

PCF Cloud

[Document subtitle]



February 5, 2020

[Company address]

Login to cloud foundry

cf login -a https://api.system.pcfpre-phx.cloud.boeing.com –sso

Push to cloud foundry

cf push crud -p crud-1.1.0.war -b <https://github.com/cloudfoundry-community/tomee-buildpack.git>

<**parent**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-parent</**artifactId**>  
 <**version**>2.0.2.RELEASE</**version**>  
 <**relativePath**/>  
</**parent**>

<**dependency**>  
 <**groupId**>org.apache.struts</**groupId**>  
 <**artifactId**>struts2-spring-plugin</**artifactId**>  
 <**version**>2.5.18</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.apache.struts</**groupId**>  
 <**artifactId**>struts2-core</**artifactId**>  
 <**version**>2.5.18</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.apache.tomcat.embed</**groupId**>  
 <**artifactId**>tomcat-embed-jasper</**artifactId**>  
 <**version**>8.5.31</**version**>  
</**dependency**>

@ SpringBootApplication == @SpringBootConfiguration+@EnableAutoConfiguration+@ComponentScan

@SpringBootApplication  
**public class** CRUDApplication **extends** SpringBootServletInitializer {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(CRUDApplication.**class**);  
  
 }  
}

@Configuration  
**public class** CRUDConfiguration {  
 @Bean  
 **public** FilterRegistrationBean filterRegistrationBean() {  
 FilterRegistrationBean registration = **new** FilterRegistrationBean();  
 registration.setFilter(**new** StrutsPrepareAndExecuteFilter());  
 registration.addUrlPatterns(**"/\*"**);  
 registration.setDispatcherTypes(DispatcherType.***REQUEST***, DispatcherType.***FORWARD***);  
 registration.setName(**"StrutsPrepareAndExecuteFilter"**);  
 **return** registration;  
 }  
}

@RestController=@Controller+@ResponseBody

**Service Discovery**

Service instances have dynamically assigned network locations. Moreover, the set of service instances changes dynamically because of auto-scaling, failures, and upgrades. Consequently, your client code needs to use a more elaborate service discovery mechanism.

There are two main service discovery patterns: [client-side discovery](http://microservices.io/patterns/client-side-discovery.html) and [server-side discovery](http://microservices.io/patterns/server-side-discovery.html). Let’s first look at client-side discovery.

**The Client-Side Discovery Pattern**

When using [client-side discovery](http://microservices.io/patterns/client-side-discovery.html), the client is responsible for determining the network locations of available service instances and load balancing requests across them. The client queries a service registry, which is a database of available service instances. The client then uses a load-balancing algorithm to select one of the available service instances and makes a request.

The client-side discovery pattern has a variety of benefits and drawbacks. This pattern is relatively straightforward and, except for the service registry, there are no other moving parts. Also, since the client knows about the available services instances, it can make intelligent, application-specific load-balancing decisions such as using hashing consistently. One significant drawback of this pattern is that it couples the client with the service registry. You must implement client-side service discovery logic for each programming language and framework used by your service clients.

**The Server-Side Discovery Pattern**

The client makes a request to a service via a load balancer. The load balancer queries the service registry and routes each request to an available service instance. As with client-side discovery, service instances are registered and deregistered with the service registry.

The [AWS Elastic Load Balancer](https://aws.amazon.com/elasticloadbalancing/) (ELB) is an example of a server-side discovery router. An ELB is commonly used to load balance external traffic from the Internet. However, you can also use an ELB to load balance traffic that is internal to a virtual private cloud (VPC). A client makes requests (HTTP or TCP) via the ELB using its DNS name. The ELB load balances the traffic among a set of registered Elastic Compute Cloud (EC2) instances or EC2 Container Service (ECS) containers. There isn’t a separate service registry. Instead, EC2 instances and ECS containers are registered with the ELB itself.

The server-side discovery pattern has several benefits and drawbacks. One great benefit of this pattern is that details of discovery are abstracted away from the client. Clients simply make requests to the load balancer. This eliminates the need to implement discovery logic for each programming language and framework used by your service clients. Also, as mentioned above, some deployment environments provide this functionality for free. This pattern also has some drawbacks, however. Unless the load balancer is provided by the deployment environment, it is yet another highly available system component that you need to set up and manage.

**Service Registration Options**

**The Self-Registration Pattern**

When using the [self-registration pattern](http://microservices.io/patterns/self-registration.html), a service instance is responsible for registering and deregistering itself with the service registry. Also, if required, a service instance sends heartbeat requests to prevent its registration from expiring.

The self-registration pattern has various benefits and drawbacks. One benefit is that it is relatively simple and doesn’t require any other system components. However, a major drawback is that it couples the service instances to the service registry. You must implement the registration code in each programming language and framework used by your services.

**The Third-Party Registration Pattern**

When using the [third-party registration pattern](http://microservices.io/patterns/3rd-party-registration.html), service instances aren’t responsible for registering themselves with the service registry. Instead, another system component known as the *service registrar*handles the registration. The service registrar tracks changes to the set of running instances by either polling the deployment environment or subscribing to events. When it notices a newly available service instance it registers the instance with the service registry. The service registrar also deregisters terminated service instances. The following diagram shows the structure of this pattern.

 Example of a service registrar is [NetflixOSS Prana](https://github.com/netflix/Prana). Primarily intended for services written in non-JVM languages, it is a sidecar application that runs side by side with a service instance. Prana registers and deregisters the service instance with Netflix Eureka.

The third-party registration pattern has various benefits and drawbacks. A major benefit is that services are decoupled from the service registry. You don’t need to implement service-registration logic for each programming language and framework used by your developers. Instead, service instance registration is handled in a centralized manner within a dedicated service.

One drawback of this pattern is that unless it’s built into the deployment environment, it is yet another highly available system component that you need to set up and manage.

@EnableDiscoveryClient

<**dependency**>  
 <**groupId**>io.pivotal.spring.cloud</**groupId**>  
 <**artifactId**>spring-cloud-services-starter-service-registry</**artifactId**>  
</**dependency**>

<**dependencyManagement**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>io.pivotal.spring.cloud</**groupId**>  
 <**artifactId**>spring-cloud-services-dependencies</**artifactId**>  
 <**version**>2.0.1.RELEASE</**version**>  
 <**type**>pom</**type**>  
 <**scope**>import</**scope**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-dependencies</**artifactId**>  
 <**version**>Finchley.RELEASE</**version**>  
 <**type**>pom</**type**>  
 <**scope**>import</**scope**>  
 </**dependency**>  
 </**dependencies**>  
</**dependencyManagement**>

**eureka.client.fetch-registry**=**true**

eureka.client.healthcheck.enabled=true should only be set in application.yml. Setting the value in bootstrap.yml causes undesirable side effects, such as registering in Eureka with an UNKNOWN status

@Autowired

private DiscoveryClient discoveryClient;

**List<ServiceInstance> instances=discoveryClient.getInstances("employee-producer");**

**ServiceInstance serviceInstance=instances.get(0);**

**String baseUrl=serviceInstance.getUri().toString();**

baseUrl=baseUrl+"/employee";

RestTemplate restTemplate = new RestTemplate();

ResponseEntity<String> response=null;

response=restTemplate.exchange(baseUrl,HttpMethod.GET, getHeaders(),String.class);

@ LoadBalanced is must in RestTemplate to work in service discovery.

**Client Side load balancing Ribbon**

Ribbon is a client-side load balancer that gives you a lot of control over the behavior of HTTP and TCP clients

You can configure some bits of a Ribbon client by using external properties in <client>.ribbon.\*, which is similar to using the Netflix APIs natively, except that you can use Spring Boot configuration files. The native options can be inspected as static fields in [CommonClientConfigKey](https://github.com/Netflix/ribbon/blob/master/ribbon-core/src/main/java/com/netflix/client/config/CommonClientConfigKey.java) (part of ribbon-core).

Spring Cloud also lets you take full control of the client by declaring additional configuration (on top of the RibbonClientConfiguration) using @RibbonClient, as shown in the following example:

**Name** should be same as **service name** in service registry

@Service  
@RibbonClient(name = **"book-server"**, configuration = RibbonConfig.**class**)  
**public class** BookServiceImpl **implements** BookService{

}

The following table shows the beans that Spring Cloud Netflix provides by default for Ribbon:

| **Bean Type** | **Bean Name** | **Class Name** |
| --- | --- | --- |
| IClientConfig | ribbonClientConfig | DefaultClientConfigImpl |
| IRule | ribbonRule | ZoneAvoidanceRule |
| IPing | ribbonPing | DummyPing |
| ServerList<Server> | ribbonServerList | ConfigurationBasedServerList |
| ServerListFilter<Server> | ribbonServerListFilter | ZonePreferenceServerListFilter |
| ILoadBalancer | ribbonLoadBalancer | ZoneAwareLoadBalancer |
| ServerListUpdater | ribbonServerListUpdater | PollingServerListUpdater |

**public class** RibbonConfig {  
 @Autowired  
 IClientConfig **ribbonClientConfig**;  
  
 @Bean  
 **public** IPing ribbonPing(IClientConfig config) {  
 **return new** PingUrl();  
 }  
  
 @Bean  
 **public** IRule ribbonRule(IClientConfig config) {  
 **return new** AvailabilityFilteringRule();  
 }  
}

**Default for All Ribbon Clients**

*@RibbonClients(defaultConfiguration = DefaultRibbonConfig.class)*

c.n.l.DynamicServerListLoadBalancer with eureka

* <clientName>.ribbon.NFLoadBalancerClassName: Should implement ILoadBalancer
* <clientName>.ribbon.NFLoadBalancerRuleClassName: Should implement IRule
* <clientName>.ribbon.NFLoadBalancerPingClassName: Should implement IPing
* <clientName>.ribbon.NIWSServerListClassName: Should implement ServerList
* <clientName>.ribbon.NIWSServerListFilterClassName: Should implement ServerListFilter

**ClientName** is **service name** in **service registry** : example book-server or book-inventory-server

**Using Ribbon with Eureka**

Ribbon.eureka.enabled= true

When Eureka is used in conjunction with Ribbon (that is, both are on the classpath), the ribbonServerList is overridden with an extension of DiscoveryEnabledNIWSServerList, which populates the list of servers from Eureka. It also replaces the IPing interface with NIWSDiscoveryPing, which delegates to Eureka to determine if a server is up.

**Ribbon Without Eureka**

Ribbon.eureka.enabled= false

Clientname.ribbon.listOfServers=google.com,yahoo.com

**Ribbon API Directly**

**@Autowired**

**private LoadBalancerClient loadBalancer;**

**public void getEmployee() throws RestClientException, IOException {**

**ServiceInstance serviceInstance=loadBalancer.choose("employee-producer");**

**String baseUrl=serviceInstance.getUri().toString();**

baseUrl=baseUrl+"/employee";

RestTemplate restTemplate = new RestTemplate();

ResponseEntity<String> response=restTemplate.exchange(baseUrl,HttpMethod.GET, getHeaders(),String.class);

IRule (com.netflix.loadbalancer)

AbstractLoadBalancerRule (com.netflix.loadbalancer)

ClientConfigEnabledRoundRobinRule (com.netflix.loadbalancer)

BestAvailableRule (com.netflix.loadbalancer)

PredicateBasedRule (com.netflix.loadbalancer)

ZoneAvoidanceRule (com.netflix.loadbalancer)

AvailabilityFilteringRule (com.netflix.loadbalancer)

RoundRobinRule (com.netflix.loadbalancer)

WeightedResponseTimeRule (com.netflix.loadbalancer)

ResponseTimeWeightedRule (com.netflix.loadbalancer)

RandomRule (com.netflix.loadbalancer)

RetryRule (com.netflix.loadbalancer)

**Feign**

[Feign](https://github.com/Netflix/feign) is a declarative web service client. It makes writing web service clients easier

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

*@EnableFeignClients=import FeignClientsRegistrar*

@FeignClient(value = **"book-server"**,fallback = BookServerFeignClient.BookServerFallback.**class**)  
**public interface** BookServerFeignClient {  
 @RequestMapping(method = RequestMethod.***GET***, value = **"/books/"**)  
 **public** String getBook();  
  
 @Component  
 **public static class** BookServerFallback **implements** BookServerFeignClient {  
 @Override  
 **public** String getBook() {  
 **return "None available!"**;  
 }  
 }  
}

So url will be //book-server/books/

@Autowired  
**private** BookServerFeignClient **bookServerFeignClient**;

**public void** getData(){  
 System.***out***.println(**bookServerFeignClient**.getBook());  
}

**Overriding Feign Defaults**

*@FeignClient(name = "stores", configuration = FooConfiguration.class)*

*@Configuration*

**public** **class** FooConfiguration {

*@Bean*

**public** Contract feignContract() {

**return** **new** feign.Contract.Default();

}

*@Bean*

**public** BasicAuthRequestInterceptor basicAuthRequestInterceptor() {

**return** **new** BasicAuthRequestInterceptor("user", "password");

}

}

**Feign request/response compression**

feign.compression.request.enabled=true

feign.compression.response.enabled=true

**Feign Hystrix Support**

If Hystrix is on the classpath and **feign.hystrix.enabled=true**, Feign will wrap all methods with a circuit breaker. Returning a com.netflix.hystrix.HystrixCommand is also available

If one needs access to the cause that made the fallback trigger, one can use the fallbackFactory attribute inside @FeignClient.

*@FeignClient(name = "hello", fallbackFactory = HystrixClientFallbackFactory.class)*

**protected** **interface** HystrixClient {

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

Hello iFailSometimes();

}

*@Component*

**static** **class** HystrixClientFallbackFactory **implements** FallbackFactory<HystrixClient> {

*@Override*

**public** HystrixClient create(Throwable cause) {

**return** **new** HystrixClient() {

*@Override*

**public** Hello iFailSometimes() {

**return** **new** Hello("fallback; reason was: " + cause.getMessage());

}

};

}

}

**Feign and @Primary**

When using Feign with Hystrix fallbacks, there are multiple beans in the ApplicationContext of the same type. This will cause @Autowired to not work because there isn’t exactly one bean, or one marked as primary.

*@FeignClient(name = "hello", primary = false)*

**public** **interface** HelloClient {

*// methods here*

}

**Circuit Breaker: Hystrix Clients**

Netflix has created a library called [Hystrix](https://github.com/Netflix/Hystrix) that implements the [circuit breaker pattern](https://martinfowler.com/bliki/CircuitBreaker.html). In a microservice architecture, it is common to have multiple layers of service calls.

A service failure in the lower level of services can cause cascading failure all the way up to the user. When calls to a particular service exceed circuitBreaker.requestVolumeThreshold (default: 20 requests) and the failure percentage is greater than circuitBreaker.errorThresholdPercentage (default: >50%) in a rolling window defined by metrics.rollingStats.timeInMilliseconds (default: 10 seconds), the circuit opens and the call is not made. In cases of error and an open circuit, a fallback can be provided by the developer.

<dependency>

<groupId>io.pivotal.spring.cloud</groupId>

<artifactId>spring-cloud-services-starter-circuit-breaker</artifactId>

</dependency>

@EnableCircuitBreaker extends EnableCircuitBreakerImportSelector

@EnableCircuitBreaker  
**public class** HystrixConfig {  
}

For non Feign client

@HystrixCommand(fallbackMethod = "getBackupGuide")

public String getGuide() {

return restTemplate.getForObject("http://company/available", String.class);

}

The fallback method must be in the same class and have the same method signature (i.e., have the same return type and accept the same parameters) as the annotated method

Hystrix Command work only a service method which is get called from restController or another service class.

If you extract method of service and then apply Hystrix will not work.

**Sleuth**

Spring Cloud Sleuth implements a distributed tracing solution for Spring Cloud, You can capture data simply in logs, or by sending it to a remote collector service(Zipkin).

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-sleuth</**artifactId**>  
</**dependency**>  
Example

applicationName, traceID,SpanId,zippkin export flag

[book-client,562dae71a8f05907,562dae71a8f05907,false]

[book-server,562dae71a8f05907,e50195a27702a7e9,false]

[book-inventory-server,562dae71a8f05907,5977afd96f15ea55,false]

* **Application name** – This is the name we set in the properties file and can be used to aggregate logs from multiple instances of the same application.
* **TraceId**– This is an id that is assigned to a single request, job, or action. Something like each unique user initiated web request will have its own *traceId*.
* **SpanId** – Tracks a unit of work. Think of a request that consists of multiple steps. Each step could have its own *spanId* and be tracked individually. By default, any application flow will start with same TraceId and SpanId.
* **Export** – This property is a boolean that indicates whether or not this log was exported to an aggregator like *Zipkin*. *Zipkin* is beyond the scope of this article but plays an important role in analyzing logs created by *Sleuth*.

@Autowired  
**private** Tracer **tracer**;

@GetMapping(value = **"/books"**)  
**public** String get() **throws** InterruptedException {  
 Span newSpan = **tracer**.nextSpan().name(**"newSpan"**).start();  
 **try** (Tracer.SpanInScope ws = **tracer**.withSpanInScope(newSpan.start())) {  
 Thread.*sleep*(1000L);  
 **LOG**.info(**"I'm in the new span doing some cool work that needs its own span"**);  
 } **finally** {  
 newSpan.finish();  
 }

}

**Zipkin**

[Zipkin](https://zipkin.io/) is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data.

http://localhost:9411/zipkin/

**Run Server**

java -DSTORAGE\_TYPE=elasticsearch -DES\_HOSTS=http://127.0.0.1:9200 -jar zipkin.jar

Download elasticserach6.7.x.zip extract and go to bin elasticserach.bat

OR

Create Zipkin server with the help SpringBoot @EnableZipkinServer

<dependency>

    <groupId>io.zipkin.java</groupId>

    <artifactId>zipkin-server</artifactId>

</dependency>

<dependency>

    <groupId>io.zipkin.java</groupId>

    <artifactId>zipkin-autoconfigure-ui</artifactId>

    <scope>runtime</scope>

</dependency>

**Client**

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-zipkin</**artifactId**>  
</**dependency**>

@Configuration  
**public class** ZipkinConfig {  
 @Bean  
 **public** Sampler defaultSampler() {  
 **return** Sampler.***ALWAYS\_SAMPLE***;  
 }  
}

**spring.zipkin.baseUrl**=**http://localhost:9411/  
spring.zipkin.enabled**=**true**

**In base url we can give zipkin server www.logserver.com**

**Zuul**

Routing is an integral part of a microservice architecture. For example, / may be mapped to your web application, /api/users is mapped to the user service and /api/shop is mapped to the shop service. [Zuul](https://github.com/Netflix/zuul) is a JVM-based router and server-side load balancer from Netflix.

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-zuul</**artifactId**>  
</**dependency**>

zuul:

ignoredServices: '\*'

routes:

users: /myusers/\*\*

In the preceding example, all services are ignored, **except** for users.

The preceding example means that HTTP calls to /myusers get forwarded to the users service (for example /myusers/101 is forwarded to /101).

Service id get resolved by service discovery if using eureka or ribbon for static servers.

zuul:

routes:

echo:

path: /myusers/\*\*

serviceId: myusers-service

myusers-service:

ribbon:

NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList

listOfServers: https://example1.com,http://example2.com

To add a prefix to all mappings, set zuul.prefix to a value, such as /api. The proxy prefix is stripped from the request before the request is forwarded by default (switch this behaviour off with zuul.stripPrefix=false)

**zuul.host.connect-timeout-millis**=**500000  
zuul.host.socket-timeout-millis**=**500000  
zuul.ignoredServices**=**"\*"  
zuul.routes.book-server.path**=**/api/book-service/\*\*  
zuul.routes.book-server.serviceId**=**book-server  
zuul.routes.book-server.stripPrefix**=**false  
zuul.routes.book-inventory-server.path**=**/api/book-deatils/\*\*  
zuul.routes.book-inventory-server.serviceId**=**book-inventory-server  
zuul.routes.book-inventory-server.stripPrefix**=**true**

If you are using @EnableZuulProxy with the Spring Boot Actuator you will enable (by default) an additional endpoint, available via HTTP as /routes. A GET to this endpoint will return a list of the mapped routes. A POST will force a refresh of the existing routes (e.g. in case there have been changes in the service catalog). You can disable this endpoint by setting endpoints.routes.enabled to false.

If you @EnableZuulProxy you can use the proxy paths to upload files and it should just work as long as the files are small. For large files there is an alternative path which bypasses the Spring DispatcherServlet (to avoid multipart processing) in "/zuul/\*". I.e. if zuul.routes.customers=/customers/\*\* then you can POST large files to "/zuul/customers/\*". The servlet path is externalized via zuul.servletPath. Extremely large files will also require elevated timeout settings if the proxy route takes you through a Ribbon load balancer, e.g.

**Filter**

* pre filters run before the request is routed.
* route filters can handle the actual routing of the request.
* post filters run after the request has been routed.
* error filters run if an error occurs in the course of handling the request.

**import static** org.springframework.cloud.netflix.zuul.filters.support.FilterConstants.***PRE\_TYPE***;  
  
**public class** PreFilter **extends** ZuulFilter {  
 @Override  
 **public** String filterType() {  
 **return *PRE\_TYPE***;  
 }  
  
 @Override  
 **public int** filterOrder() {  
 **return** 0;  
 }  
  
 @Override  
 **public boolean** shouldFilter() {  
 *//RequestContext context = getCurrentContext();  
 //Logic to know when to run filter* **return true**;  
 }  
  
 @Override  
 **public** Object run() **throws** ZuulException {  
 System.***out***.println(**"inside pre"**);  
 */\*RequestContext ctx = RequestContext.getCurrentContext();  
 HttpServletRequest request = ctx.getRequest();  
 if (request.getParameter("foo") != null) {  
 // put the serviceId in `RequestContext`  
 ctx.put(SERVICE\_ID\_KEY, request.getParameter("foo"));\*/* **return null**;  
 }  
}

**For Route**

Squareup OkHttpClient v3: Enabled by having the com.squareup.okhttp3:okhttp library on the classpath and setting ribbon.okhttp.enabled=true.

*@Autowired*

**private** ProxyRequestHelper helper;

@Override  
**public** Object run() **throws** ZuulException {  
 System.***out***.println(**"inside route filter"**);  
 OkHttpClient httpClient = **new** OkHttpClient.Builder()  
 .build();  
 Request.Builder builder = **new** Request.Builder()  
 .headers(headers.build())  
 .url(uri)  
 .method(method, requestBody);  
 Response response = httpClient.newCall(builder.build()).execute();  
 LinkedMultiValueMap<String, String> responseHeaders = **new** LinkedMultiValueMap<>();  
  
 **for** (Map.Entry<String, List<String>> entry : response.headers().toMultimap().entrySet()) {  
 responseHeaders.put(entry.getKey(), entry.getValue());  
 }  
  
 **this**.**helper**.setResponse(response.code(), response.body().byteStream(),  
 responseHeaders);  
 context.setRouteHost(**null**);  
 **return null**;  
}

in SpringBootApplication

@Bean  
**public** PreFilter preFilter() {  
 **return new** PreFilter();  
}

**Hystrix Fallbacks For Routes**

getRoute() which endpoint to apply in below it is apply for book-server. For all use “\*” in getRoute()

**public class** BookServerFallback **implements** FallbackProvider {  
  
  
 @Override  
 **public** String getRoute() {  
 **return "book-server"**;  
 }  
  
 @Override  
 **public** ClientHttpResponse fallbackResponse(**final** String route, **final** Throwable cause) {  
 **if** (cause **instanceof** HystrixTimeoutException) {  
 **return** response(HttpStatus.***GATEWAY\_TIMEOUT***);  
 } **else** {  
 **return** response(HttpStatus.***INTERNAL\_SERVER\_ERROR***);  
 }  
 }  
  
 **private** ClientHttpResponse response(**final** HttpStatus status) {  
 **return new** ClientHttpResponse() {  
 @Override  
 **public** HttpStatus getStatusCode() **throws** IOException {  
 **return** status;  
 }  
  
 @Override  
 **public int** getRawStatusCode() **throws** IOException {  
 **return** status.value();  
 }  
  
 @Override  
 **public** String getStatusText() **throws** IOException {  
 **return** status.getReasonPhrase();  
 }  
  
 @Override  
 **public void** close() {  
 }  
  
 @Override  
 **public** InputStream getBody() **throws** IOException {  
 **return new** ByteArrayInputStream(**"fallback"**.getBytes());  
 }  
  
 @Override  
 **public** HttpHeaders getHeaders() {  
 HttpHeaders headers = **new** HttpHeaders();  
 headers.setContentType(MediaType.***APPLICATION\_JSON***);  
 **return** headers;  
 }  
 };  
 }  
}

@Bean  
**public** BookServerFallback bookServerFallback(){  
 **return new** BookServerFallback();  
}