Sistemas Distribuídos

2018/2019

Aula 4 ¶

- Tratamento de falhas
  - Falhas de comunicação em REST
  - Falhas de comunicação em SOAP

- Exercícios

Falhas nas invocações remotas

As invocações REST e SOAP podem falhar por diversas razões.

Algumas razões frequentes são:

- Ocorreu um problema de comunicação: a ligação foi quebrada ou está muito lenta;
- Os servidores (ainda) não estão a correr, estão muito lentos ou falharam;

Tratamento de erros

Se as causas da falha de uma invocação remota forem **temporárias**...

É possível mascarar (ie., esconder) o problema **repetindo** a invocação até esta suceder.

Mascarar falhas de comunicação em REST

As invocações REST sem sucesso, devido a falhas na rede ou no servidor, são expostas ao cliente sob a forma de excepções: javax.ws.rs.ProcessingException
Estas excepções podem ser apanhadas e tratadas...

```
final int RETRY_PERIOD = 1000;
ClientConfig config = new ClientConfig();
Client client = ClientBuilder.newClient(config);
URI baseURI = UriBuilder.fromUri("http://localhost:9999/rest/media").build();
WebTarget target = client.target( baseURI );
for(;;)
    try {
        Response r = target.request()
            .accept(MediaType.APPLICATION_JSON)
            .post(Entity.entity( new byte[1024], MediaType.APPLICATION_OCTET_STREAM));

        if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() )
            System.out.println("Response: " + r.readEntity(String.class ) );
        else
            System.out.println("Status: " + r.getStatus() );

        break;
    } catch( ProcessingException pe ) {
        Thread.sleep( RETRY_PERIOD );
    }
```

Mascarar falhas de comunicação em SOAP

No caso dos *Web*Service SOAP, as invocações sem sucesso devido a falhas na rede ou no servidor, lançam excepções do tipo:

```
javax.xml.ws.WebServiceException
```

```
final int RETRY_PERIOD = 1000;

QName QNAME = new QName(SoapMedia.NAMESPACE, SoapMedia.NAME);

for(;;)
    try {
        Service service = Service.create( new URL(serverUrl + WSDL), QNAME);
        SoapMedia media = service.getPort( microgram.api.soap.SoapMedia.class );

        byte[] bytes = Files.readAllBytes( EARTH.toPath() );
        String uri = media.upload( bytes );
        Log.info("Upload completed: " + uri );
        break;
    } catch( MicrogramException x ) {
        Log.info("Upload failed, reason: "  + x.getMessage());
        break;
    } catch( WebServiceException ws ) {
        Log.info("IO error, reason: "  + ws.getMessage());
        Thread.sleep( RETRY_PERIOD);
    }
}
```

Gestão de timeouts

As invocações remotas podem falhar após expirar um *timeout* bastante longo (minutos) na criação da ligação TCP/HTTP com o servidor.

É possível controlar e reduzir esse *timeout* na criação da ligação, bem como desistir mais cedo da invocação, quando o servidor tarda em responder.

Isso, permitirá ao cliente tentar outro servidor, por exemplo...

Controlar timeouts em REST

```
final int CONNECT_TIMEOUT = 2000;
final int READ_TIMEOUT = 5000;

ClientConfig config = new ClientConfig();
config.property(ClientProperties.CONNECT_TIMEOUT, CONNECT_TIMEOUT);
config.property(ClientProperties.READ_TIMEOUT, READ_TIMEOUT);
```

Controlar timeouts em SOAP

```
import javax.xml.ws.BindingProvider;
import com.sun.xml.ws.client.BindingProviderProperties;

final int SOAP_CONN_TIMEOUT = 2000;
final int SOAP_RECV_TIMEOUT = 5000;

QName QNAME = new QName(SoapMedia.NAMESPACE, SoapMedia.NAME);
Service service = Service.create(new URL(serverUrl + WSDL), QNAME);
SoapMedia media = service.getPort(microgram.api.soap.SoapMedia.class);

((BindingProvider) media).getRequestContext().put(BindingProviderProperties.REQUEST_TIMEOUT,
SOAP_RECV_TIMEOUT);
((BindingProvider) media).getRequestContext().put(BindingProviderProperties.CONNECT_TIMEOUT,
SOAP_CONN_TIMEOUT);
```

Nota: A interface BindingProviderProperties não está disponível em todas as distribuições. Isso soluciona-se juntando a seguinte dependência ao pom.xml

```
<groupId>com.sun.xml.ws</groupId>
    <artifactId>jaxws-rt</artifactId>
    <version>2.3.2</version>
    <type>pom</type>
</dependency>
```

Exercício

O projeto P4-Falhas/SD2019-Labs-P4, incluído no repositório, contém os os serviços MediaStorage, implementados em REST e SOAP, à custa da classe JavaMedia, de acordo com o exercício suplementar da aula anterior. No entanto, as operações da classe JavaMedia foram alteradas e o seu tempo de execução varia aleatoriamente.

Analise o código fornecido.

1. Altere os clientes REST fornecidos, de modo a mascarar as falhas nas invocações.
   - Coloque a invocação num ciclo, repetindo até está suceder, esperando algum tempo entre tentativas.
2. Faça o mesmo, agora para os clientes SOAP fornecidos.
3. Teste os clientes, lançando o cliente antes do servidor;

Exercício Suplementar 1

O exercício suplementar da aula anterior teve como objetivo evitar duplicar código quando o mesmo serviço é implementado usando as duas tecnologias: REST e SOAP. Para tal, a parte em comum foi colocada numa classe à parte - a classe JavaMedia.

Do lado do cliente, quando o mesmo serviço existe em mais do que uma versão e em execução ao mesmo tempo, a natureza concreta do servidor que está a ser utilizado pode ser escondida ao resto da aplicação, de modo a evitar (em todas as operações invocadas) código cliente do estilo:

```
if (server.isRest() ) {
   //do client rest invocation...
} else if( server.isSoap() ) {
   //do client soap invocation...
} else {
   // unknown server type...
}
```

A solução passa por criar uma fábrica de clientes do serviço que, com base no URI, retorna um cliente REST ou um cliente SOAP. Para tal, os dois tipos de clientes terão que implementar a interface genérica do serviço, à custa das respetivas tecnologias.

O projeto P4-Falhas/SD2019-Labs-P4-Sup1, foi preparado para ilustrar a abordagem anterior, usando o serviço MediaStorage como exemplo. (*)

Analise o código fornecido, em particular:

- A classe MediaClientFactory

  Representa a fábrica de clientes. Repare que os clientes retornados são genéricos e implementam a interface genérica do serviço Media, de modo a não expor detalhes REST ou SOAP ao resto da aplicação.
- As classes RestMediaClient e SoapMediaClient

  Fornecem a implementação da interface genérica do serviço, respetivamente à custa das tecnologias REST e SOAP.
- A classe MediaUploader

  Cliente genérico do serviço MediaStorage. Utiliza o serviço de descoberta para encontrar um servidor, seja ele REST ou SOAP. Recorre à fábrica de clientes para obter a interface genérica do serviço, com a qual faz *upload* da imagem, alheio à natureza concreta do servidor.

Complete o código fornecido:

1. Implemente o resto das operações nos clientes REST e SOAP fornecidos.
2. Crie um cliente genérico `MediaDownloader`;
   - Não se esqueça substituir a `class Discovery`, incluída no projeto, pela a versão que implementou na primeira aula.
3. Teste os clientes genéricos, lançando um só servidor REST ou SOAP, ou os dois em simultâneo.

(*) O projeto depende do projeto do exercício anterior. Confira isso nas propriedades do projeto, em: Java Build Path > Projects.

Exercício Suplementar 2

O objetivo deste exercício é explorar mais oportunidades para fatorizar e evitar código repetido, entre operações de um **mesmo** serviço, e inclusivamente, entre clientes de serviços **diferentes**, mas de igual tecnologia. Ou seja, existem partes comuns nas operações de um cliente REST e entre clientes REST, independentemente da interface do serviço, que podem ser partilhadas. O mesmo se passa com os clientes SOAP.

Em particular, este exercício aplica-se ao tratamento de falhas nas invocações e no processamento de erros e resultados.

O projeto P4-Falhas/SD2019-Labs-P4-Sup2, destina-se a suportar este exercício (e depende dos anteriores).

Analise o código do projeto, prestando atenção ao seguinte:

1. Partilhar código entre clientes de serviços diferentes:

   As classes abstratas RestClient e SoapClient (já parcialmente utilizadas no projeto anterior), irão conter código comum a todos os clientes REST e SOAP, respetivamente.

   No caso do REST, a classe abstrata RestClient guarda:

   - Dados como: `ClientConfig`, `Client` e um `WebTarget`, parcialmente preenchidos.
   - Código de conversão de erros específicos REST (StatusCodes HTTP) para erros genéricos da classe Result.

2. Partilhar código entre operações de um dado serviço:

   No tratamento de falhas de invocação, a estratégia pode ser repetir a invocação, num ciclo, até esta suceder (ou até haver motivos para desistir).

   Este comportamento pode ser fatorizado entre operações, evitando codificar o ciclo de repetição;

   A super classe abstrata RetryClient, da qual deriva RestClient, exemplifica uma forma genérica de fazer o tratamento das falhas por repetição do ciclo.

   Para tal, na classe RetryClient existem duas versões do método `reTry` para tratar, respetivamente, as invocações remotas cujo resultado é `void` e aquelas que retornam um resultado, do tipo genérico *T*.

   O uso do método `reTry` está exemplificado no método `upload` da classe RestMediaClient. (Para tornar a invocação concisa, é usada uma expressão lambda. Tal é possível, pois o argumento do método `reTry` é uma interface com apenas um método.)

   Os métodos `responseContents` e `verifyResponse` da classe RestClient implementam a mesma ideia de fatorização, mas relativamente ao processamento do resultado das invocações. Verificando se a resposta obtida sucedeu, ou caso houve um erro e é necessário produzir uma `WebApplicationException`, com o status code correspondente.

Os mesmos princípios podem ser adaptados ao caso dos clientes de serviços SOAP.

Complete o código fornecido:

1. Implemente o resto das operações nos clientes REST e SOAP fornecidos;
2. Adicione a operação de **DELETE** às duas versões do serviço.

Desafio

O código fornecido para ajudar a realizar o trabalho prático está organisado segundo os princípios descritos nos exercícios suplementares desta aula.

Aplique estes princípios na resolução do trabalho prático. ;-)