Web Services (Spring cloud)

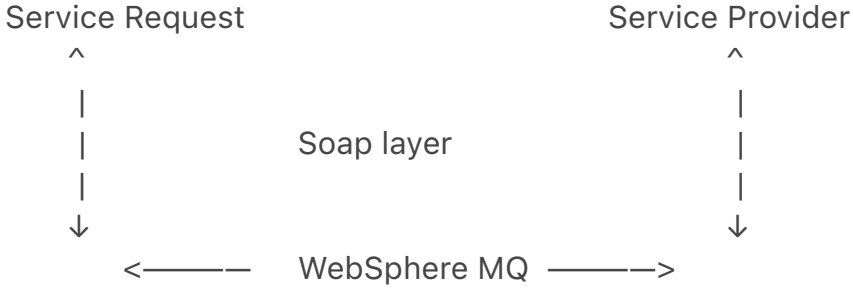
Software system designed to support interoperable (not platform dependent) machine-to-machine interaction over a network.
We can make this by making the format of data transfer common for all like json, xml, etc.

Each web service provides a service definition. It specifies

1. Request/Response format (xml, json, etc.)
2. Request Structure (how can consumer create a request structure)
3. Response Structure (the response produced by service)
4. Endpoint (the point where service is listened/ at what url the service is exposed)

Some key points are:

1. Request and response
2. Message exchange format (Json, xml, etc.)
3. Service Provider or Server
4. Service Consumer or Client
5. Service Definition (Service Definition is the contract between service provider and service consumer)
6. Transport (defines how the service is called or exposed over the internet e.g. HTTP, MQ(communication over the queue))



As we can see request travels from mq and then gets to service provider and similarly response goes in opposite direction.

Types of Web Services (Web Services Group)

SOAP and REST are not really comparable. Rest is an architectural approach whereas soap makes restrictions on the xml (message exchange format).
Simple Object Access Protocol used to be the full form of SOAP but now it is not an abbreviation but a specific way of building of web services.
It uses xml as request message exchange format.
It defines a layout to have soap envelope which contains header and body

|Soap-Env: Envelope|
|————————————
|Soap-Env: Header
|Soap-Env: Body

Soap:
Format
- Soap xml Request
- Soap xml Response

Transport
- SOAP over MQ
- SOAP over HTTP

Service Definition
- WSDL (Web Service Definition Language), it exposes endpoints, all operations, request structure and request response.

REST (representable state transfer)
It was developed by roy fielding (who also developed http). It makes best use of http. Http defines the header as well as the body.
It also defines request methods and response status.

Restful services try to define the services using the different concept that are already present in http.
The most abstraction in rest is resource. A resource has an uri and resource can have different representations (like xml, http, json).
Think from the angle of resource.
So there is no restriction on data exchange format but transport is only with http.
No standard definition (can use wadl (web application definition language), swagger, etc.)

https://github.com/in28minutes/spring-microservices-v2/tree/main/02.restful-web-services

For help
https://github.com/in28minutes/spring-microservices-v2/blob/main/02.restful-web-services/Step05.md
https://github.com/in28minutes/spring-microservices-v2/blob/main/02.restful-web-services/01-step-by-step-changes/v2.md#step-03

Spring RestFul Services:

logging.level.org.springframework=debug to understand the layout of things happening in background.

All the request is first trapped by DispatcherServlet:

Mapping servlets: dispatcherServlet urls [/]
DispatcherServletAutoConfiguration configures the dispatcher servlet

Object get converted to JSON by @ResponseBody + JacksonHttpMessageConverters.
Auto configured by JacksonHttpMessageConvertersConfiguration

Error page (white label page) is also mapped by or auto configured by ErrorMvcAutoConfiguration.

We can also download Talend api tester as a chrome extension which works as rest client.

return ResponseEntity.created(null).build();

Created method takes the location where we can find the description of the created entity. (e.g. /users/{id})

We can send using URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(savedUser.getId()).toUri();
Let say the post request was made at /users we can get the path using fromcurrentRequest() and add id in that.
And pass this location in created method.

Throw our own exception:

Actually when we throw our exception then we can label the class (custom exception class) with @ResponseStatus(code = HttpStatus.NOT_FOUND)
Here we are just throwing the exception in the controller (for that case we can have this thing).
Actually when the white label page comes we can see the whole exception and status. Because of devtools we can see a whole lot of stack trace.
When application is run as a jar file then devtools is disabled by default, so for production code devtools does not run.

But when we perform the request from a request client we get a proper json (not html like when we hit the url at browser) having timestamp, error, status, trace, message, and path.

Here we are using pre defined structure of the exception thrown response.
For custom way of handling error is to extend ResponseEntityExceptionHandler and then give our own exception handling methods.

And mention the type of exception it is handling using @ExceptionHandler over the handleException method.

This is possible if we annotate the class with @ControllerAdvice.

Annotate the class with @ControllerAdvice so that it is applicable for all controller.

Example:

```
package com.in28minutes.rest.webservices.restfulwebservices.exception;

import java.time.LocalDateTime;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;
import org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;

import com.in28minutes.rest.webservices.restfulwebservices.user.UserNotFoundException;


@ControllerAdvice
public class CustomizedResponseEntityExceptionHandler extends ResponseEntityExceptionHandler{

@ExceptionHandler(Exception.class)
public final ResponseEntity<ErrorDetails> handleAllExceptions(Exception ex, WebRequest request) throws Exception {
ErrorDetails errorDetails = new ErrorDetails(LocalDateTime.now(),
ex.getMessage(), request.getDescription(false));

return new ResponseEntity<ErrorDetails>(errorDetails, HttpStatus.INTERNAL_SERVER_ERROR);

}
```

```
@ExceptionHandler(UserNotFoundException.class)
public final ResponseEntity<ErrorDetails> handleUserNotFoundException(Exception ex, WebRequest request) throws Exception {
ErrorDetails errorDetails = new ErrorDetails(LocalDateTime.now(),
ex.getMessage(), request.getDescription(false));

return new ResponseEntity<ErrorDetails>(errorDetails, HttpStatus.NOT_FOUND);

}
}
```

Validation:

Spring-starter-validation performs validation on entity. Like in post method of controller we can add @Valid annotation on the entity so that any validation annotation (like @Size) are all checked for that entity before the mapping of values from request takes place.

To have the localDateTime variable take only values that are in past we can make it through @Past annotation.

To again give simplified own custom exception we have to override the method that handles MethodArgumentNotValidException which is present in ResponseEntityExceptionHandler.
The method is:

```
@Nullable
protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatusCode status, WebRequest request) {
   return this.handleExceptionInternal(ex, (Object)null, headers, status, request);
}
```

In the constraint like @Size() we can also pass the message that goes to consumer when the exception is thrown.

Exception trapped in the method we can give all the information clubbed up by going through ex.getFieldErrors().

To get the first message we can use ex.getFieldError().getDefaultMessage()

We can also get total errors like ex.getTotalErrors().

Advanced RestAPI Features:

1. Documentation
2. Content Negotiation (when consumer wants different format of data)
3. Internationalisation - i18n (customise the response as per the language of the user)
4. Versioning (create multiple versions of the rest api)
5. HATEOS
6. Static Filtering
7. Dynamic Filtering
8. Monitoring

Documentation

Rest api consumers need to understand the rest api:

1. Resources
2. Actions
3. Request/Response Structure (Constraints/Validations)

There are 2 ways of doing:

1. Manually Maintain documentation (hard to keep it in sync with code)
2. Generate from code

In 2011 swagger specification and tools was introduced.
In 2016 OpenAPI specification was created based on swagger specification

OpenAPI Specification -> standard, language-agnostic interface (like we see yml file containing all the details) i.e., helps in understanding and discovering rest api.

Swagger UI -> visualise and interact with the rest api.

springdoc-openapi java library helps to automate the generation of API documentation for spring boot projects.
We will use it in the next step!
Have it handy!

REMEMBER : *Please use* ***<version>2.3.0</version>*** *for* springdoc-openapi-starter-webmvc-ui.

- <dependency>
- <groupId>org.springdoc</groupId>
- <artifactId>springdoc-openapi-starter-webmvc-ui</

artifactId>

- <version>2.3.0</version>
- </dependency>

Content Negotiation

Same Resource - Same URI -> However different representations are possible. (Example different content type (like html, xml, json), or different language(English, dutch, etc.))
How can a consumer tell the rest api what they want: Content Negotiation
Example Accept header (MIME types -> application/json, application.xml, etc.)
And Accept-Language Http request header (en -> English, nl -> Dutch, fr -> French).

For xml to be transferred from the controller we have to add com.fasterxml.jackson.dataformat dependency.

Now from rest client we can add Accept: application/xml in header to have xml format of the code.

Exploring Internationalisation - i18n

Adapting rest api for customers around different parts of the world knowing different language. Internalisation has 18 characters.

Accept-Language -> indicates natural language and locale that the customer prefers.

Let say we define the messages somewhere which should be in messages.properties inside resources folder.
goog.morning.message=Good Morning

In order to interact with the message we can have
private MessageSource messageSource; // inject this in controller

It is a strategy for resolving messages with support for parameterisation, and internalisation of messages.

Now to get the text from .properties:
messageSource.getMessage(“good.morning.message”, null, “Default Message, locale);
// second parameter is to replace any variable we have in message
// third is default message
// fourth is locale

To get the locale associated with the current thread otherwise it would return system default locale:
Locale locale = LocaleContextHolder.getLocale();

Now define the messages_nl.properties and replace the same message with value as that written in dutch. When we use rest client to mention the header of Accept-Language as nl we will get the output from that variable.

Versioning REST API

It is helpful to implement a breaking change in api (like changing the way response was given without troubling the customer using the previous version). There are multiple ways to do it:

1. URL (e.g. localhost:8080/v1/person, localhost:8080/v2/person and in different method in controller we can transfer different object which get mapped to different json object) - used by Twitter
2. Request Parameter (like version = v1 or version = v2 e.g. localhost:8080/person?version=1, localhost:8080/person?version=2)
3. Header
4. Media Type

Example of Request Parameter - Amazon:
This method will be called if we have params as version=1
@GetMapping(path="/person", params= "version=1")

Example of (Custom) headers versioning - Microsoft

- SAME-URL headers=[X-API-VERSION=1]
- SAME-URL headers=[X-API-VERSION=2]

This method will be called if we have headers set up client as X-API-VERSION=1
@GetMapping(path="/person", headers= "X-API-VERSION=1")

Example of Media Type versioning (we can use the same header concept of content negotiation or accept header) - Github
SAME-URL produces=application/vnd.company.app-v1+json
SAME-URL produces=application/vnd.company.app-v2+json

@GetMapping(path="/person", produces= "application/vnd.company.app-v1+json")

In the Accept header from client side we have to pass this full name.

Factors to consider:

1. URI pollution (more in case of uri and request parameter and other 2 uses same url)

2. Misuse of Http Headers
3. Caching (typically caching is done based on url but we have to other (headers, produces, etc.) into consideration)
4. Can we execute the request on the browser? (Uri and request parameter versioning both can be handled by browser but not other 2)
5. API documentation (some api documentation tools might not support same url with different headers)

HATEOAS:

Hypermedia as the Engine of Application State
Websites allow you to

1. See data and perform actions

How about enhancing our rest api to tell consumers how to perform subsequent actions - HATEOAS

e.g.

```
{
	"name" : "Adam",
	"birthDate": "2022-08-06",
	"_links" : {
		"all-users" : {
			"href" : "http://localhost:8080/users"
		}
	}
}
```

Implementation Options:

1. Custom Format and Implementation (difficult to maintain)
2. Use standard implementation (HAL (JSON HyperText Application Language): Simple format that gives a consistent and easy to hyperlink between resources in your API) , the above example defines the way Hal uses the format.
    1. Spring HATEOAS -> Generate HAL responses with hyperlinks to resources

To make use of hateoas, we have to wrap the entity we have to transfer from controller into EntityModel.
To attach links into that EntityModel, we will make use of WebMvcLinkBuilder (builder to ease building link instances pointing to Spring MVC controllers)

```
EntityModel<User> entityModel = EntityModel.of(user);

// linking the model with specific url is quite hard coding, that is why we can
```

map it to method having that url I.e the controller method

```
WebMvcLinkBuilder link =
WebMvcLinkBuilder.linkTo(WebMvcLinkBuilder.method(this.getClass()).retrieve
AllUsers());
// here retrieveAllUsers() is a method of the controller inside the same class

entityModel.add(link.withRel("all-users"));

// make sure to change the return type of the method to EntityMode<User>
```

Customising REST API Responses - Filtering and more:

- Serialisation: convert object to stream (example: JSON)

How about customising the response returned by JackSon framework.

Customising field names in response:
Using @JsonProperty
e.g. @JsonProperty("user_name") overrides user_name for that field on which this annotation is present.

Returning selected fields:

- Filtering
- Example: Filter out passwords
- 2 types of filtering
  - Static Filtering: Same filtering for a bean across different REST API (@JsonIgnoreProperties, @JsonIgnore)
  - Dynamic Filtering: Customise filtering for a bean for specific REST API (@JsonFilter with FilterProvider)

Example of static filtering:
Use @JsonIgnore over the field of the entity we are returning as response.
@JsonIgnoreProperties appear on class where we can provide list of field names to be excluded.

Implementation of Dynamic Filtering:
First we have to change the return type from SomeBean User to MapJacsonValue:

```
SomeBean someBean = new SomeBean("value1", "value2", "value3");
MappingJacksonValue mappingJacksonValue = new
MappingJacksonValue(someBean);

// we can make use of SimpleBeanPropertyFilter to define our own filter
SimpleBeanPropertyFilter filter =
SimpleBeanPropertyFilter.filterOutAllExcept("field1","field3");
```

```
//now create a clubbed filter if we had many
// as we have to have one filter we can make use of SimpleFilterProvider
FilterProvider filters = new SimpleFilterProvider().addFilter("SomeBeanFilter",
filter);

mappingJacksonValue.setFilters(filters);

return mappingJacksonValue;

// now over the entity use @JsonFilter("SomeBeanFilter")
```

Get Production Ready with Spring Boot Actuator:

Monitor and Manage your application in your production.
Add spring-boot-starter-actuator in our project

Provides a number of endpoints:
beans - complete list of Spring beans in our app
health - Application health information
metrics - Application metrics (also about the environment in which app is running)
mappings - details around request mappings
env - to get all the environment variables app is running with
and a lot more.

If we go to localhost:8080/actuator we can see list of links to other details.
By default it only exposes /health
To configure all we can add following in application.properties
mangement.endpoints.web.exposure.include=*

Now go to localhost:8080/actuator, where we can see more of urls

Go to localhost:8080/actuator/metrics where we can see different metrics put that on the path
 localhost:8080/actuator/metrics/http.server.requests

Explore REST API using HAL Explorer

HAL (Json HyperText Application Language)

- Simple format that gives a consistent and easy way to hyperlink between different resources in your API.

HAL Explorer

- An API explorer for Restful Hypermedia APIs using HAL
- Enable non-technical teams to play with apis

Spring Boot HAL Explorer

- Auto configures HAL Explorer for Spring Boot Projects
- Spring-data-rest-hal-explorer

We can check this on localhost:8080/explore/index.html

So in edit headers we can put any uri like /actuator it will show all the links and response status and response body.

HAL Explorer parses the response body and it shows all the content divided by data and links.

To send get request on links click on left arrow and it will show the content it will provide.

For both data and link (provided by Hal as we have managed before), we will 3 sections
JsonProperties -> actual content
Links -> added using Hal
Response body -> combination of both
Response status

Sometimes data put using script (data.sql) acts first and then tables in h2 database is created. To defer datasource initialisation add the following property in application.properties

spring.jpa.defer-datasource-initialization=true