Filtered List of Servers

- Criteria by which you wish to limit the total list
- Spring Cloud default – Filter servers in the same *zone*

Ping

- Used to test if the server is up or down
- Spring Cloud default – delegate to Eureka to determine if server is up or down

Load Balancer

- The Load Balancer is the actual component that routes the calls to the servers in the filtered list
- Several strategies available, but they usually defer to a Rule component to make the actual decisions
- Spring Cloud's Default: ZoneAwareLoadBalancer

Rule

- The Rule is the single module of intelligence that makes the decisions on whether to call or not.
- Spring Cloud's Default: ZoneAvoidanceRule

Using Ribbon with Spring Cloud – part 3

- Low-level technique:
  - Access LoadBalancer, use directly:

```
public class MyClass {
   @Autowired LoadBalancerClient loadBalancer;

   public void doStuff() {
      ServiceInstance instance = loadBalancer.choose("subject");
      URI subjectUri = URI.create(String.format("http://%s:%s",
         instance.getHost(), instance.getPort()));
      // ... do something with the URI
   }
}
```

"subject" - An example of a "client-id"

Customizing

- Previously we described the defaults. What if you want to change them?
- Declare a separate config with replacement bean.

```
@Configuration
@RibbonClient(name="subject", configuration=SubjectConfig.class)
public class MainConfig {
}
```

```
@Configuration
public class SubjectConfig {
   @Bean
   public IPing ribbonPing(IClientConfig config) {
      return new PingUrl();
   }
}
```

Replaces the "Ping" strategy
Employed when calling "subject" clients

Note: Do NOT component scan
SubjectConfig!

Feign

- What is it?
  - Declarative REST client, from NetFlix
  - Allows you to write calls to REST services with no implementation code
  - Alternative to RestTemplate (even easier!)
  - Spring Cloud provides easy wrapper for using Feign

Spring REST Template

- Spring's Rest Template provides very easy way to call REST services

```
RestTemplate template = new RestTemplate();
String url = "http://inventoryService/{0}";
Sku sku = template.getForObject(url,  Sku.class, 4724352);
```

Instantiate
(or dependency inject)

Provide target URL
(note the placeholder)

Call the URL, provide expected class,
Provide value for placeholder.
Template takes care of all HTTP
and type conversion!

- Still, this code must be
  1) Written
  2) Unit-tested with mocks / stubs.

Feign Alternative –
Declarative Web Service Clients

- How does it work?
  - Define *interfaces* for your REST client code
  - Annotate interface with Feign annotation
  - Annotate methods with Spring MVC annotations
    - Other implementations like JAX/RS pluggable
- Spring Cloud will implement it at run-time
  - Scans for interfaces
  - Automatically implements code to call REST service and process response

Feign Interface

- Create an *Interface*, not a Class:

Marks interface to be implemented by Feign / Spring

```
@FeignClient(url="localhost:8080/warehouse")
public interface InventoryClient {

  @RequestMapping(method = RequestMethod.GET,
                  value = "/inventory")
  List<Item> getItems();

  @RequestMapping(method = RequestMethod.POST,
                  value = "/inventory/{sku}",
                  consumes = "application/json")
  void update(@PathVariable("sku") Long sku, @RequestBody Item item);
}
```

Base URL
(we'll replace this later with Eureka client ID!)

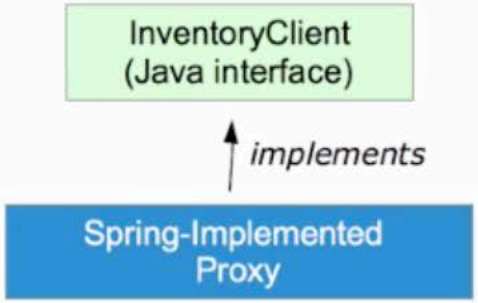
Ordinary, Spring MVC mapping metadata

Note: No extra dependencies are needed for Feign when using Spring Cloud.

What does @EnableFeignClients do?

- Before startup

- After startup



@EnableFeignClients

- You can @Autowire an InventoryClient wherever one is needed

Ribbon and Eureka

Where do they fit in?

- The previous example - hard-coded URL:

```
@FeignClient(url="localhost:8080/warehouse")
```

- ...use a Eureka “Client ID” instead:

```
@FeignClient("warehouse")
```

- Ribbon is automatically enabled
  - Eureka gives our application all “Clients” that match the given Client ID
  - Ribbon automatically applies load balancing
  - Feign handles the code.

Ribbon and Eureka

Where do they fit in?

- The previous example - hard-coded URL:

```
@FeignClient(url="localhost:8080/warehouse")
```

- ...use a Eureka “Client ID” instead:

```
@FeignClient("warehouse")
```

- Ribbon is automatically enabled
  - Eureka gives our application all “Clients” that match the given Client ID
  - Ribbon automatically applies load balancing
  - Feign handles the code.

Comparison with Physical Circuit Breaker

HYSTRIX DEFEND YOUR APP	
• "closed" when operating normally	• "closed" when operating normally
• "open" when failure is detected	• "open" when failure is detected
• Failure definition flexible – Exception thrown or timeout exceeded over time period	• Failure definition fixed – current flow exceeds amp rating.
• Definable "Fallback" option	• No fallback
• Automatically re-closes itself	• Must be closed manually

The Circuit Breaker Pattern

- Consider a household circuit breaker
  - It “watches” a circuit
  - When failure occurs (too much current flow), it “opens” the circuit (disconnects the circuit)
  - Once problem is resolved, you can manually “close” the breaker by flipping the switch.
  - Prevents cascade failure
    - i.e. - your house burning down.

Hystrix – The Software Circuit Breaker

- Hystrix – Part of Netflix OSS
- Light, easy-to-use wrapper provided by Spring Cloud.
- Detects failure conditions and “opens” to disallows further calls
  - Hystrix Default – 20 failures in 5 seconds
- Identify “fallback” - what to do in case of a service dependency failure
  - Think: catch block, but more sophisticated
  - Fallbacks can be chained
- Automatically “closes” itself after interval
  - Hystrix Default – 5 seconds.

Comparison with Physical Circuit Breaker

HYSTRIX DEFEND YOUR APP	
• "closed" when operating normally	• "closed" when operating normally
• "open" when failure is detected	• "open" when failure is detected
• Failure definition flexible – Exception thrown or timeout exceeded over time period	• Failure definition fixed – current flow exceeds amp rating.
• Definable "Fallback" option	• No fallback
• Automatically re-closes itself	• Must be closed manually

Hystrix (Spring Cloud) Setup

- Add the Dependency:

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

- Enable Hystrix within a configuration class:

```
@SpringBootApplication
@EnableHystrix
public class Application {
}
```

Hystrix (Spring Cloud) Example

- Use the @HystrixCommand to wrap methods in a circuit breaker:

Based on recent failures,
Hystrix will call
one of these two methods

```
@Component
public class StoreIntegration {

   @HystrixCommand(fallbackMethod = "defaultStores")
   public Object getStores(Map<String, Object> parameters) {
      //do stuff that might fail
   }

   public Object defaultStores(Map<String, Object> parameters) {
      return /* something useful */;
   }
}
```

Hystrix (Spring Cloud) Example

- Use the @HystrixCommand to wrap methods in a circuit breaker:

Based on recent failures,
Hystrix will call
one of these two methods

```
@Component
public class StoreIntegration {

   @HystrixCommand(fallbackMethod = "defaultStores")
   public Object getStores(Map<String, Object> parameters) {
      //do stuff that might fail
   }

   public Object defaultStores(Map<String, Object> parameters) {
      return /* something useful */;
   }
}
```

Custom Properties

- Failure / Recovery behavior highly customizeable
  - Use commandProperties and @HystrixProperty

```
@HystrixCommand(
 fallbackMethod = "defaultStores",
 commandProperties = {
  @HystrixProperty(
   name="circuitBreaker.errorThresholdPercentage", value="20"),
  @HystrixProperty(
   name="circuitBreaker.sleepWindowInMilliseconds", value="1000")
  })
 public Object yourMethod( ... ) {
  // ...
 }
```

Over 20% failure rate in 10 second period, open breaker

After 1 second, try closing breaker

Command can be called various ways

- **Synchronously** – call execute and block thread (default behavior).

- **Asynchronously** – Call in a separate thread (queue), returning a future. Deal with `Future` when you want
  - Just like Spring's @Async annotation

- **Reactively** – Subscribe, get a listener (`Observable`)

Hystrix – The Software Circuit Breaker

- Hystrix – Part of Netflix OSS
- Light, easy-to-use wrapper provided by Spring Cloud.
- Detects failure conditions and “opens” to disallows further calls
  - Hystrix Default – 20 failures in 5 seconds
- Identify “fallback” - what to do in case of a service dependency failure
  - Think: catch block, but more sophisticated
  - Fallbacks can be chained
- Automatically “closes” itself after interval
  - Hystrix Default – 5 seconds.

HYSTRIX
DEFEND YOUR APP

Custom Properties

- Failure / Recovery behavior highly customizeable
  - Use commandProperties and @HystrixProperty

```
@HystrixCommand(
 fallbackMethod = "defaultStores",
 commandProperties = {
  @HystrixProperty(
   name="circuitBreaker.errorThresholdPercentage", value="20"),
  @HystrixProperty(
   name="circuitBreaker.sleepWindowInMilliseconds", value="1000")
  })
 public Object yourMethod( ... ) {
  // ...
 }
```

Over 20% failure rate in 10 second period, open breaker

After 1 second, try closing breaker

Example: Asynchronous Command Execution

- Have method return Future
  - Wrap result in `AsyncResult`

```
@HystrixCommand( ... )
public Future<Store> getStores(Map<String, Object> parameters) {
   return new AsyncResult<Store>() {
      @Override
      public Store invoke() {
         //do stuff that might fail
      }
   };
}
```

Example: Reactive Command Execution

- Have method return Observable
  - Wrap result in `ObservableResult`

```
@HystrixCommand( ... )
public Observable<Store> getStores(Map<String, Object> parameters) {
   return new ObservableResult<Store>() {
      @Override
      public Store invoke() {
         //do stuff that might fail
      }
   };
}
```

Hystrix Properties

- *execution.isolation.thread.**timeout**InMilliseconds*

  How long should we wait for success?
- *circuitBreaker.requestVolumeThreshold*

  # of requests in rolling time window (10 seconds) that activate the circuit breaker (NOT the # of errors that will trip the breaker!)
- *circuitBreaker.errorThresholdPercentage*

  % of failed requests that will trip the breaker (default = 50%)
- *metrics.rollingStats.timeInMilliseconds*

  Size of the rolling time window (default = 10 seconds)

How to Reset the Circuit Breaker?

- When the failing service is healthy, we want to 'close' the circuit breaker again
- *circuitBreaker.sleepWindowInMilliseconds*
  - How long to wait before closing the breaker (default = 5 seconds)
- *circuitBreaker.forceClosed*
  - Manually force the circuit breaker closed

Hystrix Dashboard

- Hystrix provides a built-in dashboard to check the status of the circuit breakers: