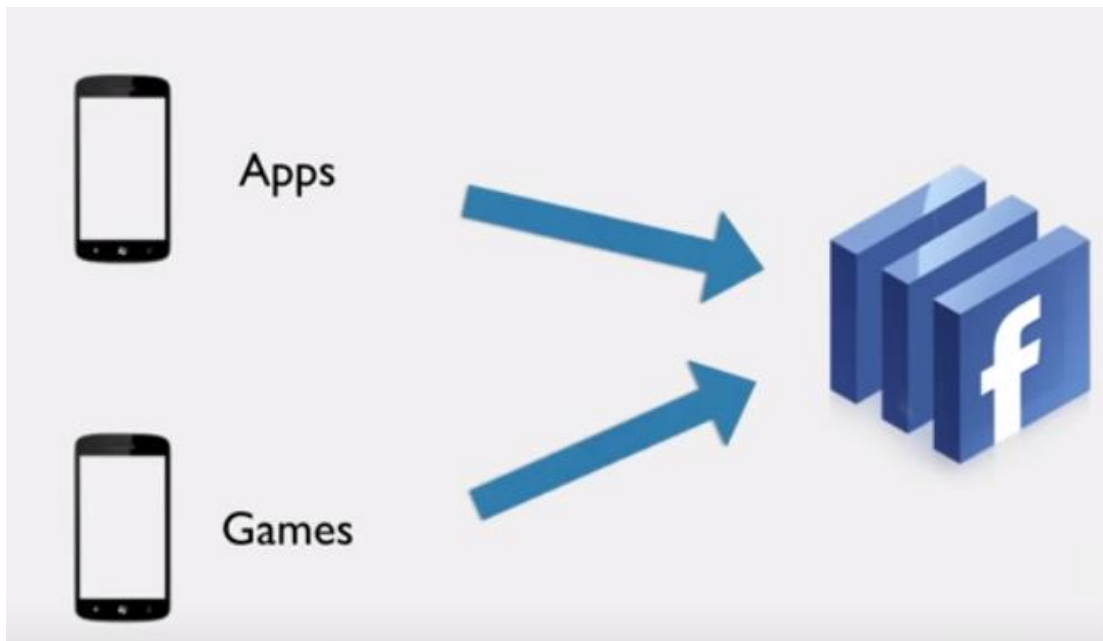
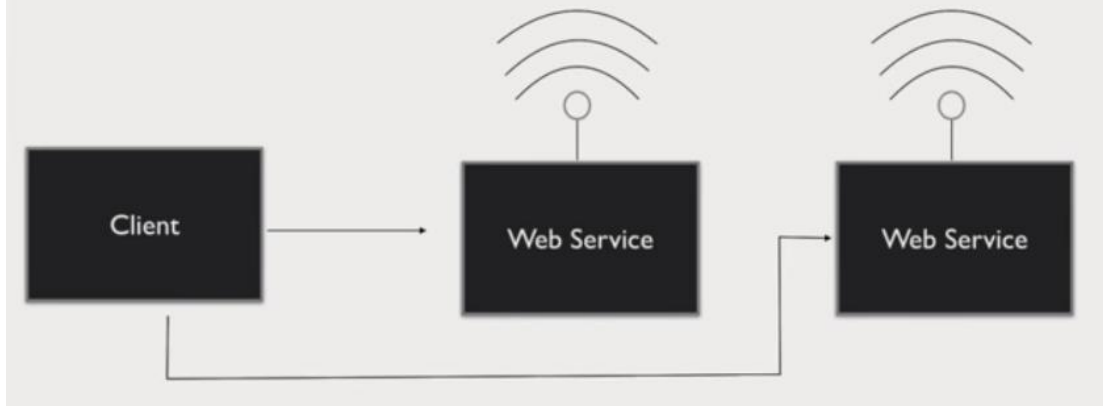


Web Services





SOAP:

Service definitions are defined in WSDL.

When we share this to a client they got to know all information about the SOAP.

REST:

No Service documentation required. Best REST web services does not require any documentation.

Addresses

Weather website:

`weatherapp.com/weatherLookup.do?zipcode=12345`

Resource based URI

`weatherapp.com/zipcode/12345`

`weatherapp.com/zipcode/56789`

`weatherapp.com/countries/brazil`

HTTP Methods

GET

POST

PUT

DELETE

HTTP Status codes

200 - Success

500 - Server error

404 - Not found

Message headers

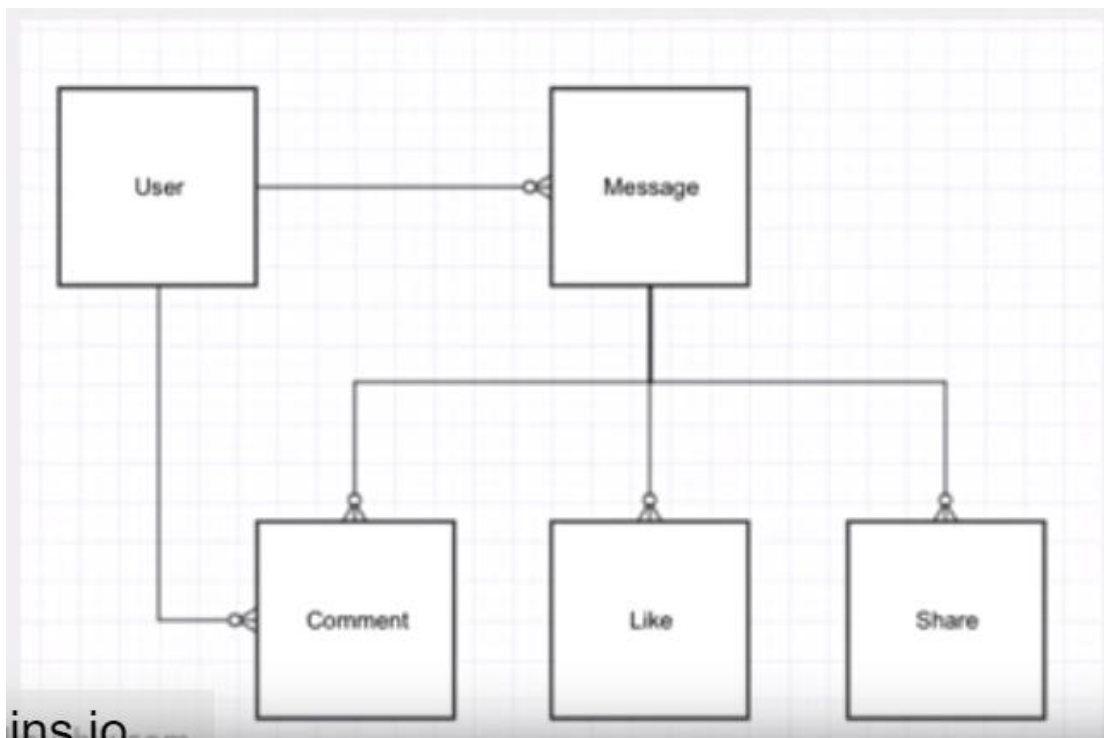
Content types

`text/xml`

`application/json`

Summary

- Resource based URIs
- HTTP methods
- HTTP status codes
- Message headers



Consumers need to remember the resource

based URIs.

Resource based URI

`/messages/{messageId}`

`/messages/1`

`/messages/10`

Resource URIs

`/profiles/{profileName}`

`/messages/{messageId}`

first level

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

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

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

second level

Examples

`/messages/{messageId}/comments`  represents all comments
for message {messageId}

`/messages/{messageId}/likes`

`/messages/{messageId}/shares`

Filtering results

`/messages?offset=30&limit=10`  starting point
 page size

Summary

- Two types of resource URIs
 - Instance resource URIs
 - Collection resource URIs
- Query parameters for pagination and filtering coll

URIs

 action based
`/getMessages.do?id=10`



`/messages/10`
 resource based

HTTP Methods

`/getProducts.do?id=10` ❌

`/deleteOrder.do?id=10` ❌



`/products/10`



`/orders/10`




Method Idempotence:

Common HTTP Methods

safely repeatable
(Idempotent)

GET POST
PUT DELETE

cannot be repeated
(Non-idempotent)



The diagram shows a cloud containing the HTTP methods GET, POST, PUT, and DELETE. An arrow points from the cloud to the text 'safely repeatable (Idempotent)' on the left, which includes the word 'Common' in red. Another arrow points from the cloud to the text 'cannot be repeated (Non-idempotent)' on the right.

Idempotent

PUT

VS

POST

Non-idempotent

The diagram compares PUT and POST methods. An arrow points from the word 'Idempotent' to 'PUT'. Another arrow points from the word 'Non-idempotent' to 'POST'. The word 'VS' is placed between 'PUT' and 'POST'.

Message Entity

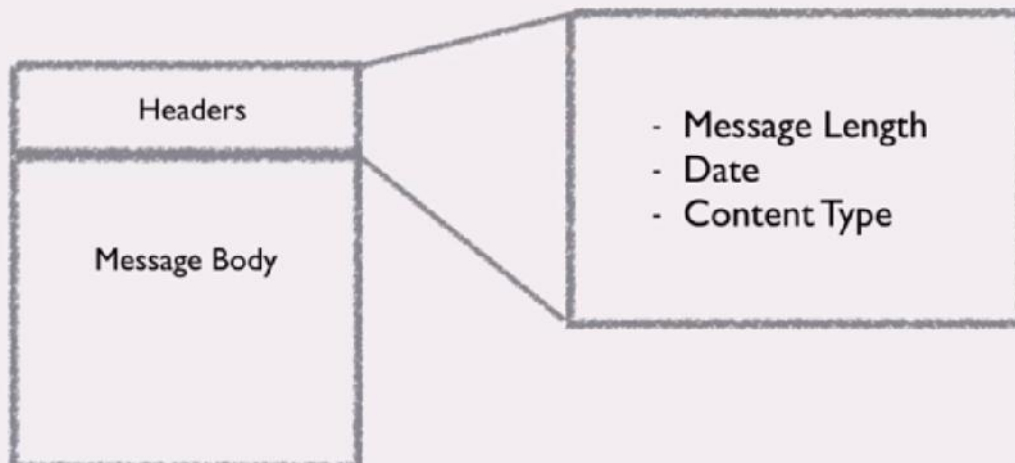
```
public class MessageEntity {  
    private long id;  
    private String message;  
    private Date created;  
    private String author;  
    ...  
}
```

JSON Response for a message

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

XML Response for the same message

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



Status Codes

1XX Informational

2XX Success

3XX Redirection

4XX Client Error

5XX Server Error

2XX - Success Codes

200 OK

201 Created

204 No Content

3XX - Redirection Codes

302 Found

304 Not Modified

307 Temporary Redirect



4XX - Client Error Codes

400 Bad Request

401 Unauthorized

403 Forbidden

404 Not Found

415 Unsupported Media Type



HATEOAS:

Hypermedia As The Engine Of Application State

A way to provide links to the API Responses,

The “rel” attribute

```
{
  "id": "1",
  "content": "Hello World!",
  "author": "koushik",
  "created": "2015-01-01T12:00:00.000",
  "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"
    }
  ]
}
```

rains.koushik.org

The Richardson Maturity Model:

Is this API fully Restful ?

Level 3

Level 2

Level 1

Level 0

Level 0:

It's like SOAP. Everything we mention inside body, i.e add a resource or delete a resource. No http methods come into picture.

Uses plain old XML to provide all information about the data and either add and/or delete.

Level 1

Individual URIs
for each resource

still http methods not used.

HTTP Methods

Level 2

Uses the right HTTP methods, status codes

HATEOAS

Level 3

Responses have links that the clients can use

JAX-RS:



=====

Udemy:

Web Service

Service delivered over the web ?

WEB

|

|

BUSINESS

|

|

DATA

W3C Definition:

Software system designed to support interoperable machine-to-machine interaction over a network.

Should be interoperable-Not platform dependent.

Should allow communication over a network.

there should be a service definition

it contains,

1. Req/Res format
2. Endpoints
3. Request Structure
4. Response Structure

Transport:
http or Queue

=====

Jackson-annotations:

2.1. *@JsonAnyGetter*

The *@JsonAnyGetter* annotation allows the flexibility of using a *Map* field as standard properties.

Here's a quick example – the *ExtendableBean* entity has the *name* property and a set of extendable attributes in the form of key/value pairs:

```
1 public class ExtendableBean {  
2     public String name;  
3     private Map<String, String> properties;  
4  
5     @JsonAnyGetter  
6     public Map<String, String> getProperties() {  
7         return properties;  
8     }  
9 }
```

When we serialize an instance of this entity, we get all the key-values in the *Map* as standard, plain properties:

```
1 {  
2     "name": "My bean",  
3     "attr2": "val2",  
4     "attr1": "val1"  
5 }
```

2.3. @JsonPropertyOrder

The `@JsonPropertyOrder` annotation is used to specify **the order of properties on serialization**.

Let's set a custom order for the properties of a *MyBean* entity:

```
1 | @JsonPropertyOrder({ "name", "id" })
2 | public class MyBean {
3 |     public int id;
4 |     public String name;
5 | }
```

And here is the output of serialization:

```
1 | {
2 |     "name": "My bean",
3 |     "id": 1
4 | }
```

2.4. @JsonRawValue

`@JsonRawValue` is used to instruct the Jackson to serialize a property exactly as is.

In the following example – we use `@JsonRawValue` to embed some custom JSON as a value of an entity:

```
1 | public class RawBean {
2 |     public String name;
3 |
4 |     @JsonRawValue
5 |     public String json;
6 | }
```

The output of serializing the entity is:

```
1 | {
2 |     "name": "My bean",
3 |     "json": {
4 |         "attr": false
5 |     }
6 | }
```


2.6. @JsonRootName

The `@JsonRootName` annotation is used – if wrapping is enabled – to specify the name of the root wrapper to be used.

Wrapping means that instead of serializing a *User* to something like:

```
1  {
2    "id": 1,
3    "name": "John"
4  }
```

It's going to be wrapped like this:

```
1  {
2    "User": {
3      "id": 1,
4      "name": "John"
5    }
6  }
```

So, let's look at an example – we're going to use the `@JsonRootName` annotation to indicate the name of this potential wrapper entity:

```
1  @JsonRootName(value = "user")
2  public class UserWithRoot {
3    public int id;
4    public String name;
5  }
```

Spring REST client – RestTemplate Consume RESTful Web Service

RestTemplate Methods

RestTemplate provides higher level methods that correspond to each of the six main HTTP methods that make invoking many RESTful services. In the following list has methods provided by Spring RestTemplate for each http methods.

Method	Spring RestTemplate's method
Get	getForObject, getForEntity
Post	postForObject(String url, Object request, Class responseType, String...? uriVariables) postForLocation(String url, Object request, String...? uriVariables),
Put	put(String url, Object request, String...? uriVariables)
Delete	delete()
Head	headForHeaders(String url, String...? uriVariables)
Options	optionsForAllow(String url, String...? uriVariables)

In this example, we are using @GetMapping, @PostMapping, @PutMapping and @DeleteMapping annotations, these annotations are introduced as of Spring 4.3 version in parallel of @RequestMapping annotation with Http Methods as below.

@GetMapping = @RequestMapping + Http GET method

@PostMapping = @RequestMapping + Http POST method

@PutMapping = @RequestMapping + Http PUT method

@DeleteMapping = @RequestMapping + Http DELETE method

Spring RestTemplate get Method :

```
/**
 *
 */
package com.doj.restclient.account;

import org.springframework.web.client.RestTemplate;

/**
 * @author Dinesh.Rajput
 *
 */
public class SpringRestClient {

    private static RestTemplate restTemplate = new RestTemplate();
    private static final String baseUrl = "http://localhost:8080/restapi/";

    /**
     * @param args
     */
    public static void main(String[] args) {
        //Read Account details for a given accountId = 1 using GET method of RestTemplate
        Account accountDetail = restTemplate.getForObject(baseUrl+"account/1", Account.class);
        System.out.println("Account Detail for given accountId : " +accountDetail);

    }

}
```

Spring RestTemplate post Method call:

```
/**
 *
 */
package com.doj.restclient.account;

import org.springframework.web.client.RestTemplate;

/**
 * @author Dinesh.Rajput
 *
 */
public class SpringRestClient {

    private static RestTemplate restTemplate = new RestTemplate();
    private static final String baseUrl = "http://localhost:8080/restapi/";

    /**
     * @param args
     */
    public static void main(String[] args) {
        //Create Account using POST method of RestTemplate
        Account account = new Account("Arnav Rajput", "Noida", 312.33);
        account = restTemplate.postForObject(baseUrl+"account", account, Account.class);
        System.out.println("Added account: " +account);

    }

}
```

HttpMessageConverter in Spring.

HttpMessageConverter is a strategy interface that specifies a converter that can convert from and to HTTP requests and responses in Spring REST Restful web services. Internally Spring MVC uses it to convert the Http request to an object representation and back. Let's see following code for REST endpoint service.

@ResponseBody

In Spring MVC, @ResponseBody annotation on a AccountController method indicates to Spring that the return Account object of the method is serialized to JSON directly to the body of the HTTP Response. It uses "Accept" header to choose the appropriate Http Converter to marshall the object.

@RequestBody

In Spring MVC, @RequestBody annotation on the argument of a AccountController method create (@RequestBody Account account) – it indicates to Spring that the body of the HTTP Request (either in JSON or XML) is deserialized to that Account Java Object. It uses "Content-Type" header specified by the REST Client will be used to determine the appropriate converter for this.

Default Message Converters in Spring MVC

In Spring MVC Framework, there is a set of default converters automatically registered which supports a whole range of different resource representation formats – json, xml for object. In RestTemplate in Spring REST, objects passed to and returned from the methods of RestTemplate class like `getForObject()`, `postForObject()`, and `put()` are converted to HTTP requests and from HTTP responses by registered `HttpMessageConverters`.

- **StringHttpMessageConverter:** it converts Strings from the HTTP request and response.
- **FormHttpMessageConverter:** it converts form data to/from a `MultiValueMap<String, String>`.
- **ByteArrayHttpMessageConverter:** it converts byte arrays from the HTTP request and response.
- **MappingJackson2HttpMessageConverter:** it converts JSON from the HTTP request and response.
- **Jaxb2RootElementHttpMessageConverter:** it converts Java objects to/from XML.
- **SourceHttpMessageConverter:** it converts `javax.xml.transform.Source` from the HTTP request and response.
- **AtomFeedHttpMessageConverter:** it converts Atom feeds.
- **RssChannelHttpMessageConverter:** it converts RSS feeds.

Customizing HttpMessageConverters with Spring MVC

Let's see Spring MVC configuration file.

```
@Configuration
@EnableWebMvc
@ComponentScan("com.doj.restapi.web.controller")
public class WebConfiguration{

}
```

In the above MVC configuration file, we are using `@EnableWebMvc` annotation, it automatically registered default Http message converters with application as listed above according to available library in the class path. Now, if there is a need to customize the default message converters in some way. Spring provides a `WebMvcConfigurerAdapter` class, which allow us to change the default list of Http Converters with our own. Now let's below changed configuration class for Spring MVC.

```

/
package com.doj.restapi.web.config;

import java.util.List;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.converter.HttpMessageConverter;
import org.springframework.http.converter.json.MappingJackson2HttpMessageConverter;
import org.springframework.http.converter.xml.MarshallingHttpMessageConverter;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

/**
 * @author Dinesh.Rajput
 *
 */
@Configuration
@EnableWebMvc
@ComponentScan("com.doj.restapi.web.controller")
public class WebConfiguration extends WebMvcConfigurerAdapter{
    @Override
    public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
        converters.add(createXmlHttpMessageConverter());
        converters.add(new MappingJackson2HttpMessageConverter());
        super.configureMessageConverters(converters);
    }
    private HttpMessageConverter<Object> createXmlHttpMessageConverter() {
        MarshallingHttpMessageConverter xmlConverter = new MarshallingHttpMessageConverter();
        XStreamMarshaller xstreamMarshaller = new XStreamMarshaller();
        xmlConverter.setMarshaller(xstreamMarshaller);
        xmlConverter.setUnmarshaller(xstreamMarshaller);
        return xmlConverter;
    }
}

```

@ResponseStatus annotation

Using HTTP Status Codes

Normally any unhandled exception thrown when processing a web-request causes the server to return an HTTP 500 response. However, any exception that you write yourself can be annotated with the `@ResponseStatus` annotation (which supports all the HTTP status codes defined by the HTTP specification). When an *annotated* exception is thrown from a controller method, and not handled elsewhere, it will automatically cause the appropriate HTTP response to be returned with the specified status-code.

For example, here is an exception for a missing order.

```
@ResponseStatus(value=HttpStatus.NOT_FOUND, reason="No such Order") // 404
public class OrderNotFoundException extends RuntimeException {
    // ...
}
```

And here is a controller method using it:

```
@RequestMapping(value="/orders/{id}", method=GET)
public String showOrder(@PathVariable("id") long id, Model model) {
    Order order = orderRepository.findOrderById(id);

    if (order == null) throw new OrderNotFoundException(id);

    model.addAttribute(order);
    return "orderDetail";
}
```

Using @ExceptionHandler


```

@Controller
public class ExceptionHandlingController {

    // @RequestHandler methods
    ...

    // Exception handling methods

    // Convert a predefined exception to an HTTP Status code
    @ResponseStatus(value=HttpStatus.CONFLICT,
                    reason="Data integrity violation") // 409
    @ExceptionHandler(DataIntegrityViolationException.class)
    public void conflict() {
        // Nothing to do
    }

    // Specify name of a specific view that will be used to display the error:
    @ExceptionHandler({SQLException.class,DataAccessException.class})
    public String databaseError() {
        // Nothing to do. Returns the logical view name of an error page, passed
        // to the view-resolver(s) in usual way.
        // Note that the exception is NOT available to this view (it is not added
        // to the model) but see "Extending ExceptionHandlerExceptionHandlerResolver"
        // below.
        return "databaseError";
    }
}

```