1. Web Service -

Services delivered over the web.

Suppose I have developed a TODO application. And my friend wants to use this TODO feature into his another social media application.

Our Application gives out an HTML output. He doesn't wants HTML. HTML is not a format which is designed for application to application interaction.

My TODO application is developed in layer like -

```
Web
 |
Business
 |
Data - DB
```

So I can build a jar out of my Business and Data layer and give it to my friend, but then my friend will have to also install the DB for the jar. Also he need to make sure that all the other dependencies are satisfied.

But what about interoperable? Will .Net application will be able to call the service in the JAR?

If I upgrade my business logic then he won't automatically get it, I will have to generate the new jar and give it to him.

jar will be installed locally.

So if the TODO application gives out the output in a format that is understandable by other application that would be useful, this is where web services concept comes into picture.

Web service - W3C Definition - Software system designed to support interoperable machine-to-machine interaction over a network.

3 Keys -
- Designed for machine-to-machine (or application-to-application) interaction
- Should be interoperable - Not platform dependent (any language application can be able to interact with it)
- Should allow communication over a network

2. How does data exchange between applications takes place?

```
Application A   -> Request ->   Web service
                <- Response <-
```

Input will go in request and out of webservice will be in response.

How can we make web services platform independent? Our webservice should be able to called from java application or .Net application or an other language application.
We need to make Request and Response platform independent. It should be able to be understood by any language. Such data exchange formats are XML and JSON.

How does the Application A know the format of Request and Response? How does it knows where to send it?
It know because of Service Definition. Every webservice offers a Service Definition.
Service Definition specifies - Request/Response format
- Request Structure
- Response Structure
- Endpoint - URL where webservice is exposed, how service consumer will call service provider.

SOAP Service Definition - WSDL.
REST Service Definition - No Standard. WADL/Swagger/...

3. Key Terminologies -

Request and Response -
- Request - input to a webservice
- Response - output from a webservice

Message/Data Exchange format -
- XML or JSON

Service Provider or Server - The one which hosts the webservice

Service Consumer or Client - The one which consumes the webservice

Service Definition - Contract between the Service provider and Service Consumer.

Transport - How service is called.
- HTTP and MQ

HTTP - service is exposed over a web and we have a URL or endpoint using which we will call it.

MQ - WebService exposed over a queue. Service requestor would place a message into the queue, the service provider would be listening on the queue and as soon as there is a request on a queue it would take the request, process the request, create a response and put it back on to the queue. The service requestor would get the response from the queue. The transport which is used is MQ. Communication is happening over a queue.

4. Web services groups/types -
- SOAP-based
- REST-styled

SOAP and REST are not really comparable - REST defines an architechtural approach, where as SOAP poses an restriction on the format of XML which is exchanged between service provider and service consumer.

SOAP - Simple Object Access Protocol - no longer an abbrevation

SOAP uses XML as a message exchange format.

SOAP defines a specific request and response structure. If we are using SOAP then we have to use this structure.
The structure has -

- SOAP-ENV - Envelope
  - SOAP-ENV - Header
  - SOAP-ENV - Body

Header contains meta information like authentication, authorization, etc.
Body has real content of request/response

SOAP header is optional.

SOAP -
- Format -
  - SOAP XML Request
  - SOAP XML Response
- Transport
  - SOAP over MQ
  - SOAP over HTTP
- Service Definition
  - WSDL - Web Service Definition Language

WSDL defines -

Endpoint
All Operations
Request Structure
Response Structure

5. REST

REpresentational State Transfer

Roy Fielding depeloped REST and Http protocol.

Diagrams

Whenever we browse an web, we enter the url in broswer, click links on webpage, etc. Lot of things are happening in the background.
Broswer sends a request to Server and Server sends a Response back.
These request and response are in format defined by HTTP Protocol (Hyper Text Transfer Protocol)
When we type a url and send a request it sends a GET HTTP Request and the Server responses back with HTTP Response which contains an HTML.
The browser looks into the response, take the HTML and renders it on the screen.
If we have a form and we are submitting it, it is a POST request.
HTTP defines the headers of the request/response and the body of request/response
In addition to request headers and request body, HTTP also defines HTTP Methods
HTTP Methods - GET, POST, PUT, DELETE, etc. - indicates what action we are trying to do.
And a HTTP Response on other hand will also inculde HTTP Response Status Codes

Roy Fielding suggested why don't we use HTTP for webservices. RESTful webservices tries to define a webservice using the concepts that are already there in HTTP.

Key Abstraction - Resource -

A resource has an URI(Uniform Resource Identifier)

A resource can be anything that we want to expose to the outside world through our application.

A resource can have different representations -

XML
JSON
HTML

A resource has an URI (Uniform Resource Identifier)
/users/Ranga/todos/1
/users/Ranga/todos
/users/Ranga

We can perform operations on these resources.

Example:

Create a User - POST /users
Delete a User - DELETE /users/1
Get all Users - GET /users
Get one Users - GET /users/1

REST:

Data Exchange Format
No Restriction. JSON is popular, can use HTML as well.
Transport
Only HTTP //this may be reason we are using SOAP in MBA.
Service Definition
No Standard. WADL/Swagger/...

6. REST VS SOAP

- Restrictions Vs Architectural Approach
- Data Exchange Format
- Service Definition
- Transport
- Ease of implementation - REST is easy to implement

In SOAP there are lot of complexity associated with the parsing of XML.
As SOAP supports on XML as message exchange format, it uses more bandwidth over the web.

7. #RESTful webservices -

Social Media Application -

User -> Posts   1 to many relationship

```
- Retrieve all users                         -> GET      /users
- Create a User                              -> POST     /users
- Retrieve one User                          -> GET      /users/{id}     -> /users/1
- Delete a User                              -> DELETE   /users/{id}     -> /users/1

- Retrieve all posts for a User              -> GET      /users/{id}/posts
- Create a posts for a User                  -> POST     /users/{id}/posts
- Retrieve details of a specific post        -> GET      /users/{id}/posts/{post_id}
```

8. @RestController - tells spring mvc this controller can handle REST requests.

```
@RequestMapping(method = RequestMethod.GET, path = "/hello-world")

@GetMapping(path = "/hello-world")
```

9. Exception - No converter found for return value type: class com.in28min.weservicees.restfulweb.HelloWorldBean

Above exception occured after hitting the rest uri in broswer.

This exception occurs because there is no getter in the bean HelloWorldBean and the automatic conversion of this bean to json won't be possible.

After adding the getting you can see the json response in broswer.

To see Formatted JSON - Install JSON Viewer Chrome Plugin.

9. What is dispatcher servlet?
Who is configuring dispatcher servlet?
What does dispatcher servlet do?
How does the HelloWorldBean object get converted to JSON?
Who is configuring the error mapping?

logging.level.org.springframework = debug => this in application.properties will set a logging level to debug only for springframework

After setting to debug we can find Auto Configuration Report in logs with lot more details.

In logs somewhere it says - DispatcherServletAutoConfiguration matched. This is because it found clas 'org.springframework.web.servlet.DispatcherServlet' in classpath.
We added in a starter on spring-boot-starter-web and spring-boot-starter-web has dependency on web mvc framework. Therefore we get DispatcherServlet class in our

classpath.
And thus it configure DispatcherServlet.

Another log we can see is - ErrorMvcAutoConfiguration matched.
Same way it found some classes in the classpath and it configures the error page.

Spring boot configures a lot more based on the classes present in our class path and this is auto-configuration.

HttpMessageConvertersAutoConfiguration - these were responsible bean to json coversion and json to bean. Jackson beans are intialized.

So our method handler has returned the Bean, so DispatcherServlet thinks how will I return this bean back as a response?
We have @RestController annotation and in @RestController annotation definition we have @ResponseBody annotation. And when we have a @ResponseBody annotation on a controller then response from that controller is mapped by MessageConverter into some other format, here MessageConverter which is used is Jackson, which will convert the bean to json and json response is send back.

10.

/users/{id} -> id is Path Parameter/Variable

```
//hello-world/path-variable/in28min

@GetMapping(path = "/hello-world/path-variable/{name}")
public HelloWorldBean helloWorldPathVariable(@PathVariable String name){

    return new HelloWorldBean(String.format("Hello World, %s", name));
}
```

11.

JSON response in broswer -

birthdate : 1500370250075   //this is json timestamp format. Since 2.0.0.RC1, this setting is auto enabled.

We can change it in application.properties -
spring.jackson.serialization.write-dates-as-timestamps = false;

After this date will come in proper format - birthdate: "2017-07-19T04:20:36.019+0000"

12.

When a GET request is executed successfully and a Response is send back, then spring mvc sends back a status code of 200 in the Response Header.

However in case of POST request, when it is executed successfully we would want a status code of CREATED.

input - details of user

```
        {
            name: "Adam",
            birthdate: "2017-07-19T04:20:36.019+0000"
        }
```

We can send this JSON as part of body of our POST request.

output - CREATED status and Return the created user URI

```
@PostMapping("/users")
public void createUser(@RequestBody User user){             //we need to map it to User and due to @RequestBody whatever is in request body will be mapped with User properties

    User savedUser = service.save(user);
}
```

To send a POST request we will need REST client - PostMan - app or chrome plugin.

After send a POST request we might get a error for older versios - Internal Server Error - Tyoe definition error... can not construct instance of ... This is because we don't have default constructor in our Bean.
But with the recent Jackson and Spring Boot versions, the default constructor is no longer needed. You will not see this error.

Now we want to return the status as CREATED and return URI of created resource -

We can do so byusing ResponseEntity, it is extension of HttpEntity, we can additionally add a Statis code to it.

To build the URI -

```
ServletUriComponentsBuilder.fromCurrentRequest() -> /users
ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")      -> path() allows to append something, we are appending /{id}

ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(savedUser.getId())    -> this will replace {id} with value of savedUser.getId()

ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(savedUser.getId()).toUri()    -> we will get the calculated URI
```

```
@PostMapping("/users")
public void createUser(@RequestBody User user){

    User savedUser = service.save(user);

    URI location = ServletUriComponentsBuilder
    .fromCurrentRequest()
    .path("/{id}")
    .buildAndExpand(savedUser.getId())
    .toUri();

    ResponseEntity.created(location).build();
}
```

Now if you send a request using postman - and check the response it will show status as 201 - Created And if we go in the response headers we can see a header called Location -> http://localhost:8080/users/4

13.

When user or resource is not found then we should not send Sucess or 200 status code

```
@PostMapping("/users/{id}")
public User retrieveUser(@PathVariable int id){

    User user = service.findOne(id);

    if(user == null)
        throw new UserNotFoundException("id - " + id);          //Custom exception class, it extends RuntimeException and not Exception, because RuntimeException is unchecked and Exception is checked.

    return user;
}

public class userNotFoundException extends RuntimeException{
    public UserNotFoundException(String message){
        super(message);
    }
}
```

After this instead of 200-Sucess status code we will get response as 500 Internal Server Error

But the problem is actually resource is not found, so we can return a status of Not found.

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class userNotFoundException extends RuntimeException{

    public UserNotFoundException(String message){
        super(message);
    }
}
```

Due to @ResponseStatus(HttpStatus.NOT_FOUND) this wherever we are throwing UserNotFoundException, it will always return Status - 404 Not Found

14.

We would want to send a same exception response structure for all our webservices.

In previous video we were getting the 404-Not Found response with a specific structure, which was defined by Spring MVC. But in an organisation we would want to define a standard structure.

Customizing the exception handling to define a structure that is defined by us -

```
public class ExceptionResponse {

    private Date timestamp;
    private String message;
    private String details;

    public ExceptionResponse(Date timestamp, String message, String details) {
        super();
        this.timestamp = timestamp;
        this.message = message;
        this.details = details;
    }

    public Date getTimestamp() {
        return timestamp;
    }

    public String getMessage() {
        return message;
    }

    public String getDetails() {
        return details;
    }

}
```

So we want our exception response to be in above ExceptionResponse format, we can do so by using ResponseEntityExceptionHandler class. It is an abstract class which can be extended to provide centralized exception handling across all the exception handlers.

```
@ControllerAdvice            //This should be application across all Controller/Resource
@RestController              //because this is providing a response back in case of
```

```
an exception
public class CustomizedResponseEntityExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(Exception.class)
    public final ResponseEntity<Object> handleAllExceptions(Exception ex, WebRequest request) {

        ExceptionResponse exceptionResponse = new ExceptionResponse(new Date(), ex.getMessage(), request.getDescription(false));        //ExceptionResponse is our bean

        return new ResponseEntity(exceptionResponse, HttpStatus.INTERNAL_SERVER_ERROR);
    }

    //For UserNotFoundException
    @ExceptionHandler(UserNotFoundException.class)
    public final ResponseEntity<Object> handleUserNotFoundException(UserNotFoundException ex, WebRequest request) {

        ExceptionResponse exceptionResponse = new ExceptionResponse(new Date(), ex.getMessage(), request.getDescription(false));

        return new ResponseEntity(exceptionResponse, HttpStatus.NOT_FOUND);
    }

    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatus status, WebRequest request) {

        ExceptionResponse exceptionResponse = new ExceptionResponse(new Date(), "Validation Failed", ex.getBindingResult().toString());

        return new ResponseEntity(exceptionResponse, HttpStatus.BAD_REQUEST);
    }
}
```

When we have multiple Controller classes and we want to share things amongst them then we can use @ControllerAdvice annotation.

15.

```
public User deleteById(int id) {
    for (User user : users) {                         //we cannot use for loop because we cannot delete from a list while iterating... check... thus we need to use a Iterator..
        if (user.getId() == id) {

            ...delete...
        }
    }
    return null;
}

public User deleteById(int id) {

    Iterator<User> iterator = users.iterator();

    while (iterator.hasNext()) {
        User user = iterator.next();
        if (user.getId() == id) {
            iterator.remove();
            return user;
        }
    }
```

```
        return null;
    }

    ....

    @DeleteMapping("/users/{id}")
    public void deleteUser(@PathVariable int id) {
        User user = service.deleteById(id);

        if(user==null)
            throw new UserNotFoundException("id-"+ id);
    }
```

When user is deleted successfully it would return status of 200

16. Implementing validations for RESTful webservices.

We will use java validation API to add validations on our beans.

When we get a request to create a user, we want to validate the content

```
    @PostMapping("/users")
    public ResponseEntity<Object> createUser(@Valid @RequestBody User user) { @Valid will enable validation on User
        ....
    }


    @ApiModel(description="All details about the user.")
    @Entity
    public class User {

        @Id
        @GeneratedValue
        private Integer id;

        @Size(min=2, message="Name should have atleast 2 characters")
        @ApiModelProperty(notes="Name should have atleast 2 characters")
        private String name;

        @Past                                                           //this will check id birthdate is the date in the past
        @ApiModelProperty(notes="Birth date should be in the past")
        private Date birthDate;

        @OneToMany(mappedBy="user")
        private List<Post> posts;

        protected User() {

        }

        public User(Integer id, String name, Date birthDate) {
            super();
            this.id = id;
            this.name = name;
            this.birthDate = birthDate;
        }


        //getters and setters
        //toString()
    }
```

Now we have added the validations and upon sending a invalid request or name with just one character, we will get response as 400 Bad Request.

But we want to give more specific response back to user.
We can do so by overriding the handleMethodArgumentNotValid() in our ResponseEntityExceptionHandler class - CustomizedResponseEntityExceptionHandler. See above.
handleMethodArgumentNotValid() method will be executed when binding to a specific method arguments fails.

```
@Override
protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatus status, WebRequest request) {

    ExceptionResponse exceptionResponse = new ExceptionResponse(new Date(), "Validation Failed", ex.getBindingResult().toString());

    return new ResponseEntity(exceptionResponse, HttpStatus.BAD_REQUEST);
}
```

ex.getBindingResult().toString() -> has a detailed description of what went wrong.

Now the response body will content detailed description.

validation-api-1.1.0.Final.jar - java validation API

The most popular implementation of validation-api is hibernate-validator

We get validation-api and hibernate-validator jars because they are defined as dependencies in spring-boot-starter-web

17.
Quick Tip : HATEOAS Recent Changes
VERSION UPDATES FOR NEXT LECTURE

There are a few modifications of HATEOAS in the latest release of Spring HATEOAS 1.0.0:

One of these should work

Option 1 : Spring Boot Release >= 2.2.0

```
import org.springframework.hateoas.EntityModel;
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.*;

EntityModel<User> model = new EntityModel<>(user);
WebMvcLinkBuilder linkTo = linkTo(methodOn(this.getClass()).retrieveAllUsers());
model.add(linkTo.withRel("all-users"));
```

Option 2: Older versions

```
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.linkTo;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.methodOn;
import org.springframework.hateoas.Resource;
import org.springframework.hateoas.mvc.ControllerLinkBuilder;

Resource<User> resource = new Resource<User>(user);
ControllerLinkBuilder linkTo = linkTo(methodOn(this.getClass()).retrieveAllUsers());
resource.add(linkTo.withRel("all-users"));
return resource;
```

18. Implementing HATEOAS for RESTful Services

If you hib git hub repositories URI, we get data not only of repositories but also other links which we can use to perform other related tasks, like we can star the

repository, we can fork it, we can see followers, etc.
Same way when we see a fb post, it not just sends a data but also links to add likes, share, comment, etc.

Same concept, in a web app when we return a resource also send other resources links which can be useful. This concept is called HATEOAS - Hypermedia as The Engine of Application State.

When we retrieve a user, we also want to tell how we can retrieve all users. Send link to retrieve all users.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>

@GetMapping("/users/{id}")
public Resource<User> retrieveUser(@PathVariable int id) {              //we are returning a Resource now.
    User user = service.findOne(id);

    if(user==null)
        throw new UserNotFoundException("id-"+ id);

    Resource<User> resource = new Resource<User>(user);

    ControllerLinkBuilder linkTo = linkTo(methodOn(this.getClass()).retrieveAllUsers());            //URI for retrieveAllUsers() handler will be send. ControllerLinkBuilder will help us in creating links from method handler.

    resource.add(linkTo.withRel("all-users"));

    return resource;
}
```

We have a static import to
```
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.linkTo;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.methodOn;
```

Check above how this can be done in new versions.

19. Internationalization for RESTful Services - i18n

```
@GetMapping(path = "/hello-world-internationalized")
public String helloWorldInternationalized() {

    return "good morning";                  //we want to return this message in different language depending upon from where the request is comming.
}
```

To achieve internationalization of our services we need to configure few things -
- LocaleResolver
    - Default Locale - Locale.US

- ResourceBundleMessageSource   - list of properties, spring concept for handling our properties.

Usage
- Autowire MessageSource
- @RequestHeader(value = "Accept-Language", required = false) Locale locale
- messageSource.getMessage("helloWorld.message", null, locale)

First we need to add a Bean in Spring-Boot application -

Simple Session Locale Resolver -

```
@SpringBootApplication
public class RestfulWebServicesApplication {

    public static void main(String[] args) {
        SpringApplication.run(RestfulWebServicesApplication.class, args);
    }

    @Bean
    public LocaleResolver localeResolver() {
        SessionLocaleResolver localeResolver = new SessionLocaleResolver();
        localeResolver.setDefaultLocale(Locale.US);
        return localeResolver;
    }
}
```

Store our properties in resources folder.

messages.properties - default properties:

```
good.morning.message=Good Morning
```

messages_fr.properties - french:

```
good.morning.message=Bonjour
```

Now we would need something to read this properties and customize them based on the input accept header - we do that by defining another bean in our application called ResourceBundleMessageSource

```
@SpringBootApplication
    public class RestfulWebServicesApplication {

        public static void main(String[] args) {
            SpringApplication.run(RestfulWebServicesApplication.class, args);
        }

        @Bean
        public LocaleResolver localeResolver() {
            SessionLocaleResolver localeResolver = new SessionLocaleResolver();
            localeResolver.setDefaultLocale(Locale.US);
            return localeResolver;
        }

        @Bean public ResourceBundleMessageSource messageSource(){       //name of method should be messageSource
            ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
            messageSource.setBaseName("messages");      //our properties file name starts with messages*
            return messageSource;
        }
    }
```

Now we need to update our controller to use messageSource:

```
@RestController
public class HelloWorldController {

    @Autowired
    private MessageSource messageSource;

    @GetMapping(path = "/hello-world-internationalized")
    public String helloWorldInternationalized(@RequestHeader(name = "Accept-Language", required=false) Locale locale) {          //So the local
```

```
        will come in header Accept-Language
            return messageSource.getMessage("good.morning.message", null, locale);      //We have added in good.morning.message in properties file.
        }

    }
```

now from postman we can send value for Accept-Language as fr we will get Bonjour.

20. Few things which we can do to simplify the internationalization we did above -

In above example we passed local as a parameter to our handler method, so this is a pain of we need to do this for every method handler. Spring provides an alternate -

```
    @GetMapping(path = "/hello-world-internationalized")
    public String helloWorldInternationalized() {
        return messageSource.getMessage("good.morning.message", null, LocaleContextHolder.getLocale());          //LocaleContextHolder will get the locale
    }
```

But this alone will not work and we will get default locale even for fr. We need to change SessionLocaleResolver to AcceptHeaderLocaleResolver:

```
    @SpringBootApplication
    public class RestfulWebServicesApplication {

        public static void main(String[] args) {
            SpringApplication.run(RestfulWebServicesApplication.class, args);
        }

        @Bean
        public LocaleResolver localeResolver() {
            AcceptHeaderLocaleResolver localeResolver = new AcceptHeaderLocaleResolver();       //because of this we will not have to add locale as RequestHeader parameter to every handler method.
            localeResolver.setDefaultLocale(Locale.US);
            return localeResolver;
        }
    }
```

Now it will work fine.

And the bean for ResourceBundleMessageSource can be removed from Application class and it can be directly added into application.properties file:

```
    spring.messages.basename=messages;
```

21. Content Negotiation - Implementing Support for XML

Resources can have multiple representations. Till now we have used JSON, so how do we use XMl.

From postman if we send Header Accept with value as application/json we will get the json response, but if we try to send its value as application/xml then we won't get the response. We get status of 406-Not Acceptable.

To resolve this we just need to make one jar available in pom.xml -

```
    <dependency>
        <groupId>com.fasterxml.jackson.dataformat</groupId>
        <artifactId>jackson-dataformat-xml</artifactId>
    </dependency>
```

After adding dependency we can get xml response back.

All the object to json and json to object is done by jackson.

22. Configuring Auto Generation of Swagger Documentation

For swagger check Spring-Spring-Boot-Interview-Guide-Udemy.txt

23.

Incompatibility in recent versions of Swagger and Hateoas

There is an incompatibility with the latest releases of Spring Boot between Swagger and HATEOAS.

While we wait for a fix, here is the set of latest dependencies working well.

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.1.3.RELEASE</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
```

You might need to update your HATEOAS Code to be compatible with 2.1.3.Release

```
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.linkTo;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.methodOn;
import org.springframework.hateoas.Resource;
import org.springframework.hateoas.mvc.ControllerLinkBuilder;

Resource<User> resource = new Resource<User>(user);
ControllerLinkBuilder linkTo = linkTo(methodOn(this.getClass()).retrieveAllUsers());
resource.add(linkTo.withRel("all-users"));
return resource;
```

24.

Check details of Acctuator at Spring-Spring-Boot-Interview-Guide-Udemy.txt

25. Implementing Static Filtering for RESTful Service

What is filtering - Suppose we are hiiting a uri - /user and we are getting response with three properties id, name, birthdate. But what if I don't want birthdate in response. This concept is called filtering.

From the attributes in our bean we want to filter out certain things.

Suppose there was password field, we don't want to share it with anyone.

```
public class SomeBean {

    private String field1;

    private String field2;

    private String field3;

    ...

    //getter, setters
}

RestController
public class FilteringController {

    @GetMapping("/filtering")
    public SomeBean retrieveSomeBean() {

        SomeBean someBean = new SomeBean("value1", "value2", "value3");
        return someBean;
    }
}
```

This will return response will all fields.

Suppose we want to ignore field3 in response. We can do -

```
public class SomeBean {

    private String field1;

    private String field2;

    @JsonIgnore
    private String field3;

    ...
}
```

Now field3 will not be there in the response.

```
@GetMapping("/filtering")
public List<SomeBean> retrieveSomeBeanLIST() {

    return Arrays.asList(new SomeBean("value1", "value2", "value3"),
                         new SomeBean("value12", "value22", "value32"));
}
```

Even if we send List back we field3 won't be there in response

This is one approach, another appraoch is to use @JsonIgnoreProperties -

```
@JsonIgnoreProperties(value={"field1", "field2"})
public class SomeBean {

    private String field1;

    private String field2;

    private String field3;

    ...
}
```

Now only field3 will be there in response.

This what we called static filtering. If we want to ignore field1 in one scenario and field2 in another scenario, we cannot do it using this way.

26. Implementing Dynamic Filtering for RESTful Service

For some requests I want field1 and field2 for some I want field3.

With dynamic filtering we cannot directly configure filtering on bean, we need to start configuring at controller where we are retrieving it.

```
@RestController
public class FilteringController {

    // field1,field2
    @GetMapping("/filtering")
    public MappingJacksonValue retrieveSomeBean() {
        SomeBean someBean = new SomeBean("value1", "value2", "value3");

        SimpleBeanPropertyFilter filter = SimpleBeanPropertyFilter.filterOutAllExcept("field1", "field2");

        FilterProvider filters = new SimpleFilterProvider().addFilter("SomeBeanFilter", filter);

        MappingJacksonValue mapping = new MappingJacksonValue(someBean);

        mapping.setFilters(filters);

        return mapping;
    }

    // field2, field3
    @GetMapping("/filtering-list")
    public MappingJacksonValue retrieveListOfSomeBeans() {
        List<SomeBean> list = Arrays.asList(new SomeBean("value1", "value2", "value3"),
                new SomeBean("value12", "value22", "value32"));

        SimpleBeanPropertyFilter filter = SimpleBeanPropertyFilter.filterOutAllExcept("field2", "field3");

        FilterProvider filters = new SimpleFilterProvider().addFilter("SomeBeanFilter", filter);

        MappingJacksonValue mapping = new MappingJacksonValue(list);                                //LIST

        mapping.setFilters(filters);

        return mapping;
    }

}
```

FilterProvider has only one implementation i.e. SimpleFilterProvider.

List of valid filters needs to be defined on the bean, if we don't then the filters won't work -

```
@JsonFilter("SomeBeanFilter")            //SomeBeanFilter is the name we gave to filters in our handler method.
public class SomeBean {

    private String field1;

    private String field2;

    private String field3;

    ....

    //getter and setters.
}
```

Now filtering will work as we wanted.

We can avoid the duplication of code in both methods.

27. Versioning RESTful Services - Basic Approach with URIs

Check details of Versioning at Spring-Spring-Boot-Interview-Guide-Udemy.txt

28. Implementing Basic Authentication with Spring Security

There are many ways for Authentication, one of the basic way is Basic Authentication, it is done by send username and password as part of your request. Only after providing the correct username and password you will be allowed to access the resource.

There are other advance form of authentication like digest authentication where password digest is created and send, so actual password is not send to server.

Other option is to use Oauth2 authentication.

To implement Basic Authentication we need to add a dependency -

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Due to this spring-boot auto configuration will help us to auto configure basic security for us.

Once the server is restarted after adding the dependency you will see - Using default security password: ...... in the console. So from now on ...... will be the password. Each time server starts up the password would be different.

Now from postman if you try to send a POST/GET request your will get 401 Unauthorized response.
So now go in Authentication tab, select type as Basic Auth and enter username - user - this is default username and the password would be the one in the console.
Now request would be success.
All resources will only work with default username and password.

If we don't want password to be changed everytime the server is started then we can confiure the password in the application.properties - We can also configure username -

```
    security.user.name=username
    security.user.password=password
```

29. Overview of Connecting RESTful Service to JPA

30. Creating User Entity and some test data

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

In application.properties we can enable h2 console and enable logging -

```
    spring.h2.console.enabled = true
    spring.jpa.show-sql = true
```

****** To see dropdown of suggessions in application.properties files install spring-tools-eclipse plugin.

Once we add @Entity annotation and run hibernate will create table automatically.
Now we also want data to be added into that table, so we can do that by adding a sql file in resources/data.sql - add insert statements in this sql file. It will automatically pick this sql file and execute it.
Use a single quote inside sql file.

To go to h2 console - localhost:8080/h2-console

make sure jdbc url has jdbc:h2:mem:testdb

31. Updating GET methods on User Resource to use JPA

```
@Repository
public interface UserRepository extends JpaRepository<User, Integer>{ //<Entity, Primary Key>

}

@GetMapping("/jpa/users/{id}")
public Resource<User> retrieveUser(@PathVariable int id) {
    Optional<User> user = userRepository.findById(id);

    if (!user.isPresent())
        throw new UserNotFoundException("id-" + id);

    // "all-users", SERVER_PATH + "/users"
    // retrieveAllUsers
    Resource<User> resource = new Resource<User>(user.get());

    ControllerLinkBuilder linkTo = linkTo(methodOn(this.getClass()).retrieveAllUsers());

    resource.add(linkTo.withRel("all-users"));

    // HATEOAS

    return resource;
}
```

32. Updating POST and DELETE methods on User Resource to use JPA

33. Creating Post Entity and Many to One Relationship with User Entity

```
@Entity
public class Post {

    @Id
    @GeneratedValue
    private Integer id;
    private String description;

    @ManyToOne(fetch=FetchType.LAZY)
    @JsonIgnore
    private User user;

    ...
    @Override
    public String toString() {
        return String.format("Post [id=%s, description=%s]", id, description);		//don't try to print user here, else post will try to print user and user will try to print post, in loop
    }
}

@ApiModel(description="All details about the user.")
@Entity
public class User {

    @Id
    @GeneratedValue
    private Integer id;

    @Size(min=2, message="Name should have atleast 2 characters")
    @ApiModelProperty(notes="Name should have atleast 2 characters")
    private String name;

    @Past
    @ApiModelProperty(notes="Birth date should be in the past")
    private Date birthDate;

    @OneToMany(mappedBy="user")          //user_id_pk will be foreign key in Post table.
    private List<Post> posts;

    ...
}
```

34. Implementing a GET service to retrieve all Posts of a specific User

```
@Entity
public class Post {

    @Id
    @GeneratedValue
    private Integer id;
    private String description;

    @ManyToOne(fetch=FetchType.LAZY)
    @JsonIgnore
    private User user;

    ...

    @Override
    public String toString() {
        return String.format("Post [id=%s, description=%s]", id, description);
    }
}

@ApiModel(description="All details about the user.")
@Entity
public class User {

    @Id
    @GeneratedValue
    private Integer id;

    @Size(min=2, message="Name should have atleast 2 characters")
    @ApiModelProperty(notes="Name should have atleast 2 characters")
    private String name;

    @Past
    @ApiModelProperty(notes="Birth date should be in the past")
    private Date birthDate;

    @OneToMany(mappedBy="user")
    private List<Post> posts;

    ...
}
```

Why we added @JsonIgnore - When we fetch user we will also get all his posts. Thats okay. But if when get a post then it will also get user, thus user will again get post, again due to this post will get user, and so on. It becomes recurssive. So we don want user if we are getting post.

35. Implementing a POST service to create a Post for a User

```
@Repository
public interface PostRepository extends JpaRepository<Post, Integer>{

}

.............................


@PostMapping("/jpa/users/{id}/posts")
public ResponseEntity<Object> createPost(@PathVariable int id, @RequestBody Post post) {

    Optional<User> userOptional = userRepository.findById(id);

    if(!userOptional.isPresent()) {
        throw new UserNotFoundException("id-" + id);
    }

    User user = userOptional.get();

    post.setUser(user);

    postRepository.save(post);

    URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(post.getId()).toUri();

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

}
```

36. Richardson Maturity Model

Important best practices for RESTful web services -

We are using REST, but how RESTful are you? Richardson Maturity Model helps in evaluating this.
It defines three different levels of RESTful services.

Level 0 - Expose SOAP web services in REST style.
Its exposing URLs like below. They are not talking about resources, they are like actions.

- https://server/getPosts
- https://server/deletePosts
- https://server/doThis

Level 1 - Exposing resources with proper URIs

We have started thinking in terms of resources now, like my resources are users, accounts, todos, etc.

- http://server/accounts
- http://server/accounts/10

Note: Improper use of HTTP methods. We are not making proper use of HTTP methods yet.

Level 2 - Level 1 + HTTP Methods
Adding DELETE HTTP method while deleting a resource, GET while fetching a resource, etc.

Level 3 - Level 2 + HATEOAS

Data + Next Possible actions

37. Best Practices

Check Spring-Spring-Boot-Interview-Guide-Udemy.txt

Section - Microservices with Spring Cloud

38. Introduction - Microservices with Spring Cloud

1. Spring Cloud config server and bus
2. Load balancing with Ribbon and Feign
3. Implement Naming Server with Eureka
4. Implementing API Gateway with Zuul
5. Distributed tracing with Zipkin
6. Fault Tolerance with Hystrix

39.

There are many definitions for microservices.

Small autonomous services that work together
- definition by Sam Newman

In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.
These services are built around business capabilities and independently deployable by fully automated deployment machinery.
There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies
- definition by James Lewis and Martin Fowler

MICROSERVICES
- REST
- Small Well Chosen Deployable Units
- Cloud Enabled

Microservices are services which are exposed by REST, in addition we have small deployable units and they are cloud enabled.

How does it looks? - Check the diagrams in pdf presentation.

Cloud enabled -
each microservice can have one or multiple instances.
if one of the instance goes down then we should be able to bring up new

instance easily

We should be able to bring up new instance or bring down a instance easily without having any huge problems and easy configurations.

So in this section we will see how to make it cloud enabled, how to bring new instance up and old one down.

40. Challenges with Microservices

BOUNDED CONTEXT -

As we said eairlier that instead of one big monolithic applicaiton we will build multiple microservices. So how do you identify the boundary of each microservice? How do we identify what to do with each of these microservices?

CONFIGURATION MANAGEMENT -

We said that we will have multiple microservice and each of these microservices have multiple instances in different enviornment and there are multiple enviornments.

eg - we have 10 microservies with 5 enviornments and 50 instances. So there is tons of configuration. And its lot of work to maintain.

DYNAMIC SCALE UP AND SCALE DOWN -

The loads on different microservices can be different at different instance of time.

At particular time I might need two instances of microservice-2 but later at differnt point of time I may need 10 instances.

So I should be able to bring up the new instances and bring down older one when they are not needed.

All this with dynamic load balancing. Because when there is 4 instances of a service we would like to load to be distributed between them, but when instances becomes 8 then it should again get distributed amongst these 8 instances.

So wee need ability to dynamically bring new instances and also distribute load between these new instances.

VISIBILITY -

Suppose we have 10 microservices and there is a bug, so how will we identify where the bug is? We need to have a centralized logs where we can go and find out what happened for a specific request? A single request can call multiple microservices? Which microservice was a problem.

We also need some monitoring around these microservices. Because as we will have hundreds of services, we should be able to indentify which microservice when down, we would want to automatically indentify server's where there is not enough disk space. All these needs to be automated.

So we need great visibility into what happening into these microservices.

PACK OF CARDS -

If it is not well designed then microservices architechture can be like a pack of cards.

Mean generally one microservice call another and another call another and so on, so there might be a fundamental microservice for all and when that goes down then entire applicaiton might goes down.

And therefore it is very importance to have fault tolerance into our microservices.

How do we prevent one microservice from taking down our entire applicaiton? How do we fault tolerance our applicaiton.

41. Introduction to Spring Cloud.

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state).

Spring cloud is not the only project, there are number of projects under the umbrella of Spring Cloud like -

Spring Cloud Config - provides centralized configuration management
Spring Cloud Netflix - wide variety of components which Netflix has open sourced (Eureka, Hystrix, Zuul, Archaius, etc.)
Spring Cloud Bus
Spring Cloud Cloudfoundary
etc,

We talked about challenges earlier now lets see what are soltions provided by Spring Cloud for those -

Configuration management -

CENTRALIZED CONFIGURATION MANAGEMENT ->

- Spring Cloud Config Server -

Spring Cloud Config Server provides a approach where we can store all the configuration for all the different enviornment of all the microservices in a git repository. So we can store it in just one place in a centralized location. And Spring Cloud Config Server can be used to expose that configuration to all the microservices. This helps us to keep the configuration in one place and that makes it very easy to maintain for all microservices.

DYNAMIC SCALE UP AND SCALE DOWN ->

- Naming Server (Eureka)
- Ribbon (Client Side Load Balancing)
- Feign (Easier REST Clients)

Check diagram of Ribbon Load Balancing.

From the diagram we can see that there is a microservice called CurrencyCalculationService which is talking with CurrencyExchangeService.

As we can see in the diagram that there are multiple instances of CurrencyExchangeService and its possible that at any point of time a new instance is added or removed out.

And we want CurrencyCalculationService to be able to distribute all the load amongst the CurrencyExchangeService instances.

We will want to dynamically check what are the instances available for CurrencyExchangeService and make sure load is distributed amongst all of them.

The solution in this course -

All the instances of all the microservices would register with Naming Server (Eureka)

Naming server has two important features -

- Service Registration
- Service discovery

In our example the CurrencyCalculationService can ask the Eureka Naming Server to give the current instance of the CurrencyExchangeService. And the Naming Service would provide those URLs to CurrencyCalculationService. This helps in establishing dynamic relationship between the CurrencyCalculationService and instances of CurrencyExchangeService.

We would use Ribbon for client side load balancing. That means the CurrencyCalculationService will host Ribbon. It will make sure that the load is evenly distributed amongst the existing instances of the CurrencyExchangeService that it will get from the Naming Server.

We will also use Feign in CurrencyCalculationService as a mechanism to write sime RESTful clients.

VISIBILITY AND MONITORING ->

- Zipkin Distributed Tracing
- Netflix API Gateway

We would use Spring Cloud Sleuth, to assign a id to a Request across multiple components. And we would use Zipkin Distributed Tracing to trace a request across multiple components.

One of the important things about microservices is that these microservices have lot of common features, like, logging, security, analytics, etc. We don't want to implement these common features in every microservice. API Gateway provides great solutions to these kind of challenges. We will use Netflix Zuul API Gateway in this course.

FAULT TOLERANCE ->

- Hystrix

If a service is down Hystrix helps us to configure the default response.

42. Advantages of Microservices -

- It enables us to adapt new technology and processes very easily.

When we build an applicaiton as a combination of microservices which can communicate with each other using simple messages, each of these microservice can be build in different technologies.

In typical monolithic applicaiton we would not have that flexibility.

And also the new microservice which we create, we can bring in new process.

- Dynamic Scaling

Consider a online shopping applicaiton like Amazon, they don't usually have sam amount of traffic or load throughtout the year. During holidays the load can be huge. If our applicaiton is cloud enabled, then they can scale dynamically and you can procure hardware and release it dynamically as well.

So we can scale up our applicaiton and scale it down depending upon the load.

- Faster Release Cycles

Beacuse we are developing smaller components its much easier to release the microservices compared to monolithic applicaitons. This means we can bring new features faster to market.

43. Microservice Components - Standardizing Ports and URL

We would be developing lot of components. We would be installing atleast 7 different projects. And therefore its very important to standardize the ports on which we would run these applicaiton.

Ports:

Limits Service	8080, 8081, ...
Spring Cloud Config Server	8888
Currency Exchange Service	8000, 8001, 8002, ..
Currency Conversion Service	8100, 8101, 8102, ...
Netflix Eureka Naming Server	8761
Netflix Zuul API Gateway Server	8765
Zipkin Distributed Tracing Server	9411

URLs

Application	URL
Limits Service http://localhost:8080/limits POST -> http://localhost:8080/actuator/refresh	
Spring Cloud Config Server http://localhost:8888/limits-service/default http://localhost:8888/limits-service/dev	
Currency Converter Service - Direct Call http://localhost:8100/currency-converter/from/USD/to/INR/quantity/10	

Currency Converter Service - Feign
http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/10000

Currency Exchange Service
http://localhost:8000/currency-exchange/from/EUR/to/INR
http://localhost:8001/currency-exchange/from/USD/to/INR

Eureka
http://localhost:8761/

Zuul - Currency Exchange & Exchange Services
http://localhost:8765/currency-exchange-service/currency-exchange/from/EUR/to/INR

http://localhost:8765/currency-conversion-service/currency-converter-feign/from/USD/to/INR/quantity/10

Zipkin
http://localhost:9411/zipkin/

Spring Cloud Bus Refresh
http://localhost:8080/bus/refresh

44. Part 1 - Intro to Limits Microservice and Spring Cloud Config Server

Centralized Microservice Configuration.

Check diagram Microservices Environment.

We will have three microservices and each one will have its own configuration, it can db config, etc.

There are multiple environments for each of these microservice - DEV, QA, STAGE, PROD, etc.

Example Currency Conversion Service can have 1 DEV environment, 2 QA, and 1 PROD. check diagrams.

And some of these environments may have multiple instances of the same microservice.

So we are talking about configuring a lot of instances of different microservices. Managing the configuration for each microservice for each environment is very difficult thing. That where we will have a centralized microservice configuration.

Spring Cloud Config Server says, you put all the configurations for your application in a git repository and I will take care of managing the configuration and providing it to specific microservice.
If LimitService says I would like a configuration for LimitService for DEV environment, Spring Cloud Config Server will be able to provide to it.
If CurrencyCalculationService says that I would like to have a configuration for 3rd instance of PROD environment Spring Cloud Config Server will be able to provide it.

Spring Cloud Config Server will act as centralized microservice configuration applicaiton.

All we need to do is put configuration of differnt microservices for different enviornment in git repository. And we can connect Spring Cloud Config Server to git repository.

45. Step 02 - Setting up Limits Microservice

Initially retrieve values from application.properties, later connect this service to Spring Cloud Config Server to retrieve the configured values.

devtools - picking up application changes without restarting the server.

config client dependency - client that connects to a Spring Cloud Config Server to fetch the application's configuration.

46. Step 02 - Creating a hard coded limits service

Now we want our limit service to be able to connect to Spring Cloud Config Server and fetch the application's configuration.

Before that we would configure few values in application.properties and expose a service within limit service to retrieve those values.

47. Step 03 -Enhance limits service to get configuration from application properties

We can use @Value annotation to get values of applicaiton.properties, but spring boot provides better approach - @ConfigurationProperties

```
@Component
@ConfigurationProperties("limits-service")      //anything starting with this will be read
public class Configuration {

    private int minimum;
    private int maximum;

....
```

Spring Boot Update - @ConfigurationProperties is sufficient to register the bean as a Component. @Component can be removed.

If we are using @ConfigurationProperties its not sufficient to just create getters we will have to create setter as well, without setters it would give error.

48. Step 04 - Setting up Spring Cloud Config Server

We will create a new project and give it a name as spring-cloud-config-server. We will a add new dependency of Config Server in it. So we have only two dependencies -

Devtools - picking up application changes without restarting the server.

Config Server - Centralized management for configuration via Git, SVN or HashiCorp Vault.

50. Installing git -

As Discussed Spring Cloud Config Server accepts the configuration from the git repository.

We will connect LimitService to SpringCloudConfigServer.

Install a local git.

51. Creating local Git Repository -

Go to cmd.
mkdir git-localconfig-repo
cd to newly created folder
git init - will create a new git repo

Now from STS right click on SpringCloudConfigServer and goto Build -> Link Source
Select local git git-localconfig-repo

Now that folder will appear in explorer

Create file to store configuration for limit service in that folder - limits-servive.properties

Once we have added the file we will have to add it to local repo i.e. commit it in local repo ->

```
git add -A
git commit -m "first commit"
```

52. Connect Spring Cloud Config Server to Local Git Repository -

In application.properties -

```
spring.cloud.config.server.git.uri=file:///C:/Users/inarajp/Desktop/temp/spring-microservices-in28min-udemy/Practice/git-localconfig-repo
```

//here we are giving the location of local git repo, we can give url for external repo as well. As this local git repo we will have to add file://
//Observe three ///

Now we need to enable config server by adding - @EnableConfigServer to application class.

Now go to url - localhost:8888/limits-service/default   //limits-service is name we gave to properties file.

We can see Json properties in browser, which we had added in the limits-service.properties file.

We have now successfully established a connection between git repository with Spring Cloud Config Server. Next we need to connect limit-service with spring-cloud-config-server.

One of the important thing about SpringCloudConfigServer is that it stores configurations for multiple services, so we can store configurations for CurrencyCalculationService, CurrencyExchangeService, LimitService etc.
Also it can store configurations for each of these services for different enviornment.
For eg. the LimitService has 4 enviornment like DEV, QA, PROD and STAGE, we can store configuration related to all those 4 enviornments using the SpringCloudConfigServer.

53. Configuration for Multiple Environments in Git Repository

The limits-service.properties becomes the default configuration for LimitService.
However we can override them for specific enviornment.

Create a new files limits-service-dev.properties and limits-service-qa.properties

Now suppose content of limits-service-dev.properties are -

```
limits-service.minimum=1
#limits-service.maximum=111
```

the maximum is commented out, so it will pick up the value for maximum for the default file i.e. limits-service.properties

Now we need to commit these files. Whenever we are making any changes in git repo make sure to commit them -

```
git add -A  => to add new files in
git status => will show what the changed files
git commit -m "DEV and QA properties"
```

Now we check in browser -

```
http://localhost:8888/limits-service/qa
http://localhost:8888/limits-service/dev
```

We can see propertySources properties according to priority. Like limits-service-dev will be first and then default

54. Connect Limits Service to Spring Cloud Config Server

Now we don;t want LimitService to pick up the properties from applicaiton.properties, instead we want it to connect to spring-cloud-config-server and pick the properties from git repo.

If we want to pick up the configuration from spring-cloud-config-server then the applicaiton.properties file has to be renamed. We will start calling it bootstrap.properties
We also don't need to configure value in there as all the configuration of values will happen on spring-cloud-config-server.
Then we need to tell which url can be used to talk with spring-cloud-config-server

Note: We have named the applicaiton as limits-service => spring.application.name=limits-service
So we keep the name of properties file like that only - limits-service-dev.properties, limits-service-qa.properties, etc.
Based on that applicaiton name we will pick up the values from git repository.

When we start LimitService, we can see such log there -

```
: Fetching config from server at : http://localhost:8888

: Located environment: name=limits-service, profiles=[default], label=null, version=4068f6bbffb0e39aabd33c59ba43dc691b9e30a8, state=null
```

As we haven't configured the profile, it picks up the default

If we now execute http://localhost:8181/limits we will see the default properties from git repo file

So in short, LimitService connected to spring-cloud-config-server and picked properties from git repo default file. See code in LimitService rest method handler.

55. Debugging problems with Spring Cloud Config Server

https://github.com/in28minutes/in28minutes-initiatives/tree/master/The-in28Minutes-TroubleshootingGuide-And-FAQ#debugging-problems-with-spring-cloud-config-server

56. Configuring Profiles for Limits Service

We will now configure DEV propfile and QA profile and see what values are picked up.

Right now all the configurations for LimitService is comming from git repository, we are not configuring anything in the LimitService. Only thing which we configured in the LimitService is what is the URI of spring-cloud-config-server. Other than that all the other configuration for LimitService is been configure in git repository.
Advantage is that the entire configuration for LimitService is now separated from deployment of LimitService.

In bootstrap.properties -

```
spring.profiles.active=dev
```

We can configure profiles in n number of ways.

Now if we now execute http://localhost:8181/limits we will get =>

```
{
    "maximum": 888,                  // this comes from default as maximum in dev was commented out
    "minimum": 1
}
```

If we are making any change in git repo file, commit it (else changes won't reflect), then restart LimitService, because at the applicaiton startup the values are picked up from spring-cloud-config-server. So for new changes to reflect restart it.
Later we will see something called as refresh url to refresh the configuration from spring-cloud-config-server.

57. A review of Spring Cloud Config Server

One thing which we saw was we had to restart the LimitService to pick up the changes in configuration on git repository. It becomes more problem when there are multiple instances of LimitService.

58. Introduction to Currency Conversion and Currency Exchange Microservice

We will now create CurrencyCalculationService and CurrencyExchangeService

Check diagram

The CurrencyExchangeService will use JPA to talk with database and return a exchange value for a currency. Like USD to INR.

So the CurrencyCalculationService will use the CurrencyExchangeService to do the conversion from one currency to another of the any amount we provide.

59. Setting up Currency Exchange Microservice

60. Create a simple hard coded currency exchange service

```
@RestController
public class CurrencyExchangeController {

    @GetMapping("/currency-exchange/from/{from}/to/{to}")
    public ExchangeValue retrieveExchangeValue(@PathVariable String from, @PathVariable String to) {

        return new ExchangeValue(1000L, from, to, BigDecimal.valueOf(65));
    }
}


......

public class ExchangeValue {

    private Long id;
    private String from;
    private String to;
    private BigDecimal conversionMultiple;

    ....

    //getters, setters, constructors.

}
```

Hit - http://localhost:8000/currency-exchange/from/USD/to/INR

Response -

```
{
  "id": 1000,
  "from": "USD",
  "to": "INR",
  "conversionMultiple": 65
}
```

61. Setting up Dynamic Port in the the Response

As we dicussed we want CurrencyCalculationService to call CurrencyExchangeService.
We will create multiple instances of CurrencyExchangeService, right now we have one instance running on port 8000. Later we will run another instance on 8001 another on 8002, and so on.
And we would want CurrencyCalculationService to be talking to all these instances.
And we should be able to determine with which instance of CurrencyExchangeService that the CurrencyCalculationService is talking with. And to be able to do that we will use port as a distinguishing factor. So from every server(method handler) from CurrencyExchangeService we will return a port back. So that we know which instance is reponding back.

So add private int port;

```
public class ExchangeValue {

    private Long id;
    private String from;
    private String to;
    private BigDecimal conversionMultiple;

    private int port;

    ....

    //getters, setters, constructors.

}

@RestController
public class CurrencyExchangeController {

    @Autowired
    private Environment environment;

    @GetMapping("/currency-exchange/from/{from}/to/{to}")
    public ExchangeValue retrieveExchangeValue(@PathVariable String from, @PathVariable String to) {

        ExchangeValue exchangeValue = new ExchangeValue(1000L, from, to, BigDecimal.valueOf(65));

        exchangeValue.setPort(Integer.parseInt(environment.getProperty("local.server.port")));

        return exchangeValue;
    }
}
```

Now we should be able to run two instances at the same time. We will have to do this by selecting Run Configuration menu.
In VM arguments => -Dserver.port=8001

Whateverwe pass in VM arguments will override the properties in applicaiton.properties.

Afte running it, we will have two instances.

Try - http://localhost:8000/currency-exchange/from/USD/to/INR
Response -

```
{
  "id": 1000,
  "from": "USD",
  "to": "INR",
  "conversionMultiple": 65,
  "port": 8000
}
```

Try - http://localhost:8001/currency-exchange/from/USD/to/INR
Response -

```
{
  "id": 1000,
  "from": "USD",
  "to": "INR",
  "conversionMultiple": 65,
  "port": 8001
}
```

62. Step 16 - Configure JPA and Initialized Data

In previouse example we have hard coded the repsonse, it should come from DB.

Check the Code.

63. Step 18 - Setting up Currency Conversion Microservice

Create a new project with artifact id as currency-conversion-service

currency-conversion-service is CurrencyCalculationService

64. Step 19 - Creating a service for currency conversion

CurrencyExchangeService is telling you the rate

CurrencyCalculationService or currency-conversion-service will get the rate and then calculate for specified quantity

```
public class CurrencyConversionBean {

    private Long id;
    private String from;
    private String to;
    private BigDecimal conversionMultiple;
    private BigDecimal quantity;
    private BigDecimal totalCalculatedAmount;
    private int port;
    ......


}

.........


@RestController
public class CurrencyConversionController {

    @GetMapping("/currency-converter/from/{from}/to/{to}/quantity/{quantity}")
    public CurrencyConversionBean convertCurrency(@PathVariable String from, @PathVariable String to, @PathVariable BigDecimal quantity) {

            return new CurrencyConversionBean(1L, from, to, BigDecimal.ONE, quantity, quantity, 0);          //we have hard coded values here
        }
    }
```

....

Hit - http://localhost:8100/currency-converter/from/USD/to/INR/quantity/1600

Response -

```
        {
          "id": 1,
          "from": "USD",
          "to": "INR",
          "conversionMultiple": 1,
          "quantity": 1600,
          "totalCalculatedAmount": 1600,
          "port": 0
        }
```

65. Step 20 - Invoking Currency Exchange Microservice from Currency Conversion Micro

We can invoke CurrencyExchangeService from CurrencyCalculationService or currency-conversion-service using RestTemplate.

```
    @RestController
    public class CurrencyConversionController {

        @GetMapping("/currency-converter/from/{from}/to/{to}/quantity/{quantity}")
        public CurrencyConversionBean convertCurrency(@PathVariable String from, @PathVariable String to, @PathVariable BigDecimal quantity) {

            Map<String, String> uriVariables = new HashMap<>();
            uriVariables.put("from", from);
            uriVariables.put("to", to);

            ResponseEntity<CurrencyConversionBean> responseEntity = new RestTemplate().getForEntity(
                    "http://localhost:8000/currency-exchange/from/{from}/to/{to}",                //URI
                    CurrencyConversionBean.class,                //Response to be mapped to this entity
                    uriVariables);                //path variables values

            CurrencyConversionBean response = responseEntity.getBody();                                        //get response from ResponseEntity

            return new CurrencyConversionBean(
                    response.getId(),
                    from,
                    to,
                    response.getConversionMultiple(),
                    quantity,
                    quantity.multiply(response.getConversionMultiple()),
                    response.getPort());
        }
    }
```

Hit - http://localhost:8100/currency-converter/from/EUR/to/INR/quantity/1600

Response -

```
    {
      "id": 10002,
      "from": "EUR",
      "to": "INR",
      "conversionMultiple": 75.00,
      "quantity": 1600,
      "totalCalculatedAmount": 120000.00,
      "port": 8000
    }
```

66. Use Spring Cloud - Greenwich.RC2 and Spring Boot - 2.1.1.RELEASE

67. Step 21 - Using Feign REST Client for Service Invocation

One of the thing which we encounter was how difficult was it to call a rest webservice. We had to write lot of code. Lot of manual stuff to call a simple service. Feign solves this problem 1.

Feign makes it very easy to invoke other microservices or restfull web services.

Feign also provides integration with Ribbon which is client side load balancing framework.

Feign is one of the component which spring cloud inherits from Netflix.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

Once we have the dependency added we need to enable Feign to scan for clients - @EnableFeignClients("com.in28minudemy.microservices.currencyconversionservice")

Now what do we need to do to use Feign to invoke the service? Just like we use Repository to talk to JPA we need to create a Feign proxy to be able to talk to external microservice.

```
@FeignClient(name="currency-exchange-service", url="localhost:8000")
//This is a feign client, this is going to use feign to talk to external
microservice. We need to give name of the service which we are going to call (take
the name from CurrencyExchangeService application.properties)
public interface CurrencyExchangeServiceProxy {

    //define a method to talk to currency exchange service. Observe below method
    declare is same as method handler declaration in CurrencyExchangeController

    @GetMapping("/currency-exchange/from/{from}/to/{to}")
    public CurrencyConversionBean retrieveExchangeValue(@PathVariable("from")
    String from, @PathVariable("to") String to) ;
}
```

So the proxy know what is the name of microservice, what url to call, URI of the service(handler method), from and to, etc.

```
@RestController
public class CurrencyConversionController {

    @Autowired
    private CurrencyExchangeServiceProxy proxy;

    @GetMapping("/currency-converter-feign/from/{from}/to/{to}/quantity/{quantity}")
    public CurrencyConversionBean convertCurrencyFeign(@PathVariable String from,
    @PathVariable String to, @PathVariable BigDecimal quantity) {

        CurrencyConversionBean response = proxy.retrieveExchangeValue(from, to);

        return new CurrencyConversionBean(response.getId(),
                from,
                to,
                response.getConversionMultiple(),
                quantity,
                quantity.multiply(response.getConversionMultiple()),
                response.getPort());
    }
}
```

Hit - http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/1000

Response -

```
    {
      "id": 10002,
      "from": "EUR",
      "to": "INR",
      "conversionMultiple": 75.00,
      "quantity": 1000,
      "totalCalculatedAmount": 75000.00,
      "port": 8000
    }
```

Feign helps us to simple the client code to talk to a RestFul webservice.

Imagine a senario where this CurrencyExchangeService is offering 15 services, all the details of how to talk with those services will be at just one place i.e proxy. All the rest of the applicaiton need not know that CurrencyExchangeService is a RestFul service or I am talking to a other applicaiton. As far as the other component or class go, you just talk to a proxy, you are not worried how proxy is getting the details.

68. Step 22 - Setting up client side load balancing with Ribbon

Now we need to note the enviornments used by CurrencyCalculationService (currency-conversion-service) and CurrencyExchangeService. Check Diagram.

CurrencyCalculationService has 1 instance in PROD, however CurrencyExchangeService has 4 instances in PROD.

What we have done now is hard coded the URL for CurrencyExchangeService instance in the proxy - localhost:8000 (port - 8000)

We want currency-conversion-service instance to talk to any of the one instance of CurrencyExchangeService depending on the load. Load should be distributed amongst the instance of CurrencyExchangeService. This is where Ribbon comes in picture.

Check diagram.

We are using Feign to call CurrencyExchangeService. Ribbon can make use of the Feign configuration that we have already done and helps us distribute the calls between different instances of the CurrencyExchangeService.

So now we will enable Ribbon on CurrencyCalculationService.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
</dependency>
```

After adding the dependency enable the Ribbon -
@RibbonClient(name="currency-exchange-service")   //pass name of the applicaiton we want to talk to

```
//@FeignClient(name="currency-exchange-service", url="localhost:8000")
```

```
@FeignClient(name="currency-exchange-service")          //don't add url now, we will configure it in applicaiton.properties
@RibbonClient(name="currency-exchange-service")
public interface CurrencyExchangeServiceProxy {
    @GetMapping("/currency-exchange/from/{from}/to/{to}")
    public CurrencyConversionBean retrieveExchangeValue(@PathVariable("from") String from, @PathVariable("to") String to) ;
}
```

Now configure the urls for which the load for CurrencyExchangeService has to be distributed.
Now add in applicaiton.properties -

```
    currency-exchange-service.ribbon.listOfServers=http://localhost:8000, http://localhost:8001         //list of instances for CurrencyExchangeService we would want to talk to
```

69. Running client side load balancing with Ribbon

After running currency-conversion-service we can see that the response is comming from different instance for different requests.

In this step we lauched two instances of CurrencyExchangeService, and saw that Ribbon distributes the load from currency-conversion-service between these two instances.

70. Debugging problems with Feign and Ribbon

https://github.com/in28minutes/in28minutes-initiatives/tree/master/The-in28Minutes-TroubleshootingGuide-And-FAQ#debugging-problems-with-feign-and-ribbon

72. Step 24 - Understand the need for a Naming Server

Check diagram.

Lets assume that we started a third instance of CurrencyExchangeService, will Ribbon be able to distribute load to it, as the same code we have? If we want Ribbon to distribute load to the server we would have to add it to the configuration i.e. in applicaiton.properties. This means that we will have to change our configuration whenever a new server is created.

What we want to do is based on load we want to increase and decrease the number of services or instances. Dynamically increase or decrease them.

If we keep on changing configuration of currency-conversion-service based on the increase or decrease of number of instances, its becomes a difficult task. This is where the naming server comes in.

All the instances of all the microservices will register with the naming server. Whenever the instance of microservice comes up it will register itself with Eureka naming server. This is called a Service Registration.

And whenever a service wants to talk with another service, like currency-conversion-service wants to talk with CurrencyExchangeService, it would talk with naming server and ask what are the instances of CurrencyExchangeService that are currently running. This is called Service Discovery.
So the currency-conversion-service is asking for location of CurrencyExchangeService, the instances of CurrencyExchangeService.

The two important features of Naming Server is Service Registration and Service Discovery.

At startup of every application it will register itself with the Naming Server. And whenever they want details of another microservice they will do a Service Discovery.

73. Step 24 - Understand the need for a Naming Server

Things to do -
- Create component for Eureka Naming Server
- Update CurrencyCalculationService
- Connect the CurrencyExchangeService to Eureka Naming Server
- COnfigure Ribbon

Eureka Naming Server offered by Netflix.

Dependencies -

Eureka Server - spring-cloud-netflix Eureka Server

Config Client - If we want to store the configuration of Eureka server as well, something like Spring Config Server

Actuator

Devtools

After importing, let the build complete. The update maven project, then do clean install and again update maven project.

Enable Eureka server in applicaiton class - @EnableEurekaServer

In applicaiton.properties -

```
spring.application.name=netflix-eureka-naming-server
server.port=8761                                //default port for Naming Server

eureka.client.register-with-eureka=false        //for now we don't want server itself to register for naming server
eureka.client.fetch-registry=false
```

We can now run as java applicaiton.

Hit - http://localhost:8761/

For now there are no instances registered with Eureka naming server.

74. Step 26 - Connecting Currency Conversion Microservice to Eureka

To be able to make service connect to Eureka server we need to add dependency in then, i.e. in CurrencyCalculationService and CurrencyExchangeService -

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
</dependency>
```

We need to now add @EnableDiscoveryClient annotation in the applicaiton class, in order to make the applicaiton register to Naming server -

```
@SpringBootApplication
@EnableFeignClients("com.in28minudemy.microservices.currencyconversionservice")
@EnableDiscoveryClient
public class CurrencyConversionServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(CurrencyConversionServiceApplication.class, args);
    }
}
```

After this we will have to configure the url for Eureka in applicaiton.properties -

```
eureka.client.service-url.default-zone=http://localhost:8761/eureka
```

We can then launch up the currency-conversion-service, as soon as the applicaiton is up it will register itself with naming server.

75. Step 27 - Connecting Currency Exchange Microservice to Eureka

After following the same steps above we can see in Eureka that two instances of CurrencyExchangeService has been registered.

76. Step 28 - Distributing calls using Eureka and Ribbon

Now we have all three microservices registered with Eureka naming server.

Now we want CurrencyCalculationService to user Naming Server to find out details of CurrencyExchangeService, if it wants to talk with it.
So instead of hardcoding the url for Ribbon, we would want Ribbon to be talking to naming server and retrieve the details of all the instances of the service.
We earlier had done lot of things like we had add name of CurrencyExchangeService directly -

```
@FeignClient(name="currency-exchange-service")
@RibbonClient(name="currency-exchange-service")
public interface CurrencyExchangeServiceProxy {
    .....
}
```

In Eureka also the name of service is same.

We had also configured the list of services in applicaiton.properties -

```
currency-exchange-service.ribbon.listOfServers=http://localhost:8000,
http://localhost:8001
```

All we need to do, to enable Ribbon to talk to Naming server using the name in @RibbonClient is remove above configuration from applicaiton.properties.
Because in the application we have already configured Eureka -

```
eureka.client.service-url.default-zone=http://localhost:8761/eureka
```

So it already knows about Eureka, all we need to do is disable the list of servers.

So in currency-conversion-service, no where we have the location of where the CurrencyExchangeService is located. In proxy also we don't have any urls hardcoded.

Now we can try and hit CurrencyExchangeService and CurrencyCalculationService with feign urls. It should give proper response.

Thing to understand is without configuring the location of CurrencyExchangeService we are able to talk with it.

We can also bring up another instance and try.

We have achieved scaling up and scaling down of instances.

77. Debugging Problems with Naming Server (Eureka) and Ribbon

https://github.com/in28minutes/in28minutes-initiatives/tree/master/The-in28Minutes-TroubleshootingGuide-And-FAQ#debugging-problems-with-naming-server-eureka-and-ribbon

78. Step 29 - A review of implementing Eureka, Ribbon and Feign

CurrencyCalculationService is consumer of CurrencyExchangeService.

We started with direct connection between them.

So if we have other CurrencyExchangeService instances comming up the

CurrencyCalculationService was not able to talk to them.

And to be able to do load balancing we introduced Ribbon.

Before that we made use of Feign to CurrencyCalculationService to make it easier to call Rest services from CurrencyExchangeService. With Feign it becomes very easy to call RestFul webservices.

After that we added Ribbon and we load balanced between two instances of CurrencyExchangeService. Hardcoded its location urls at CurrencyCalculationService.

We thought that this is not good enough and we introduced the Naming Server.

We connected the CurrencyCalculationService and CurrencyExchangeService to the Naming Server.

And instead of hardcoding the urls for CurrencyExchangeService, we told CurrencyCalculationService to talk to the Naming Server to get the details of CurrencyExchangeService instances location or url.

In last step we were dynamically able to bring up and bring down the instances of CurrencyExchangeService without causing the problem to consumers (CurrencyCalculationService) and distribute the load between them.

This is what is needed in the world of microservices. There so many microservices that are talking to each other that we should be able to bring new instances up and old instances down without causing the problem to other consumers.

Ribbon - Client side load balancing, it enables clients to distribute load between the multiple service providers.

79. Step 30 - Introduction to API Gateways

Check diagram for Microservices Environments.

API GATEWAYS:

- Authentication, authorization and security
- Rate Limits
- Fault Tolerance
- Service Aggregation

Typically we would have 100s of microservices talking to each other and there are commmon features which we would want to implement for all these microservices.

We would want to make sure that every call to every microservice is Authenticated.

We would also want to implement things like Rate Limits. For a specific client we would want a certain number of calls per hour or per day.

We would want all microservices to be fault tolerant. If there is a service that I am dependent on and if it is not up I should be able to give default response back.

And in typical microservices enviornment there should also be some kind of Service Aggregation provided. Lets say there is an external consumer who wants to call 15 different services as part of one process. Its better to aggregate those 15 services and provide one service call for external consumer.

These are common features and these are implemented at the level of API Gateways.

So instead of allowing microservices to call each other directly what we will do is, we will make all the call go through a API Gateway. And API Gateway will take care of providing common features like Authentication, making sure that all service calls are logged, rate limts, fault tolerance, etc.

Beacuse all calls gets routed through the API Gateways, API Gateway also serve a great place for debugging as well as doing analytics.

We want to intercept all calls between all microservices and have them pass through a API Gateway. (Just like interceptors/filters, we did in ODB-Adapter)

80. Step 31 - Setting up Zuul API Gateway

We want to intercept all calls between all microservices and have them pass through a API Gateway.

Netflix provides an implementation called Zuul.

Steps -

- Create a component for Zuul API Gateway.
- Decide what it should do when it intercepts a request
- Make sure that all the important requests are configured to pass through the Zuul API Gateway

Create new project, dependency -

Zuul - Intelligent and programmable routing with Spring Cloud Netflix Zuul

Eureka Discovery Client - a REST based service for locating services for the purpose of load balancing and failovers of middle-tier servers.

So whenever Zuul instance is up and running we would like to see it in Eureka.

After importing the project -

Enable the Zuul Proxy - @EnableZuulProxy

Register with the Naming Server Eureka - @EnableDiscoveryClient

Now configure the applicaiton name, Eureka url, port -

```
spring.application.name=netflix-zuul-api-gateway-server
server.port=8765
eureka.client.service-url.default-zone=http://localhost:8761/eureka
```

API Gateway is now ready but we didn't yet told it what it should do when it intercepts a request.

81. Step 32 - Implementing Zuul Logging Filter

We will now add some logging to the Zuul API gateway.
So any request that come through the gateway, we will log it.

```
@Component
public class ZuulLoggingFilter extends ZuulFilter{

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    //Should this filter be executed or not. We can actually implement business logic and check certain things and decide if we want to execute the filter or not
    //For now we need to execute this filter for every request so we will return true.
    @Override
    public boolean shouldFilter() {
        return true;
    }

    //Real logic of interception
    @Override
    public Object run() throws ZuulException {

        HttpServletRequest request = RequestContext.getCurrentContext().getRequest();
        logger.info("request => {} request uri => {}", request, request.getRequestURI());

        return null;
    }

    //Defines whether the filtering should happen before the request is executed - return pre, or after the request is executed - return post or if we want to only filter the error requests that has caused exception to happen - return error
    @Override
    public String filterType() {
        return "pre";
    }


    //If we have multiple filters like ZuulSecurityFilter, ZuulLoggingFilter, etc. Then we can set priorty order between them over here
    //So we are return 1, means filter prority order is 1 for ZuulLoggingFilter
    @Override
    public int filterOrder() {
        return 1;
    }
}
```

Next we will see how to execute request using Zuul API gateway proxy.

82. Step 33 - Executing a request through Zuul API Gateway

Now when we hit CurrencyExchangeService - http://localhost:8000/currency-exchange/from/USD/to/INR

It executes fine. Now we want to execute this request through Zuul API Gateway.

If consumer directly calls thus url, the request would not go through the Zuul API Gateway. So how do we make the request to go through the API Gateway?

The port configured for API Gateway is 8765, so the url for invoking the request through the API Gateway would be http://localhost:8765/{applicaiton-name}/{uri-of-service}

{applicaiton-name} we can see in Naming Server or we can see in applicaiton.properties

So the url will be - http://localhost:8765/currency-exchange-service/currency-exchange/from/USD/to/INR

This request will now go through API Gateway and API Gateway will log request and then send the request out to the microservice.

We can see log in console.

However we want to send a request from currency-conversion-service to CurrencyExchangeService, we want it to be routed through the API Gateway. We will see it next.

83. Step 34 - Setting up Zuul API Gateway between microservice invocations

Previouly we used direct url to execute the CurrencyExchangeService through Zuul API Gateway. We will see how to do it from currency-conversion-service to CurrencyExchangeService.

So how do we get the request from currency-conversion-service to go through Zuul API Proxy.

The thing which actually makes a call inside currency-conversion-service is the proxy (CurrencyExchangeServiceProxy). We have already configured the Naming Server.

Everything is registered with Naming Server - CurrencyCalculationService, CurrencyExchangeService and also Zuul API Gateway.

So we will tell Feign, do not connect to currency-exchange-service, connect to Zuul API Gateway proxy. We will tell FeignClient to talk to netflix-zuul-api-gateway-server. It will talk to the Naming Server and get the uri for netflix-zuul-api-gateway-server.

Also We will have to add {applicaiton-name} to the URI.

```
@FeignClient(name="netflix-zuul-api-gateway-server")
@RibbonClient(name="currency-exchange-service")
public interface CurrencyExchangeServiceProxy {

    //define a method to talk to currency exchange service

    //@GetMapping("/currency-exchange/from/{from}/to/{to}")

    @GetMapping("/currency-exchange-service/currency-exchange/from/{from}/to/{to}")
    public CurrencyConversionBean retrieveExchangeValue(@PathVariable("from") String from, @PathVariable("to") String to) ;
}
```

Hit the URL - http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/1000
Now we are going through Feign.

Now the request from currency-conversion-service is going through Zuul API Gateway to CurrencyExchangeService.

Lets say we want API Gateway to be executed even before the CurrencyCalculationService is invoked. The above url - http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/1000, will not do that.
So we can do this by using below url -

http://localhost:8765/{app-name}/{uri} i.e. => http://localhost:8765/currency-conversion-service/currency-converter-feign/from/EUR/to/INR/quantity/1000

Tip: Zuul uses AppName in the url to talk to Eureka and find the url of the service.

After hitting the URL we can see that Zuul API Gateway Filter is logging both, before request is executed for currency-conversion-service and before the request is executed for CurrencyExchangeService -

```
2020-03-04 11:35:23.307  INFO 1027228 --- [nio-8765-exec-3] c.i.m.n.ZuulLoggingFilter                : request => org.springframework.cloud.netflix.zuul.filters.pre.Servlet30RequestWrapper@37f6205d request uri => /currency-conversion-service/currency-converter-feign/from/EUR/to/INR/quantity/1000

2020-03-04 11:35:23.326  INFO 1027228 --- [nio-8765-exec-4] c.i.m.n.ZuulLoggingFilter                : request => org.springframework.cloud.netflix.zuul.filters.pre.Servlet30RequestWrapper@5d32751b request uri => /currency-exchange-service/currency-exchange/from/EUR/to/INR
```

84. 98. Debugging Problems with Zuul API Gateway

https://github.com/in28minutes/in28minutes-initiatives/tree/master/The-in28Minutes-TroubleshootingGuide-And-FAQ#debugging-problems-with-zuul-api-gateway

85. Step 35 - Introduction to Distributed Tracing

Now lets say that the service is not working properly and there is a small defect and we would want to debug it. How do we do that? where would we look? CurrencyCalculationService, CurrencyExchangeService or API Gateway? Where the defect is? How do I know what is happening with that total request?

One of the important thing we should have with microservices architechture is distributed tracing.

I would want one place where I would go and see what happened with the specific request. Centralized location where we would see a complete chain of what happened with a specific request.

As n number of components are invovled we would need a centralized information. This is where the distributed tracing comes into picture.

There are variety of options available for distributed tracing, we will use Spring Cloud Sleuth with Zipkin.

One of important thing is to assign a unique id to a request.

So lets say a request is going through a set of application components -> API Gateway -> CurrencyCalculationService -> API Gateway -> CurrencyExchangeService, so how do we identify this request is same one. Only way to identify is by assigning an id to the request. That's what is Spring Cloud Sleuth.
Spring Cloud Sleuth would assign a unique id to a request, so that we could trace it across the components.

Zipkin is what we call a distributed tracing system.

What we would do is, we would put all the logs from all these services to a MQ (Rabbit MQ) and we will send it out to Zipkin Server, where it would be consolidated and we would be able to look throught the different request and find what happens with a specific request.

86. Step 36 - Implementing Spring Cloud Sleuth

Decide where all we would like to use Spring Cloud Sleuth. It would add a unique id to a request so that we can trace it across multiple components.

We would add Spring Cloud Sleuth in CurrencyCalculationService, CurrencyExchangeService and API Gateway. Do exerside to add it to other components.

Two steps to add -
- Adding a dependency to pom.xml
- Tell what all request we want to intercept

If we want to trace all the request then we need to create something called always sampler.

```
import brave.sampler.Sampler;

@EnableZuulProxy
@EnableDiscoveryClient
@SpringBootApplication
public class NetflixZuulApiGatewayServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(NetflixZuulApiGatewayServerApplication.class, args);
    }

    @Bean
    public Sampler defaultSampler() {
        return Sampler.ALWAYS_SAMPLE;
    }
}
```

Implement these two changes to all services.

We added few logs in controller of CurrencyCalculationService and CurrencyExchangeService.

So when we hit the url we can see in log that same request id has been there for all. But this log is distributed in multiple places. Its in multiple consoles. This is where the need for distributed tracing comes in.

We would want to centralized all this logs at one place. This is where Zipkin comes in.

87. Step 37 - Introduction to Distributed Tracing with Zipkin

Check Diagram - Zipkin Distributed Tracing.

There are variety of options for Distributed Tracing - ELK Stack - elastic search, Log Stash and Kibana.

Here we will use Zipkin to get consolidated view to see what is happening with our microservices.

We will get all the logs from the individual microservices to go to the Zipkin Distributed Tracing Server. After that we can use a UI provided by Zipkin to look at what happened to a specific request.

Now the question is how we get logs from a microservice to Zipkin Distributed Tracing Server?

We will use a Rabbit MQ, whenever there is a log message the microservice will put it on the queue and Zipkin Distributed Tracing Server will be picking it up from the queue.

Typically Zipkin Distributed Tracing Server is connected to a database. We will use in memory db. We will have all log messages in memory and Zipkin will search through them and give us a big picture of what happening with a request.

88. Step 38 - Installing Rabbit MQ

89. Updates to Step 39 - Running Zipkin on Windows

90. Step 39 - Setting up Distributed Tracing with Zipkin

Installing Zipkin and making it listen on Rabbit MQ.

In earlier versions of springs we could have found Zipkin UI and other more dependencies in Spring Initializr. But it was removed. We now need to download zipkin server.

Zipkin jar is copied in Practice folder.

```
Open cmd -> java -jar zipkin.jar                // this will start zipkin server
```

We can check zipkin dashboard - http://localhost:9411/zipkin/

Now we want Zipkin to listen on Rabbit MQ, so start Zipkin with below two commands =>

```
                Command 1 => SET RABBIT_URI=amqp://localhost
                Command 2 => java -jar zipkin.jar
```

91. Step 40 - Connecting microservices to Zipkin

Now we will connect CurrencyCalculationService, CurrencyExchangeService and API Gateway to Rabbit MQ. To do that we will have to add some dependencies.

```
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-sleuth-zipkin</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-bus-amqp</artifactId>
        </dependency>
```

Because of spring-cloud-sleuth-zipkin we would start logging these messages in the format that zipkin will understand.

spring-cloud-starter-bus-amqp, we are establishing a connection to amqp bus and the default amqp installation which is used is Rabbit MQ. We will be able to connect to Rabbit MQ.

92. Updates to Step 40 : Use spring-cloud-starter-zipkin and spring-rabbit

93. Step 41 - Using Zipkin UI Dashboard to trace requests

So now following applicaitons will be running - CurrencyCalculationService, CurrencyExchangeService, API Gateway, Naming server and Zipkin Server.

We didn't do is connect the Zipkin distributed server to Eureka Naming Server. This is a exersize.

94. Debugging Problems with Zipkin

https://github.com/in28minutes/in28minutes-initiatives/tree/master/The-in28Minutes-TroubleshootingGuide-And-FAQ#debugging-problems-with-zipkin

LOTS OF USERFUL VEDIOS LINK HERE... CHECK....

95. Step 42 - Understanding the need for Spring Cloud Bus

Peviously we had connected the LimitService to the Spring Cloud Config Server.

We had stored the configurations of the different enviornments of the LimitService into the git repository and we were able to connect LimitService to Spring Cloud Config Server to retrieve the configuration.

However there is one problem unsolved, in this step we will understand that.

Now when we start SpringCloudConfigServer and LimitService we can fetch the configuration of specified enviornment.

Now we will start one more instance of LimitService. So two instances of LimitService are up and running. Check - http://localhost:8183/limits

Changed properties for qa and git commit. Now when we hit http://localhost:8183/limits we still don't get updated value, even if we did a commit. We are not seeing changes on both the instances. So how do we make changes reflect in LimitService?

We can do this by executing a simple request - http://localhost:8080/actuator/refresh   - this gives us error - Resource not found error.

We need to have actuator in LimitService pom.

Spring Boot 2.0.0+ > Enable all Actuator URLs => management.endpoints.web.exposure.include=*

We will need to turn off the security on Spring Boot starter Actuator => management.security.enabled=false

Now hit POST request => http://localhost:8182/actuator/refresh, we can see the request executes sucessfully and when we again hit http://localhost:8182/limits, we will get the updated value.

Now we had refreshed instance with port 8182 so changes will reflect for this instance but not for 8183 instance. To get changes for 8183 again hit => http://localhost:8183/actuator/refresh, and now check http://localhost:8183/limits, we will get the updated value.

In this example we have only two instances of LimitService, but suppose we have 100 instances, then to referesh the changes in git we will have to hit 100 refresh urls. This is not good.
Image this with multiple microservices each with multiple instances.
Every time we make change in configuration and we want configuration to reflect in a microservice we don't want to call thousand urls.
This is where Spring Cloud Bus provides us the solution.

We can have one URL for all the instances and once we hit that URL all the instances of the microservice will be updated with the latest values from the git configuration.

96. Step 43 - Implementing Spring Cloud Bus

There are multiple options present with Spring Cloud Bus - Kafka, Rabbit MQ, etc.
We will use Rabbit MQ.
Check if Rabbit MQ service is running.

We need to connect both SpringCloudConfigServer and LimitService to Spring Cloud Bus. Add a dependency -

```
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-bus-amqp</artifactId>
    </dependency>
```

Now make a change in configuraition and hit - http://localhost:8182/actuator/bus-refresh => this would refresh configuration of all the instances of a microservice. (Changes were picked up even without a commit. But I think we should commit.)

The way the Spring Cloud Bus works is at applicaiton start up all the microservices instances register with the cloud bus. When there is any change in the configuraition and refresh is called on any of these instances, the microservice instance would send an event over to the Spring Cloud Bus. And the Spring Cloud Bus will propagate that event to all the microservice instances that are registered with it.

Thing about spring boot, as soon as we add a dependency everything is configured for us. We have Rabbit MQ running in the back ground, spring boot detects that it sees that there is an amqp dependency in the class path, it would automatically configure a connection to Rabbit MQ.

97. Step 44 - Fault Tolerance with Hystrix

Microservices architechture consists of number of components. Instead of having one big monolithic applicaiton we have a number of microservices interacting with each other.
It is possible that a couple of microservices might be down somewhere in the entire architechture. If these microservices are down then they can pull down entire chain of microservices that are dependent on them.

For example CurrencyCalculationService dependents on CurrencyExchangeService and CurrencyExchangeService dependents on LimitService. In this case if the LimitService goes down then both those services will also not be available. This is not good. This is where fault tolerance comes into picture.

We need to check if some service goes down then can it send a good enough response back, so that its dependent microservice can work. It would prevent from entire chain from going down.

Hystrix framework helps us to develop fault tolerance microservices.

Add Hystrix as dependency in LimitService pom -

```
    <dependency>
```

```
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
        </dependency>
```

After this enable it in applicaiton class - @EnableHystrix - this will enable hystrix fault tolerance on all the controllers. And on all the controller methods we can add a annotation @HystrixCommand(fallbackMethod), we can specify fallback method.

```
    @GetMapping("/fault-tolerance-example")
    @HystrixCommand(fallbackMethod = "fallbackRetrieveConfiguration")
    public LimitConfiguration retrieveConfiguration() {
        throw new RuntimeException("Not available");        // when exception is thrown the fallback method will be called
    }

    public LimitConfiguration fallbackRetrieveConfiguration() {
        return new LimitConfiguration(9, 999) ;             //here we can return default
    }
```

Now hit fault-tolerance-example => it will return values which we have set in fallbackMethod