is a part of the response that the server sends. If an API implements this, it is said to be at Level 3 of RMM, and is considered fully RESTful.

And there you go. Now you can look at any REST API design and easily identify which level in RMM it belongs to. Again, this is not supposed to be a strict rule. I encourage you to use this model as a guideline when designing your REST APIs, as a tool for learning and understanding, rather than a scorecard to measure with. You may not choose to make every API achieve Level 3 of RMM, but it helps to understand what the theoretical ideal is.

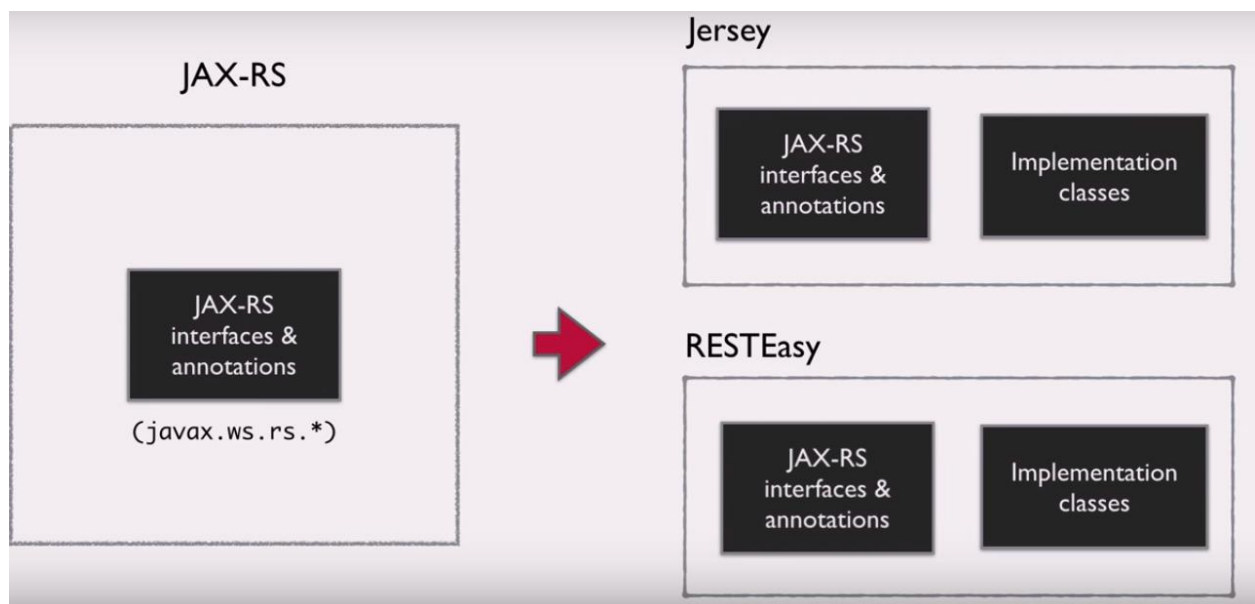
10 What is JAX-RS

In Java, we have libraries to write everything to implement, in the same we have libraries to implement REST API application. In market we have lot of libraries. Which library to choose

RestLet
RestEasy
Jersey,

They all follow common API, so if you know one implementation then its same for other API. This common API is called JAX-RS. All these implement this JAX-RS API.

Which library to learn, Jersey is reference implementation.



11 Setting up

Create a Maven project by select jersey artifact, default project can be deployed to any web or app server like Tomcat.

Prerequisites : Latest Java EE version of Eclipse installed (Make sure you install the Java EE version, not the Java version)

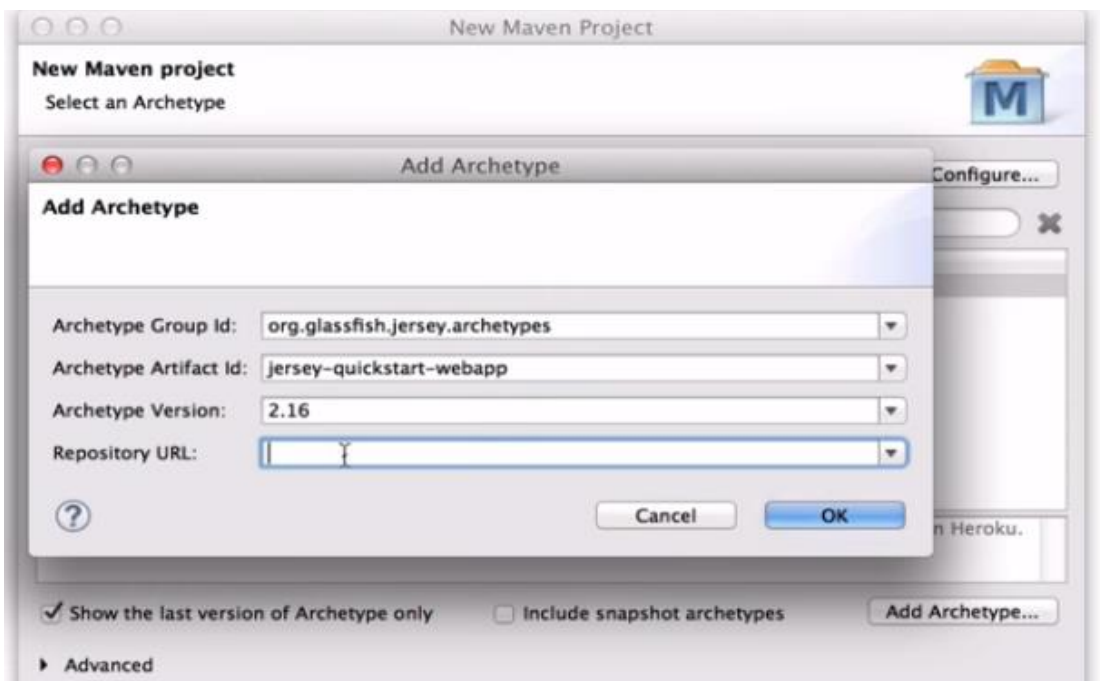
Steps

1. Open Eclipse and Choose New > Project > Maven Project
2. Choose "Add Archetype" and enter the following details:
Archetype Group ID: org.glassfish.jersey.archetypes
Archetype Artifact ID: jersey-quickstart-webapp
Archetype Version: 2.16
3. Choose the newly entered archetype from the Archetype selection screen
4. Enter your project details - Group ID, Artifact ID and version.
5. Setup Tomcat in your Eclipse workspace
6. Right click on the project and choose Run As > Run on server.

Maven

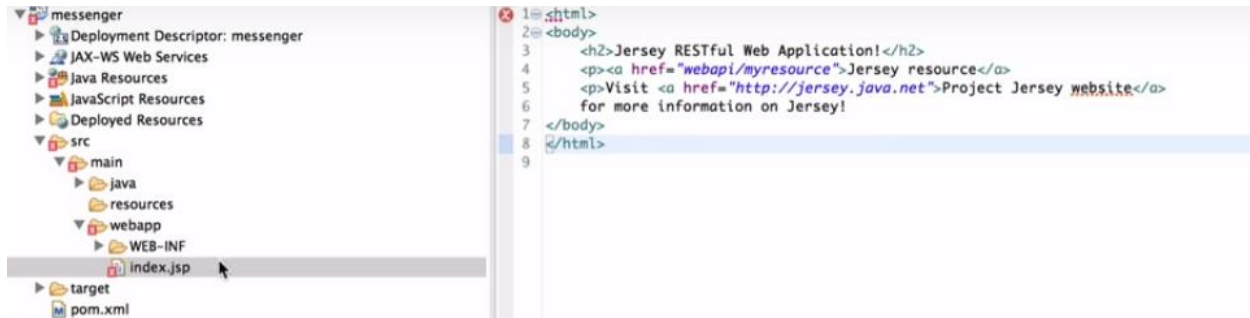
Take the Introduction to Maven course here:

http://javabrainz.koushik.org/courses/buildsys_mavenintro



12 Understanding the Application Structure

1. Rest is a webapplication
2. <http://localhost:8080/messenger> - is the url to access webapplication.
3. <http://localhost:8080/messenger/webapi> - is choosed for all the Rest Services.
4. <http://localhost:8080/messenger/webapi/myResource> - is a rest resource.



In web.xml –



13 Creating resource

Create Resource handler for messages, like below –

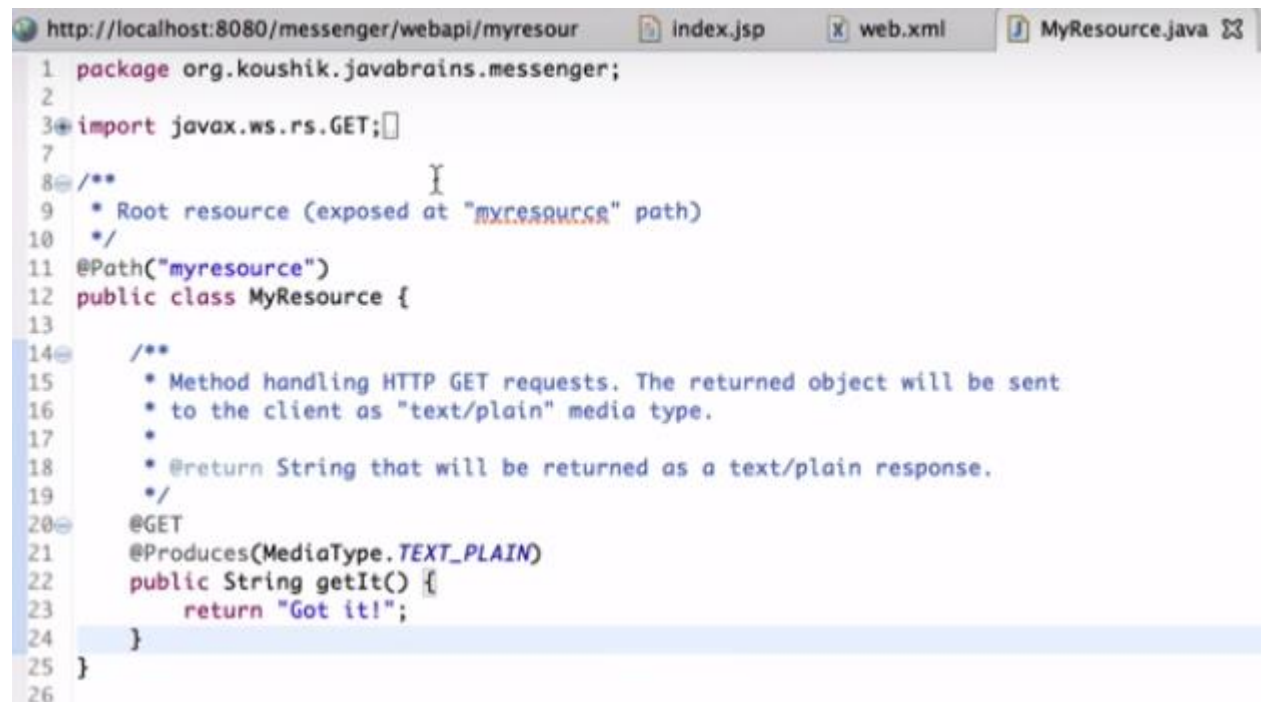
1. Create a java class to handle rest service – MessageResource.
2. Add method that returns response
3. Make sure class in the package configured in jersey servlet init param.
4. Annotate the class with @Path.
5. Annotate the method with @Get
6. Annotate the method with @Produce to specify response format

The above “`org.glassfish.jersey.servlet.ServletContainer`”, will look for below init param to read all the rest class in the project –

```
<init-param>
  <param-name>jersey.config.server.provider.packages</param-name>
  <param-value>org.koushik.javabrain.messenger</param-value>
</init-param>
```

matches the “url path” and Maps to a java class “MessageResource”

@Get annotations handles the get requests, Maps http method to a Java method “getMessage”.



```
http://localhost:8080/messenger/webapi/myresour  index.jsp  web.xml  MyResource.java
1 package org.koushik.javabrain.messenger;
2
3 import javax.ws.rs.GET;
4
5
6
7
8 /**
9  * Root resource (exposed at "myresource" path)
10  */
11 @Path("myresource")
12 public class MyResource {
13
14     /**
15      * Method handling HTTP GET requests. The returned object will be sent
16      * to the client as "text/plain" media type.
17      *
18      * @return String that will be returned as a text/plain response.
19      */
20     @GET
21     @Produces(MediaType.TEXT_PLAIN)
22     public String getIt() {
23         return "Got it!";
24     }
25 }
26
```

14 Return XML response

1. Create message class

```
package org.koushik.javabrainz.messenger.model;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlTransient;
@XmlRootElement
public class Message {
    private long id;
    private String message;
    private Date created;
    private String author;
    public Message() {
    }
    public Message(long id, String message, String author) {
        this.id = id;
        this.message = message;
        this.author = author;
        this.created = new Date();
    }

    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public Date getCreated() {
        return created;
    }
    public void setCreated(Date created) {
        this.created = created;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
}
```


2. Create another service class which return Messages, i.e MessageService.

```
package org.koushik.javabrainz.messenger.service;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.List;
import java.util.Map;
import org.koushik.javabrainz.messenger.database.DatabaseClass;
import org.koushik.javabrainz.messenger.model.Message;

public class MessageService {

    private Map<Long, Message> messages = DatabaseClass.getMessages();
    public MessageService() {
        messages.put(1L, new Message(1, "Hello World", "koushik"));
        messages.put(2L, new Message(2, "Hello Jersey", "koushik"));
    }
    public List<Message> getAllMessages() {
        return new ArrayList<Message>(messages.values());
    }
}
```

3. Annotate the Message class with **@XmlElement** to use JAXB to convert Java object to XML, to produce XML return type.
4. Use **@Produces(MediaType.APPLICATION_XML)** annotation on `getMessages()` method to return messages in xml format.
5. Below is the response after rest call on browser.

This XML file does not appear to have any style information associated with it. The document below.

```
<?xml version="1.0" encoding="UTF-8"?>
<messages>
  <message>
    <author>Koushik</author>
    <created>2015-03-01T23:19:57.438-08:00</created>
    <id>1</id>
    <message>Hello World!</message>
  </message>
  <message>
    <author>Koushik</author>
    <created>2015-03-01T23:19:57.438-08:00</created>
    <id>2</id>
    <message>Hello Jersey!</message>
  </message>
</messages>
```

15 PostMan as client

Use Postman as client for testing the rest services, this a plugin available on chrome

16 Accessing Path Param

```
@Path("/messages")
@Consumes(MediaType.TEXT_PLAIN)
@Produces(MediaType.APPLICATION_XML)
public class MessageResource{
    @GET
    @Path("/test")
    public Message getMessage(){
        return "TEST";
    }
}
```

@Path param at method level – appends the method level path **"/test"** to class level path **"/messages"**

Ex - <http://localhost:8080/messenger/webapi/messages/test>

Note : **@Path** can be used at class and method level, can be a **regular expression to capture** a pattern.

1. To tell jersey a portion of url is variable thing. Use curly braces like below to access

```
@GET
@Path("/{messageId}")
public Message getMessage(){
    return "TEST";
}
```

Ex –

<http://localhost:8080/messenger/webapi/messages/1345>

2. How can we get hold of messageId passed in?

```
@GET
@Path("/{messageId}")
public Message getMessage(@PathParam(messageId) String messageId){
    s.o.p("message Id from client" + messageId) – which can get hold of messageId from client
    return "TEST";
}
```

3. MessageId can be converted to target data Type if it's compatible.

```
@GET
@Path("/{messageId}")
public Message getMessage(@PathParam(messageId) Long messageId){
    s.o.p("message Id from client" + messageId) – which can get hold of messageId from client
    return "TEST";
}
```

17 Sample message Service

```
import java.net.URI;
import java.util.List;
import javax.ws.rs.*;
import org.koushik.javabrains.messenger.model.Message;
import org.koushik.javabrains.messenger.resources.beans.MessageFilterBean;
import org.koushik.javabrains.messenger.service.MessageService;

@Path("/messages")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class MessageResource {
    MessageService messageService = new MessageService();

    @GET
    public List<Message> getMessages(@BeanParam MessageFilterBean filterBean) {
        if (filterBean.getYear() > 0) {
            return messageService.getAllMessagesForYear(filterBean.getYear());
        }
        if (filterBean.getStart() >= 0 && filterBean.getSize() > 0) {
            return messageService.getAllMessagesPaginated(filterBean.getStart(),
filterBean.getSize());
        }
        return messageService.getAllMessages();
    }

    @POST
    public Response addMessage(Message message, @Context UriInfo uriInfo) {
        Message newMessage = messageService.addMessage(message);
        String newId = String.valueOf(newMessage.getId());
        URI uri = uriInfo.getAbsolutePathBuilder().path(newId).build();
        return Response.created(uri)
            .entity(newMessage)
            .build();
    }

    @PUT
    @Path("/{messageId}")
    public Message updateMessage(@PathParam("messageId") long id, Message message) {
        message.setId(id);
        return messageService.updateMessage(message);
    }

    @DELETE
    @Path("/{messageId}")
```

```

public void deleteMessage(@PathParam("messageId") long id) {
    messageService.removeMessage(id);
}

@GET
@Path("/{messageId}")
public Message getMessage(@PathParam("messageId") long id, @Context UriInfo uriInfo) {
    Message message = messageService.getMessage(id);
    message.addLink(getUriForSelf(uriInfo, message), "self");
    message.addLink(getUriForProfile(uriInfo, message), "profile");
    message.addLink(getUriForComments(uriInfo, message), "comments");
    return message;
}

private String getUriForComments(UriInfo uriInfo, Message message) {
    URI uri = uriInfo.getBaseUriBuilder()
        .path(MessageResource.class)
        .path(MessageResource.class, "getCommentResource")
        .path(CommentResource.class)
        .resolveTemplate("messageId", message.getId())
        .build();
    return uri.toString();
}

private String getUriForProfile(UriInfo uriInfo, Message message) {
    URI uri = uriInfo.getBaseUriBuilder()
        .path(ProfileResource.class)
        .path(message.getAuthor())
        .build();
    return uri.toString();
}

private String getUriForSelf(UriInfo uriInfo, Message message) {
    String uri = uriInfo.getBaseUriBuilder()
        .path(MessageResource.class)
        .path(Long.toString(message.getId()))
        .build()
        .toString();
    return uri;
}

@Path("/{messageId}/comments")
public CommentResource getCommentResource() {
    return new CommentResource();
}
}

```

18 Return JSON response

1. Use `MediaType.APPLICATION_JSON` in `@Produce` annotation
2. Include below dependency for JSON conversion

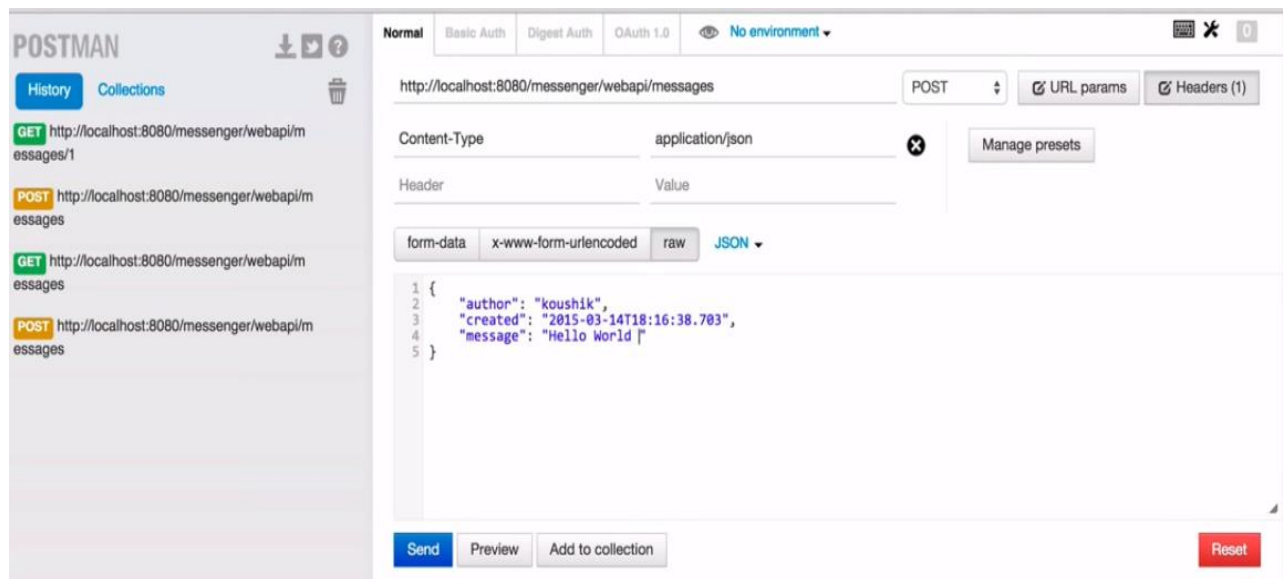
```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-moxy</artifactId>
</dependency>
```
3. Or Include below jar in class path
`jersey-media-json-jackson.jar`.

19 Implementing POST Method

```
@Post
public Message addMessage(){
    return "Post works";
}
```

To pass message object in json format, set `@consumes` annotation to the method

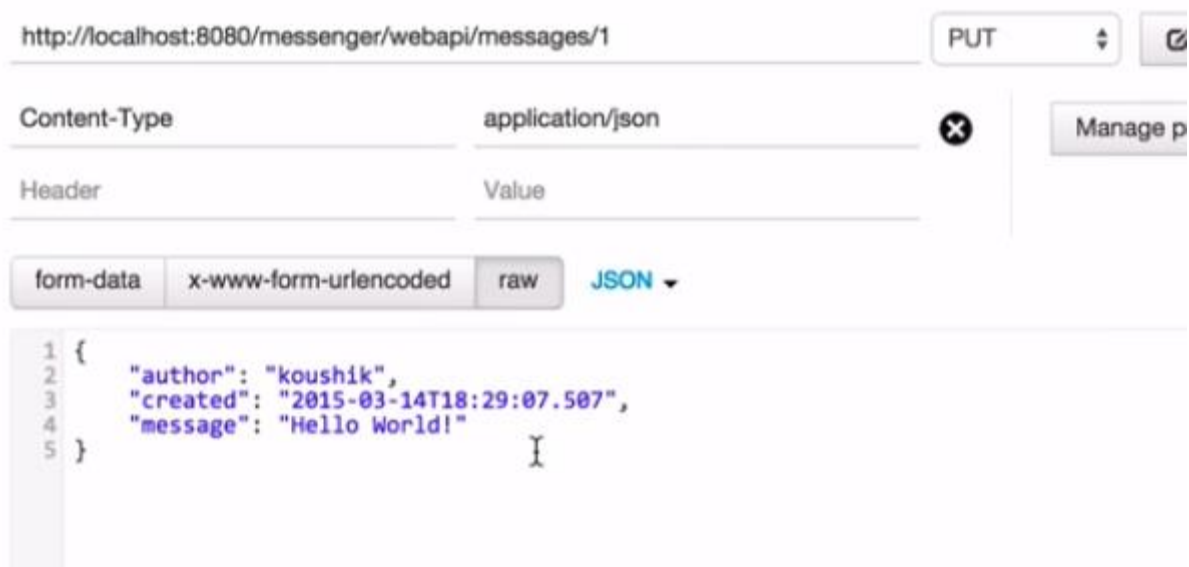
```
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@Post
public Message addMessage(Message message){
    addMessage(message)
    return message.getMessageId();
}
```



20 Implementing Update and Delete Method

20.1 Implementing PUT method for update

```
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@PUT
@Path("/{messageId}")
public Message updateMessage(@PathParam("messageId") long id, Message message) {
    message.setId(id);
    return messageService.updateMessage(message);
}
```



20.2 Implementing DELETE method

```
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@DELETE
@Path("/{messageId}")
public void deleteMessage(@PathParam("messageId") long id) {
    messageService.removeMessage(id);
}
```

```

@Path("/messages")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class MessageResource {

    MessageService messageService = new MessageService();

    @GET
    public List<Message> getMessages() {
        return messageService.getAllMessages();
    }

    @POST
    public Message addMessage(Message message) {
        return messageService.addMessage(message);
    }

    @PUT
    @Path("/{messageId}")
    public Message updateMessage(@PathParam("messageId") long id, Message message) {
        message.setId(id);
        return messageService.updateMessage(message);
    }

    @DELETE
    @Path("/{messageId}")
    public void deleteMessage(@PathParam("messageId") long id) {
        messageService.removeMessage(id);
    }

    @GET
    @Path("/{messageId}")
    public Message getMessage(@PathParam("messageId") long id) {
        return messageService.getMessage(id);
    }
}

```

21 Pagination and Filtering (@QueryParam)

@PathParam always expects a param in URL, but not @QueryParam.

Apply pagination and filtering by passing params to restservice like below

/messages?year=2016 && /messages?start=10&size=20

1. In MessageService, do this

```
public List<Message> getAllMessagesForYear(int year) {
    List<Message> messagesForYear = new ArrayList<>();
    Calendar cal = Calendar.getInstance();
    for (Message message : messages.values()) {
        cal.setTime(message.getCreated());
        if (cal.get(Calendar.YEAR) == year) {
            messagesForYear.add(message);
        }
    }
    return messagesForYear;
}

public List<Message> getAllMessagesPaginated(int start, int size) {
    ArrayList<Message> list = new ArrayList<Message>(messages.values());
    if (start + size > list.size()) return new ArrayList<Message>();
    return list.subList(start, start + size);
}
```

2. In MessageResource do this.

```
@GET
public List<Message> getMessages (@QueryParam("year") int year,
                                   @QueryParam("start") int start,
                                   @QueryParam("size") int size) {
    if (year > 0) {
        return messageService.getAllMessagesForYear(year);
    }
    if (start >= 0 && size > 0) {
        return messageService.getAllMessagesPaginated(start, size);
    }
    return messageService.getAllMessages();
}
```


22 Param Annotations

22.1 MatrixParam Annotations

Matrix Params are separated by semicolon (;).

```
GET
public List<Message>getMessages (@MatrixParam("param") String matrixParam) {
    s.o.p("matrix param" + matrixParam);
}
```

22.2 HeaderParam Annotations

```
GET
public List<Message>getMessages (@HeaderParam("headerParam") String headerParam) {
    s.o.p("header param" + headerParam);
}
```

22.3 CookieParam Annotations

```
GET
public List<Message>getMessages (@CookieParam("cookieParam") String cookieParam) {
    s.o.p("cookieParam" + cookieParam);
}
```

22.4 BeanParam Annotations

Instead of using different params in method definition we can do BeanParam like below by capturing all the params in a Bean class.

```
@GET
public List<Message>getMessages (@BeanParam MessageFilterBean filterBean) {
    if (filterBean.getYear() > 0) {
        return messageService.getAllMessagesForYear(filterBean.getYear());
    }
    if (filterBean.getStart() >= 0 && filterBean.getSize() > 0) {
        return messageService.getAllMessagesPaginated(filterBean.getStart(),
            filterBean.getSize());
    }
    return messageService.getAllMessages();
}
```

MessageFilterBean class

```
import javax.ws.rs.QueryParam;
public class MessageFilterBean {
    private @QueryParam("year") int year;
    private @QueryParam("start") int start;
    private @QueryParam("size") int size;
    // getters and setters.
}
```

```
}
```

22.5 Context Annotations

@Context annotation will be used to pull all the different parameters, like queryParams, pathParam, and lot more other information.

```
@POST
public Response addMessage(Message message, @Context UriInfo uriInfo) {

    Message newMessage = messageService.addMessage(message);
    String newId = String.valueOf(newMessage.getId());
    URI uri = uriInfo.getAbsolutePathBuilder().path(newId).build();
    return Response.created(uri)
        .entity(newMessage)
        .build();
}

@POST
public Response addMessage(Message message, @Context HttpHeaders headers) {
    headers.setCookie();
}
```

23 Implementing Sub Resources

Pulling the sub resources is fairly easy, delegating request to “Sub Resource” by returning corresponding RestFull resource.

In MessageResource

```
@Path("/{messageId}/comments")
public CommentResource getCommentResource() {
    return new CommentResource();
}
```

In CommentResource

```
@Path("/")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class CommentResource {

    private CommentService commentService = new CommentService();

    @GET
    public List<Comment> getAllComments(@PathParam("messageId") long messageId) {
        return commentService.getAllComments(messageId);
    }

    @GET
    @Path("/{commentId}")
    public Comment getMessage(@PathParam("messageId") long messageId,
                             @PathParam("commentId") long commentId) {
        return commentService.getComment(messageId, commentId);
    }
}
```

To get the comments it looks URL path like below –

```
messages/{messageId}/comments/{commentId}
```

24 Sending Status codes and Location Headers

When we create a message, return both the entity and status code, we can set cookies, uri and whole lot of things.

To do that use Response builder to create **Response** and return.

```
@POST
public Response addMessage(Message message, @Context UriInfo uriInfo) {
    Message newMessage = messageService.addMessage(message);
    return Response.status(Status.CREATED)
        .entity(newMessage)
        .build();
}
```

There are some shortcut method on Response builder to set the status and locations, see a sample method snippet for that.

```
@POST
public Response addMessage(Message message, @Context UriInfo uriInfo) {
    Message newMessage = messageService.addMessage(message);
    String newId = String.valueOf(newMessage.getId());
    URI uri = uriInfo.getAbsolutePathBuilder().path(newId).build();
    return Response.created(uri)
        .entity(newMessage)
        .build();
}
```

25 Exception handling

```
@GET
@Path("/{messageId}")
public Message updateMessage(@PathParam("messageId") long id, Message message) {
    if messageId == null)
        throw RuntimeException ("no messageId");
    return messageService.updateMessage(message);
}
```

In JAX-Rs exception handling can be done by using `ExceptionHandler`, every exception mapper needs to implement `ExceptionHandler`, it's a raw type. We need to provide corresponding exception in the type like below.

`DataNotFoundException` is define in the project for no data found cases.

```
@Provider
public class DataNotFoundExceptionMapper implements ExceptionMapper<DataNotFoundException> {
    @Override
    public Response toResponse(DataNotFoundException ex) {
        ErrorMessage errorMessage = new ErrorMessage(ex.getMessage(), 404,
"http://javabrainz.koushik.org");
        return Response.status(Status.NOT_FOUND)
            .entity(errorMessage)
            .build();
    }
}
```

@Provider – registries the `ExceptionHandler` in JAX-RS.

In case of an exception, JAX-RS look up for all the registered exception mappers and calls its corresponding `toResponse()`.

Below is a generic exception mapper.

```
@Provider
public class GenericExceptionHandler implements ExceptionMapper<Throwable> {
    @Override
    public Response toResponse(Throwable ex) {
        ErrorMessage errorMessage = new ErrorMessage(ex.getMessage(), 500,
"http://javabrainz.koushik.org");
        return Response.status(Status.INTERNAL_SERVER_ERROR)
            .entity(errorMessage)
            .build();
    }
}
```

25.1 Using WebApplicationException

There are bunch predefined exceptions defined in JAX-RS, we can use any of those exceptions as per the scenarios.

In commentsService –

```
public Comment getComment(long messageId, long commentId) {
    ErrorMessage errorMessage = new ErrorMessage("Not found", 404,
"http://javabrainz.koushik.org");
    Response response = Response.status(Status.NOT_FOUND)
        .entity(errorMessage)
        .build();

    Message message = messages.get(messageId);
    if (message == null) {
        throw new WebApplicationException(response);
    }
    Map<Long, Comment> comments = messages.get(messageId).getComments();
    Comment comment = comments.get(commentId);
    if (comment == null) {
        throw new NotFoundException(response);
    }
    return comment;
}
```

26 HATEOAS

```
@GET
@Path("/{messageId}")
public Message getMessage(@PathParam("messageId") long id, @Context UriInfo uriInfo) {
    Message message = messageService.getMessage(id);
    message.addLink(getUriForSelf(uriInfo, message), "self");
    message.addLink(getUriForProfile(uriInfo, message), "profile");
    message.addLink(getUriForComments(uriInfo, message), "comments");
    return message;
}

private String getUriForComments(UriInfo uriInfo, Message message) {
    URI uri = uriInfo.getBaseUriBuilder()
        .path(MessageResource.class)
        .path(MessageResource.class, "getCommentResource")
        .path(CommentResource.class)
        .resolveTemplate("messageId", message.getId())
        .build();
    return uri.toString();
}

private String getUriForProfile(UriInfo uriInfo, Message message) {
    URI uri = uriInfo.getBaseUriBuilder()
        .path(ProfileResource.class)
        .path(message.getAuthor())
        .build();
    return uri.toString();
}

private String getUriForSelf(UriInfo uriInfo, Message message) {
    String uri = uriInfo.getBaseUriBuilder()
        .path(MessageResource.class)
        .path(Long.toString(message.getId()))
        .build()
        .toString();
    return uri;
}
```

In Message class

```
public void addLink(String url, String rel) {
    Link link = new Link();
    link.setLink(url);
    link.setRel(rel);
    links.add(link);
}
```

Link class

```
public class Link {  
    private String link;  
    private String rel;  
    public String getLink() {  
        return link;  
    }  
    public void setLink(String link) {  
        this.link = link;  
    }  
    public String getRel() {  
        return rel;  
    }  
    public void setRel(String rel) {  
        this.rel = rel;  
    }  
}
```

Note: There is a class called Link in Jax-RS for Link.

27 Bootstrap JAX-RS Application

1. We don't need to use definition in web.xml of web application to bootstrap, instead we can use special class call "Application" to boot strap Jersey apps.
2. Extending javax.ws.rs.core.Application, which reads all the resource class, which defines your Jersey application.
3. Where the resources are, it looks at all the classes in our class path and locates all the resources, we still need to setup to only look at only specific packages.
4. Setting up app context - **@ApplicationPath("webapi")**, this will take all the web resource in class path and creates a resource like - /webapi/comments.
5. **Application** is an abstract class, which has some method, not no methods to implement in sub class. We can still implement all the methods in sub class

BootStrap MyApp -

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

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

@ApplicationPath("webapi")
public class MyApp extends Application {

}
```

MyResource class

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("test")
public class MyResource {
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String testMethod() {
        return "It works!";
    }
}
```

28 Resource Life Cycle

By default resource is in a request scope.

```
@Path("test")
public class MyResource {
    private int count=0;
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String testMethod(){
        count = count+1;
        s.o.p("my resource count" + count);
        return "It works!";
    }
}
```

In each request, the count will not be changed, because for each request a new resource instance will be created.

How can we change this behavior and create a singleton resource.

Just use a **@singleton**.

```
@Path("test")
@singleton
public class MyResource {
    private int count=0;
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String testMethod(){
        count = count+1;
        s.o.p("the method called count" + count);
        return "It works!";
    }
}
```

Now, method call count will be incremented.

29 Param Annotations and Member Variables

So far, you have QueryParam, PathParam at method level. We can have these params in class level (resource level).

Member variable instead of method variable.

```
@Path("{pathParam}/test")
public class MyResource {
    @PathParam("pathParam") private String pathParamExample;
    @QueryParam("query") private String queryParamExample;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String testMethod() {
        return "It works! Path param used " + pathParamExample + " and Query param used " + queryParamExample;
    }
}
```

Following is URL - <http://localhost:8080/App/webapi/value/test?query=queryValue>

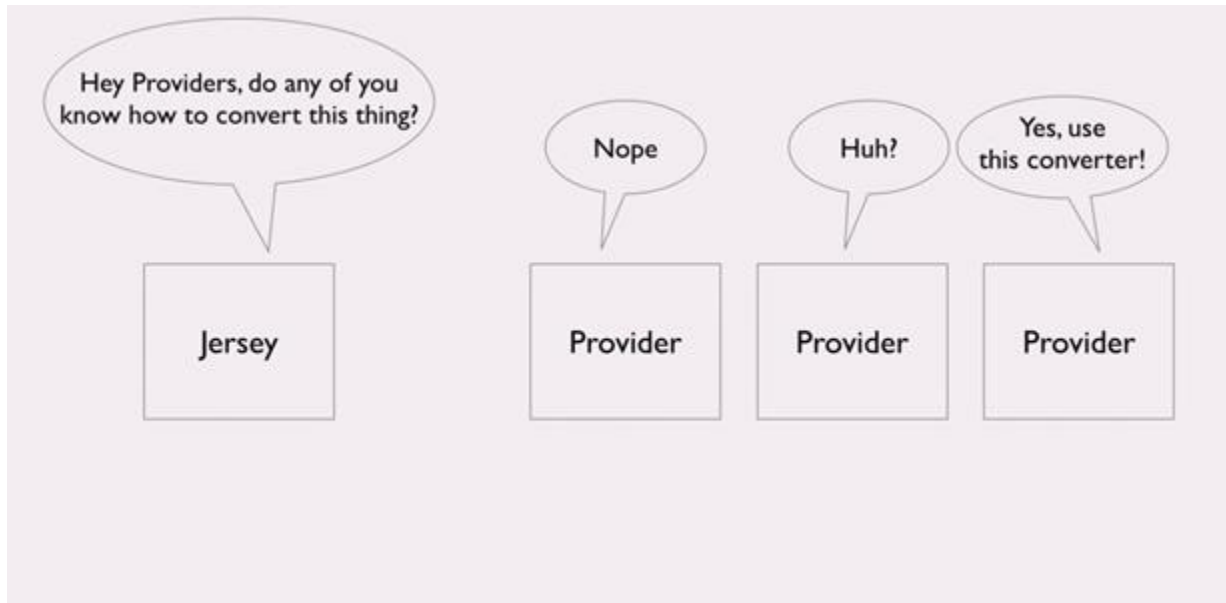
In we makes our resource as @singleton, then there is a problem, injection of member variable is not possible. Because Singleton object is created well before restful request. So you can inject a value to a singleton, hence you can't have member level variables with path are query params.

But it's possible with member variables.

30 Param Converters

There are implicit conversion happens for primitive types, Jersey can handles it by using bunch of converters provided by Jersey.

30.1 Custom Converters



DateResource is converts string to date by using custom date converter.

```
@Path("date/{dateString}")
public class DateResource {
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getRequestedDate(@PathParam("dateString") MyDate myDate) {
        return "Got " + myDate.toString();
    }
}

public class MyDate {
    private int date;
    private int month;
    private int year;
    // settrs and getters
    @Override
    public String toString() {
        return "MyDate [date=" + date + ", month=" + month + ", year=" + year
            + "]\n";
    }
}
```

```

import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import java.util.Calendar;
import javax.ws.rs.ext.ParamConverter;
import javax.ws.rs.ext.ParamConverterProvider;
import javax.ws.rs.ext.Provider;

```

@Provider

```

public class MyDateConverterProvider implements ParamConverterProvider {
    @Override
    public <T> ParamConverter<T> getConverter(final Class<T> rawType, Type genericType,
Annotation[] annotations) {
        if (rawType.getName().equals(MyDate.class.getName())) {
            return new ParamConverter<T>() {
                @Override
                public T fromString(String value) {
                    Calendar requestedDate = Calendar.getInstance();
                    if ("tomorrow".equalsIgnoreCase(value)) {
                        requestedDate.add(Calendar.DATE, 1);
                    } else if ("yesterday".equalsIgnoreCase(value)) {
                        requestedDate.add(Calendar.DATE, -1);
                    }
                    MyDate myDate = new MyDate();
                    myDate.setDate(requestedDate.get(Calendar.DATE));
                    myDate.setMonth(requestedDate.get(Calendar.MONTH));
                    myDate.setYear(requestedDate.get(Calendar.YEAR));
                    return rawType.cast(myDate);
                }
            };
        }
        @Override
        public String toString(T myBean) {
            if (myBean == null) {
                return null;
            }
            return myBean.toString();
        }
    }
}

```

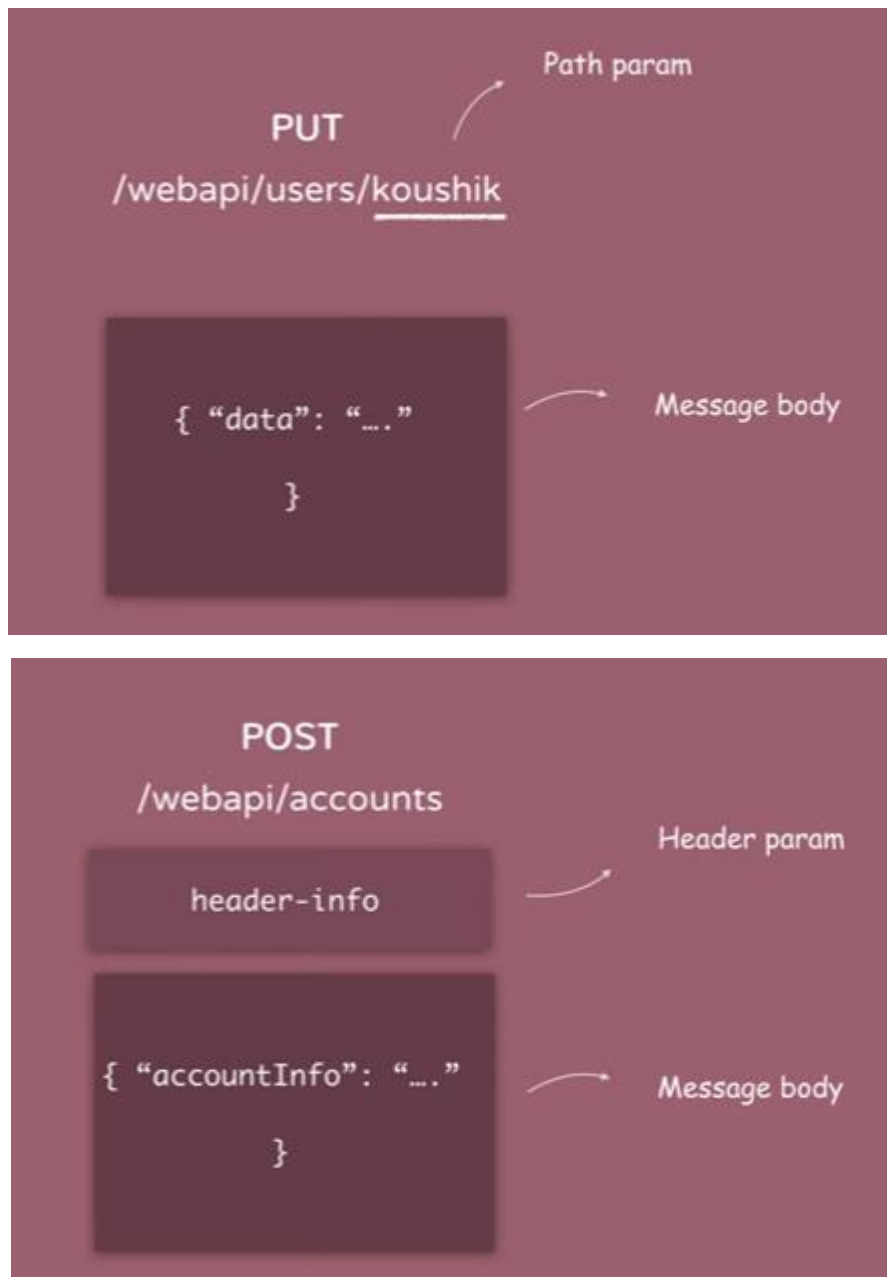
31 MessageBodyReaders and MessageBodyWriters

Method level Producer and consumers of the useful to consume the data from client to into jersey method and from jersey method to client.

MessageBodyReaders and MessageBodyWriters are useful to do this conversion.

31.1 MessageBody vs Param

Message body has some payload., content in the request body and response body, Params are just params to methods.



31.2 Implementing MessageBodyWriters

Method level Producer and consumers of the useful to consume the data from client to into jersey method and from jersey method to client.

Below class will handle date type and converts it into string and passes it to Client as a response.

Ex-

```
import java.io.IOException;
import java.io.OutputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import java.util.Date;
import javax.ws.rs.*;

@Provider
@Produces(MediaType.TEXT_PLAIN)
public class DateMessageBodyWriter implements MessageBodyWriter<Date> {
    @Override
    public long getSize(Date arg0, Class<?> arg1, Type arg2, Annotation[] arg3,
        MediaType arg4) {
        return -1;
    }
    @Override
    public boolean isWriteable(Class<?> type, Type arg1, Annotation[] arg2,
        MediaType arg3) {
        return Date.class.isAssignableFrom(type);
    }

    @Override
    public void writeTo(Date date, Class<?> type,
        Type type1, Annotation[] antns, MediaType mt,
        MultivaluedMap<String, Object> mm,
        OutputStream out) throws IOException, WebApplicationException {
        out.write(date.toString().getBytes());
    }
}
```

31.3 Implementing Custom Media Type

To implement custom media type just implement a custom message body writer and produce the desired media type

```
import java.io.IOException;
import java.io.OutputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import java.util.Date;
import javax.ws.rs.*;

@Provider
@Produces("text/shortdate")
public class ShortDateMessageBodyWriter implements MessageBodyWriter<Date>{
    @Override
    public long getSize(Date arg0, Class<?> arg1, Type arg2, Annotation[] arg3,
        MediaType arg4) {
        return -1;
    }
    @Override
    public boolean isWriteable(Class<?> type, Type arg1, Annotation[] arg2,
        MediaType arg3) {
        return Date.class.isAssignableFrom(type);
    }
    @Override
    public void writeTo(Date date, Class<?> type,
        Type type1, Annotation[] antns,
        MediaType mt, MultivaluedMap<String, Object> mm,
        OutputStream out) throws IOException, WebApplicationException {
        String shortDate = date.getDate() + "-" + date.getMonth() + "-" + date.getYear();
        out.write(shortDate.getBytes());
    }
}

@Path("test")
public class MyResource {
    @GET
    @Produces(value = { MediaType.TEXT_PLAIN, "text/shortdate" })
    public Date testMethod() {
        return Calendar.getInstance().getTime();
    }
}
```


32 JAX-RS Client

There is a client API from jersey 2.0, Before 2.0 there are bunch of API's which are used to invoke restful webservises from Java.

32.1 JAX-RS Client code

Ex -

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import org.koushik.javabrainz.messenger.model.Message;

public class RestApiClient {
    public static void main(String[] args) {
        Client client = ClientBuilder.newClient();
        WebTarget baseTarget =
            client.target("http://localhost:8080/advanced-jaxrs-06/webapi/");
        WebTarget messagesTarget = baseTarget.path("messages");
        WebTarget singleMessageTarget = messagesTarget.path("{messageId}");

        Message message1 = singleMessageTarget
            .resolveTemplate("messageId", "1")
            .request(MediaType.APPLICATION_JSON)
            .get(Message.class);
        Message message2 = singleMessageTarget
            .resolveTemplate("messageId", "2")
            .request(MediaType.APPLICATION_JSON)
            .get(Message.class);

        Message newMessage = new Message(4, "JAX-RS client msg", "koushik");
        Response postResponse = messagesTarget
            .request()
            .post(Entity.json(newMessage));
        if (postResponse.getStatus() != 201) {
            System.out.println("Error");
        }
        Message createdMessage = postResponse.readEntity(Message.class);
    }
}
```

```

        System.out.println(createdMessage.getMessage());
    }
}

```

32.2 Creating Invocations

Create invocation object to execute the get or post method. This is to make available Get or Post invocation objects

1. Build the request
2. Invoke the request

Ex –

```

public class InvocationDemo {
    public static void main(String[] args) {
        InvocationDemo demo = new InvocationDemo();
        Invocation invocation =
demo.prepareRequestForMessagesByYear(2015);
        Response response = invocation.invoke();
        System.out.println(response.getStatus());
    }
    public Invocation prepareRequestForMessagesByYear(int year) {
        Client client = ClientBuilder.newClient();
        return client.target("http://localhost:8080/advanced-jaxrs-06/webapi/")
            .path("messages")
            .queryParams("year", year)
            .request(MediaType.APPLICATION_JSON)
            .buildGet();
    }
}

```

32.3 Handling Generics

```
import java.util.List;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

public class GenericDemo {
    public static void main(String[] args) {
        Client client = ClientBuilder.newClient();
        List<Message> messages =
            client.target("http://localhost:8080/advanced-jaxrs-06/webapi/")
                .path("messages")
                .queryParams("year", 2015)
                .request(MediaType.APPLICATION_JSON)
                .get(new GenericType<List<Message>>() { });
        System.out.println(messages);
    }
}
```

33 Rest API Authentication

33.1 Implementing Filters

Cross cutting concern, extract common logic and apply it across all the resources.

```
import java.io.IOException;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.ext.Provider;

@Provider
public class LoggingFilter implements ContainerRequestFilter, ContainerResponseFilter {
    @Override
    public void filter(ContainerRequestContext requestContext) throws IOException {
        System.out.println("Request filter");
        System.out.println("Headers: " + requestContext.getHeaders());
    }
    @Override
    public void filter(ContainerRequestContext requestContext,
        ContainerResponseContext responseContext) throws IOException {
        System.out.println("Response filter");
        System.out.println("Headers: " + responseContext.getHeaders());
    }
}
```

33.2 Basic Authorization

```
import java.io.IOException;
import java.util.List;
import java.util.StringTokenizer;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.Provider;
import org.glassfish.jersey.internal.util.Base64;
```

@Provider

```
public class SecurityFilter implements ContainerRequestFilter {
```

```
    private static final String AUTHORIZATION_HEADER_KEY = "Authorization";
```

```
    private static final String AUTHORIZATION_HEADER_PREFIX = "Basic";
```

```
    private static final String SECURED_URL_PREFIX = "secured";
```

@Override

```
    public void filter(ContainerRequestContext requestContext) throws IOException {
```

```
        if (requestContext.getUriInfo().getPath().contains(SECURED_URL_PREFIX)) {
```

```
            List<String> authHeader =
```

```
                requestContext.getHeaders().get(AUTHORIZATION_HEADER_KEY);
```

```
            if (authHeader != null && authHeader.size() > 0) {
```

```
                String authToken = authHeader.get(0);
```

```
                authToken =
```

```
                    authToken.replaceFirst(AUTHORIZATION_HEADER_PREFIX, "");
```

```
                    String decodSt = Base64.decodeAsString(authToken);
```

```
                    StringTokenizer tokenizer = new StringTokenizer(decodSt, ":");
```

```
                    String username = tokenizer.nextToken();
```

```
                    String password = tokenizer.nextToken();
```

```
                    if ("user".equals(username) && "password".equals(password)) {
```

```
                        return;
```

```
                    }
```

```
            }
```

```
            Response unauthorizedStatus = Response
```

```
                .status(Response.Status.UNAUTHORIZED)
```

```
                .entity("User cannot access the resource.")
```

```
                .build();
```

```
            requestContext.abortWith(unauthorizedStatus);
```

```
        }
```

```
    }
```

```
}
```

34 Filter and Interceptors

Filters modifies header information or meta data information

Interceptors manipulates entities (input and output streams).

Two kind of :

1. ReaderInterceptors
2. WriterInterceptors

```
public class GZIPWriterInterceptor implements WriterInterceptor {  
    @Override  
    public void aroundWriteTo(WriterInterceptorContext context)  
        throws IOException, WebApplicationException {  
        final OutputStream outputStream = context.getOutputStream();  
        context.setOutputStream(new GZIPOutputStream(outputStream));  
        context.proceed();  
    }  
}
```

Interceptors

- Used to manipulate entities (input and output streams)
- Two kinds:
 1. ReaderInterceptor
 2. WriterInterceptor
- Example: Encoding an entity response

vs

Filters

- Used to manipulate request and response params (headers, URIs etc)
- Two kinds:
 1. ContainerRequestFilter
 2. ContainerResponseFilter
- Example: Logging, security

Not: You can apply filters and interceptors in client side as well.



Order of filters and interceptors execution.



35 Conclusion

What we covered

- Setting up a container-agnostic JAX-RS Application
- Resource Life Cycle
- Custom ParamConverters
- MessageBodyReaders and MessageBodyWriters
- Custom Media Types
- Writing a JAX-RS client
- Making GET and POST request
- Handling generic types in the client
- Implementing Filters
- Understanding REST API authentication
- Implementing Basic Auth
- Interceptors, Request response flow, EJBs