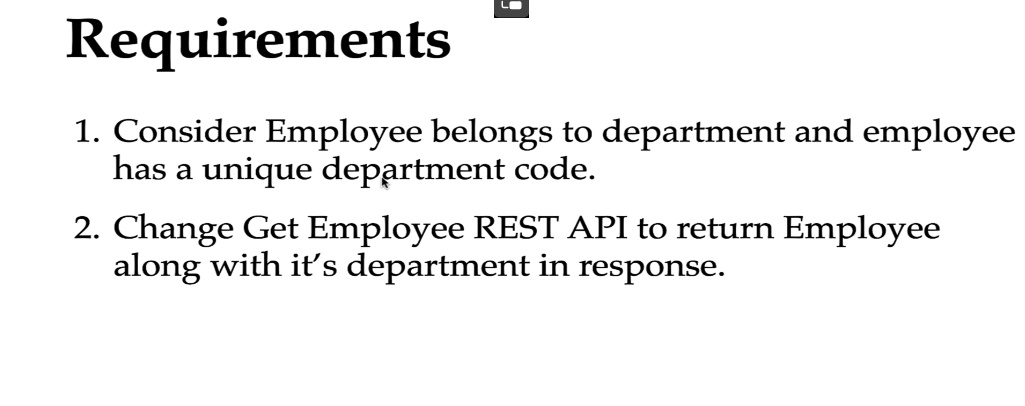
**RestTemplate**



* RestTemplate foloseste Thread per request approach, ceea ce e cam rau cand avem foarte multe requests si toate o sa astepte pentru response, caci threadurile vor fi active cat response nu va veni, ca sa il astepte. Iata de ce, RestTemplate este o **blocking** solution.
* El fooseste doar comunicarea **syncronizata**

Deci, putem crea un APIResponseDto ca sa returnam si EmployeeDto si DepartmetnDto al lui:

@Override  
public APIResponseDto getEmployeeById(Long id) {  
 Employee employee = employeeRepository.findById(id)  
 .orElseThrow(() ->  
 new ResourceNotFoundException("User","id",String.*valueOf*(id)));  
 ResponseEntity<DepartmentDto> responseEntity = restTemplate.getForEntity("http://localhost:8080/api/departments/"+employee.getDepartmentCode(), DepartmentDto.class);  
 DepartmentDto departmentDto = responseEntity.getBody();  
  
 APIResponseDto apiResponseDto = new APIResponseDto(employeeMapper.employeeToEmployeeDto(employee),departmentDto);  
  
 return apiResponseDto;  
}

.getBody() – returneaza obiectul de la ResponseEntity

* Asa se face un post

public void addEmployeeDemo() throws URISyntaxException {

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

URI uri = new URI("http://localhost:8080/employee");

Employee objEmp = new Employee();

objEmp.setName("Krishna");

objEmp.setCity("Noida");

HttpEntity<Employee> httpEntity = new HttpEntity<>(objEmp, headers);

RestTemplate restTemplate = new RestTemplate();

Employee employee = restTemplate.postForObject(uri, httpEntity, Employee.class);

System.out.println("Id: " + employee.getEmpId());

}

**WebClient**

* Suporta atat syn communication cat async
* Suporta si streaming scenarios pentru async
* este parte din WebFux dependency
* El nu mai foloseste Thread Per Request approach, si fiecare reqeust va fi un task, si taskurile date vor fi executate in cateva threaduri doar si nu se va astepta response pentru fiecare request
* WebClient foloseste Event-driven architecture
* Deci, request e trimis, si nu se asteapta ca el sa vina. Cand va veni, se va declansa un event si va fi returnat, si asa nu se consuma resurse degeaba cat e asteptat.Iata de ce, WebClient este un **non-blocking** solution
* Avem nevoie de dependenta:
* <dependency>  
   <groupId>org.springframework.boot</groupId>  
   <artifactId>spring-boot-starter-webflux</artifactId>  
  </dependency>

Se creaza asa

@Bean  
@Lazy  
public WebClient restTemplate(){  
 return WebClient.*create*();  
}

* **Metode:**

.get().uri(“”)

.post().uri(“”)

.put().uri(“”)

.delete().uri(“”)

DepartmentDto departmentDto = webClient.get().uri("http://localhost:8080/api/departments/"+employee.getDepartmentCode())  
 .retrieve().bodyToMono(DepartmentDto.class).block();

totusi, requestul e abea trimis cand se apeleaza block(). Atunci se prea block si se trimite propriu zis request-ul.

* **Flux** – este un stream de elemente
* **Mono** –

Asa ar fi un POST:

WebClient webClient = WebClient.create("http://localhost:3000");

Employee createdEmployee = webClient.post()

.uri("/employees")

.header(HttpHeaders.CONTENT\_TYPE, MediaType.APPLICATION\_JSON\_VALUE)

.body(Mono.just(empl), Employee.**class**)

.retrieve()

.bodyToMono(Employee.**class**);

**Spring Cloud OpenFeign**

1. Mai trebuie dependenta asta, pe langa cea de spring cloud:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

<version>4.0.4</version>

</dependency>

1. La main class folosim anotatia

@EnableFeignClients

Aceasta anotatie permite component scaning pentru interfetele care se declara a fi Feign clients

1. Cream un Feign API Client

Pentru asta cream o interfata si o anotam cu @FeignClient

@FeignClient  
public interface APIClient {  
}

Si ea va deveni un FeignClient ve va fi implementanta de Spring Cloud

* In ea vom pune metodele care vo trebui apelate din alt microservice. De ex, interfata e creata in Employee service, care va apela Department service, deci metoda ce va fi apelata in RestController la Department service o punem in interfata, dar punem doar prototipul, fara ResponseEntity la return si cu tot cu @GetMapping sau orice punem, dar cu full path!
* @FeignClient  
  public interface APIClient {  
   @GetMapping("api/departments/{department-code}")  
   DepartmentDto getDepartmentByDepartmentCode(@PathVariable("department-code") String departmentCode);  
   }

1. Acum, anotatia @FeignClient ne ofera suport pentru a insera host:port la microservice strain si numele la feign client

@FeignClient(url = "http://localhost:8080", value = "DEPARTMENT-SERVICE")  
public interface APIClient {  
 @GetMapping("api/departments/{department-code}")  
 DepartmentDto getDepartmentByDepartmentCode(@PathVariable("department-code") String departmentCode);  
}

1. Acum pur si simplu folosim acest feign client in service la Employee

si folosim metoda necesara din el si gata:

@Override  
public APIResponseDto getEmployeeById(Long id) {  
 Employee employee = employeeRepository.findById(id)  
 .orElseThrow(() ->  
 new ResourceNotFoundException("User","id",String.*valueOf*(id)));  
 DepartmentDto departmentDto = apiClient.getDepartmentByDepartmentCode(employee.getDepartmentCode());  
  
  
 APIResponseDto apiResponseDto = new APIResponseDto(employeeMapper.employeeToEmployeeDto(employee),departmentDto);  
  
 return apiResponseDto;  
}

* Partea buna e ca nu trebuie sa scriem unit tests pentru el