1. How Function interface advantage
2. How to handle exception in microservices

@ControllerAdvice

public class ExceptionController {

@ExceptionHandler(NotFoundException.class)

public ResponseEntity<List<ErrorDto>> handleNotFoundException(NotFoundException e){

final String message = "The provided id is not in DB" + e.getMessage() + " " + "not found";

final List<ErrorDto> errorDto = new ArrayList<>();

errorDto.add(new ErrorDto("400", message, e.getField()));

return new ResponseEntity<>(errorDto, HttpStatus.BAD\_REQUEST);

}

}

1. **Feign vs RestTemplate vs Webclient**

**Feign:**

URLs are not hardcoded.

You don't have to write unit test cases for feign as there is no code to test however you have to write integration tests.

We can use Eureka Client ID instead of the URL.

Feign handled the actual code.

Feign integrates with Ribbon and Eureka Automatically.

Feign provides a very easy way to call RESTful services.

**RestTemplate**: is synchronous and blocking i.e. when you do a rest call you need to wait till the response comes back to proceed further.

RestTemplate are blocking in nature and uses one thread-per-request model of Java Servlet API. It means that RestTemplate will wait for the response everytime it dispatches a request to the remote server.

By default, RestTemplate creates new Httpconnection every time and closes the connection once response is received and processed.

**WebClient**: is complete opposite of this. The caller need not wait till response comes back. Instead he will be notified when there is a response.

Opposite to RestTemplate, WebClient is asynchronous and non-blocking in nature. It follows events-driven architecture from reactive framework of Spring WebFlux.

Using WebClient, the client need not wait till response comes back. Instead it will be notified usign a callback method when there is a response from the server.

When we invoke an API through WebClient that returns a Mono or a Flux, it will return immediately. The results of the call will be delivered to us through the mono or flux callbacks when they become available.

1. Future, Executor, Runnable, Callable
2. wait and sleep
3. Fail fast and fail safe
4. Group by deprt a employee in java
5. 2nd max salary

* Select max(Salary) from Employee where Salary < (Select max(Salary) from Employee)
* Select max(Salary) from Employee where Salary NOT IN (Select max(Salary) from Employee)
* Select TOP 1 Salary from (Select DISTINCT TOP 2 Salary from Employee Order By Salary Desc) Result Order By Salary
* With RESULT AS ( Select Salary DENSE\_RANK() over (Order By Salary Desc) as DenseRank from Employee) select top 1 salary from RESULT Where DenseRank = N

1. **Print one to 100 without loop**

public class print1to100 {

public static void main(String[] args) {

int number = 1;

printNumbers(number);

}

public static void printNumbers(int num){

if(num <= 100){

System.out.print(num +" ");

printNumbers(num + 1);

}

}

}

**Difference between Application context and Beanfactory in Spring framework**

1. BeanFactory uses lazy initialization **but** ApplicationContext uses eager initialization. In case of BeanFactory, bean is created when you call getBeans() method, but bean is created upfront in case of ApplicationContext when the ApplicationContext object is created.
2. BeanFactory explicitly provide a resource object using syntax **but** ApplicationContext creates and manages resource objects on its own.
3. BeanFactory doesnt support internatiolization **but** ApplicationContext supports internationalization.
4. With BeanFactory annotation based dependency injection is not supported **but** annotation based dependency injection is supported in ApplicationContext.

**Using BeanFactory:**

BeanFactory beanfactory = new XMLBeanFactory(new FileSystemResource("spring.xml"));

Triangle triangle =(Triangle)beanFactory.getBean("triangle");

**Using ApplicationContext:**

ApplicationContext context = new ClassPathXMLApplicationContext("spring.xml")

Triangle triangle =(Triangle)context.getBean("triangle");

PUT vs POST

* PUT method is called when you have to modify a single resource while POST method is called when you have to add a child resource.
* PUT method response can be cached but you cannot cache POST method responses.
* You can use UPDATE query in PUT whereas you can use create query in POST.
* In PUT method, the client decides which URI resource should have, and in POST method, the server decides which URI resource should have.
* PUT works as specific while POST work as abstract.
* If you send the same PUT request multiple times, the result will remain the same but if you send the same POST request multiple times, you will receive different results.
* PUT method is idempotent whereas POST method is not idempotent.
* HTTP PUT http://www.google.com/users/234
* HTTP PUT http://www.google.com/users/234/accounts/567
* HTTP POST http://www.google.com/users
* HTTP POST <http://www.google.com/users/234/accounts>

1. How To Get The Context Path In A Spring Web Application

Simply you can add a HttpServletRequest parameter to your controller method and then get the context path using getContextPath() method

@RequestMapping(value = "/", method = RequestMethod.GET)

public String home(HttpServletRequest request) throws IOException {

System.out.println(request.getContextPath());

return "home";

}

If you want to get the context path from within a service or a component or anywhere inside your application and you don’t want to pass it as a parameter from your controller, then you can use ServletContext.

Simply add a class field of type ServletContext and annotate it with @Autowired.

@Autowired

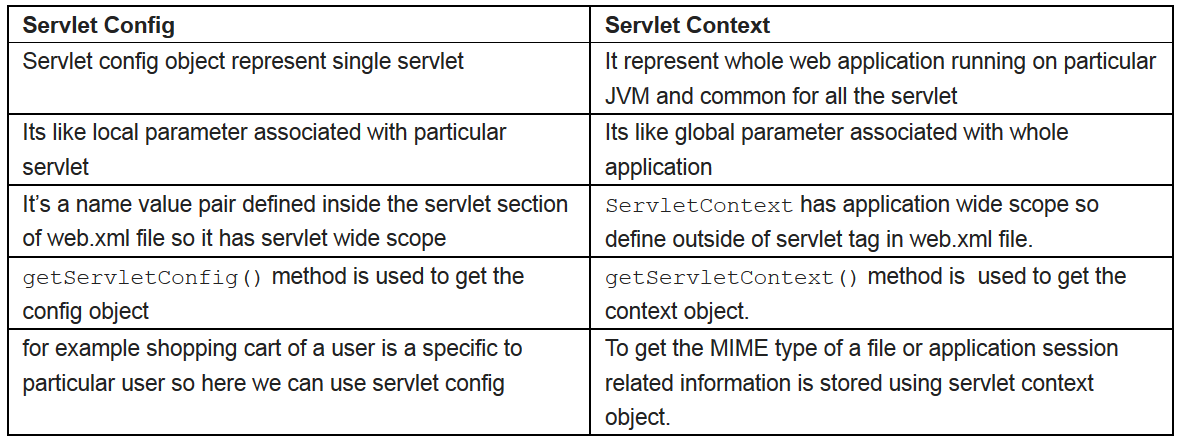
private ServletContext context;

Now inside your method, you can get the context path through:

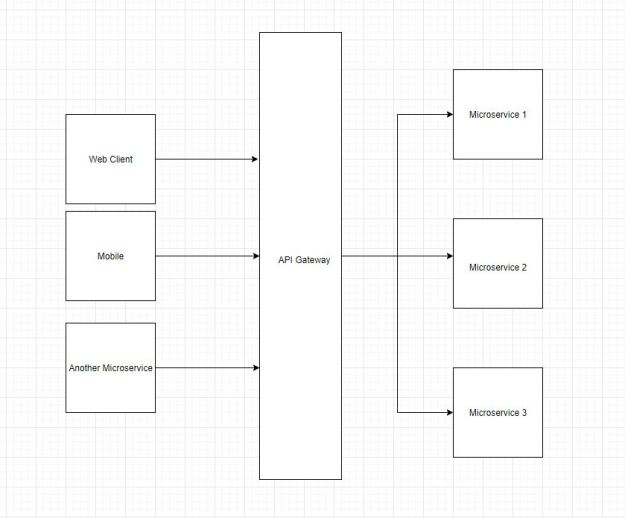
context.getContextPath()

https://roytuts.com/how-to-get-servletcontext-and-servletconfig-object-in-a-spring-bean/

| ServletConfig | ServletContext |
| --- | --- |
| ServletConfig is servlet specific | ServletContext is for whole application |
| Parameters of servletConfig are present as name-value pair in <init-param> inside <servlet>. | Parameters of servletContext are present as name-value pair in <context-param> which is outside of <servlet> and inside <web-app> |
| ServletConfig object is obtained by getServletConfig() method. | ServletContext object is obtained by getServletContext() method. |
| Each servlet has got its own ServletConfig object. | ServletContext object is only one and used by different servlets of the application. |
| Use ServletConfig when only one servlet needs information shared by it. | Use ServletContext when whole application needs information shared by it |



1. Microservices integration patterns



API Gatwey: API Gateway is routing the request from the client to the appropriate server or microservice.

API Gateway can also implement extra common functionalities and in-process reducing the load from microservices.  These common functionalities include logging, authentication, authorization, load balancing, response caching, retry policies, circuit breakers, rate limiter.

* Spring Cloud API Gateway
* Netflix API Gateway (Zuul)
* [Amazon API Gateway](https://aws.amazon.com/api-gateway/)
* Mulesoft
* Kong API Gateway
* Azure API Gateway

@Configuration

public class SpringCloudConfig

{

@Bean

public RouteLocator gatewayRoutes(RouteLocatorBuilder routeLocatorBuilder)

{

return routeLocatorBuilder.routes()

.route("customerModule", rt -> rt.path("/customer/\*\*").uri("http://localhost:8081/"))

.route("vendorModule", rt -> rt.path("/vendor/\*\*").uri("http://localhost:8082/"))

.build();

}}

web services vs rest api vs microservices