DISTRIBUTED SYSTEMS

Lab 5

João Leitão, Sérgio Duarte, Pedro Camponês

GOALS

In the end of this lab you should be able to:

- Understand what is required for the first project.
- Understand some of the semantics of the operations.
- Know how to use the automated tester to test your solution.

GOALS

In the end of this lab you should be able to:

- **Understand what is required for the first project.**
- **Understand some of the semantics of the operations.**
- Know how to use the automated tester to test your solution.

PRIMEIRO PROJECTO

Neste projeto vão desenvolver um Sistema distribuído composto por vários serviços (que correm em servidores independents) que modela uma versão simplicada (e diferente) do Sistema Reddit!

Arquitetura do Sistema

Users Service

Manipula entidades do tipo **User**

Operações:

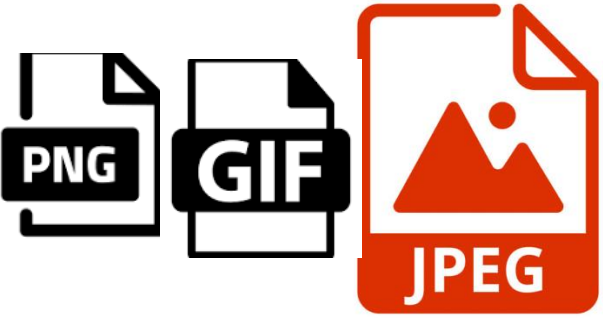
- Criar utilizador
- Obter informação de um utilizador
- Atualizar um utilizador
- Apagar um utilizador
- Procurar um utilizador

Notas relevantes:

- Tem a sua propria base de dados a que os outros serviços não têm acesso.
- A operação de apagar um utilizador têm implicações nos restantes serviços.

Arquitetura do Sistema

Image Service



Manipula **imagens (e a sua meta informação)**

Operações:

- Criar uma imagem (associada a um utilizador)
- Obter uma imagem
- Apagar uma imagem (apenas o dono da imagem pode apagar)

Notas relevantes:

- As imagens devem ser persistidas (mas talvez não numa base de dados).
- É preciso de alguma forma manter informação relativa aos donos de cada imagem.

Arquitetura do Sistema

Content Service

Manipula entidades do tipo **Post**

Notas relevantes:

- Posts quando são criados podem ou não estar associado a outro post (dependendo se é um post de topo ou uma resposta a um post ou outra reposta)

Operações:

- Criar um post.
- Obter um post.
- Listar posts (com potenciais ordens diferentes).
- Listar posts que respondem a um post especifico.
- Apagar um post.
- Adicionar/Remover upvotes/downvotes
- Consultar upvotes/downvotes

Arquitetura do Sistema

Content Service

Manipula entidades do tipo **Post**

Notas relevantes:

- Upvotes e Downvotes não podem ser repetidos (por utilizador) pelo que um Contador apenas não consegue lidar com esta semântica.

Operações:

- Criar um post.
- Obter um post.
- Listar posts (com potenciais ordens diferentes).
- Listar posts que respondem a um post especifico.
- Apagar um post.
- Adicionar/Remover upvotes/downvotes
- Consultar upvotes/downvotes

Arquitetura do Sistema

Content Service

Manipula entidades do tipo **Post**

Notas relevantes:

- A edição de um post (por um utilizador) nunca deve manipular certos atributos do post: e.g., postId, authorId, creationTimestamp, parentUrl, upVote, downVote.

Operações:

- Criar um post.
- Obter um post.
- Listar posts (com potenciais ordens diferentes).
- Listar posts que respondem a um post especifico.
- Apagar um post.
- Adicionar/Remover upvotes/downvotes
- Consultar upvotes/downvotes

Arquitetura do Sistema

Content Service

Manipula entidades do tipo **Post**

Notas relevantes:

- Um post não pode ser editado (nem pelo seu autor) quando alguma outra entidade o referencia (outro post, upVote ou downVote)

Operações:

- Criar um post.
- Obter um post.
- Listar posts (com potenciais ordens diferentes).
- Listar posts que respondem a um post especifico.
- Apagar um post.
- Adicionar/Remover upvotes/downvotes
- Consultar upvotes/downvotes

ARQUITETURA DO SISTEMA

Content Service

Manipula entidades do tipo **Post**

Notas relevantes:

- Atenção ao parametro opcional desta operação.

Listar posts que respondem a um post especifico.

- Esta operação têm um parametro opcional chamado timeout, que serve para suportar uma funcionalidade valorativa do projeto.
- Se o timeout estiver definido, esta operação deve bloquer até que uma nova resposta ao post apareça depois de operação ser recebida pelo servidor (pelo tempo máximo do timeout).

ARQUITETURA DO SISTEMA

Content Service

Manipula entidades do tipo **Post**

Notas relevantes:

- Atenção ao parametro opcional desta operação.

Listar posts que respondem a um post especifico.

- Esta operação têm um parametro opcional chamado timeout, que serve para suportar uma funcionalidade valorativa do projeto.
- Se o timeout estiver definido, esta operação deve bloquer até que uma nova resposta ao post apareça depois de operação ser recebida pelo servidor (pelo tempo máximo do timeout).

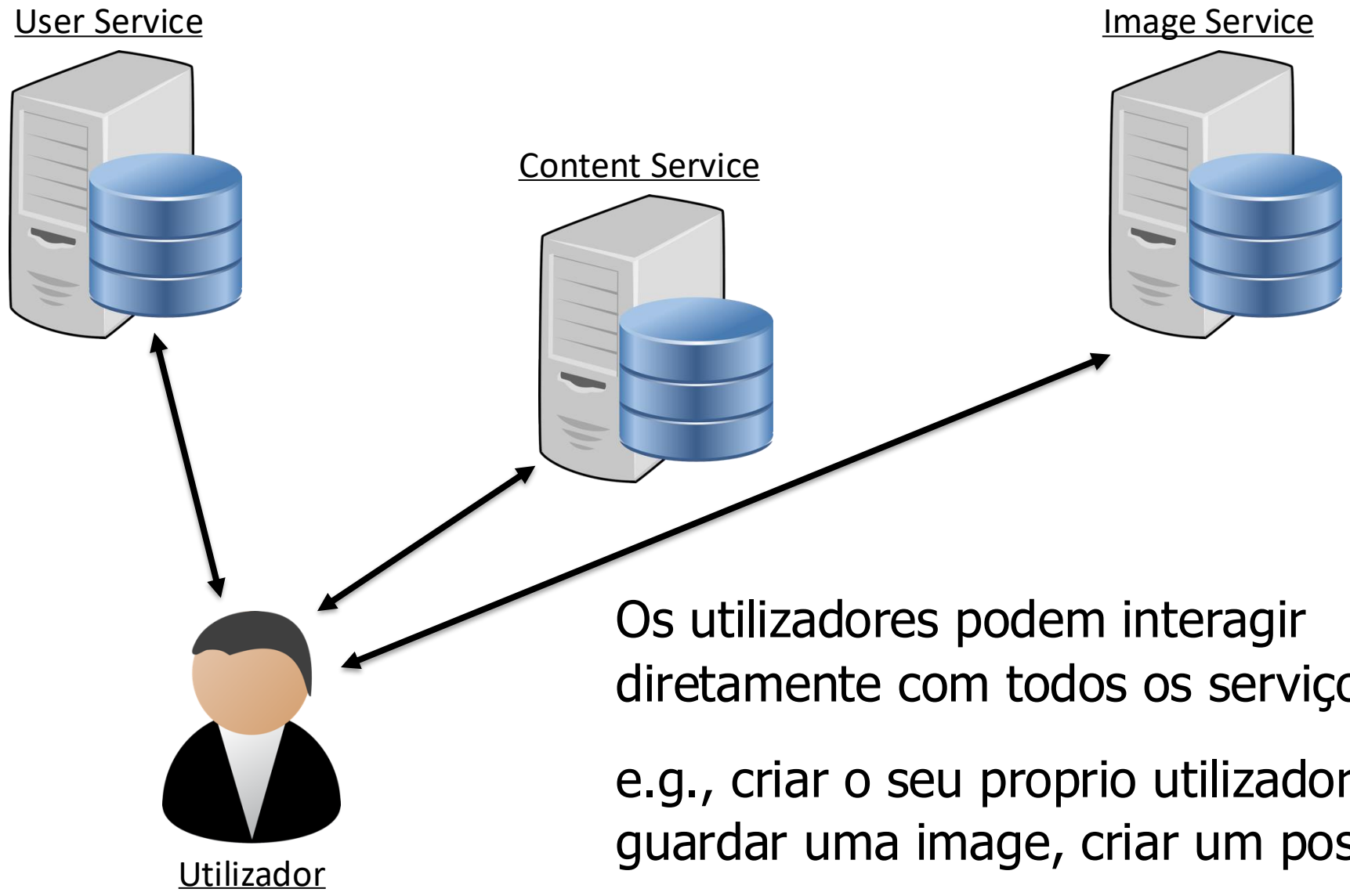
ARQUITETURA DO SISTEMA

User Service

Content Service

Image Service

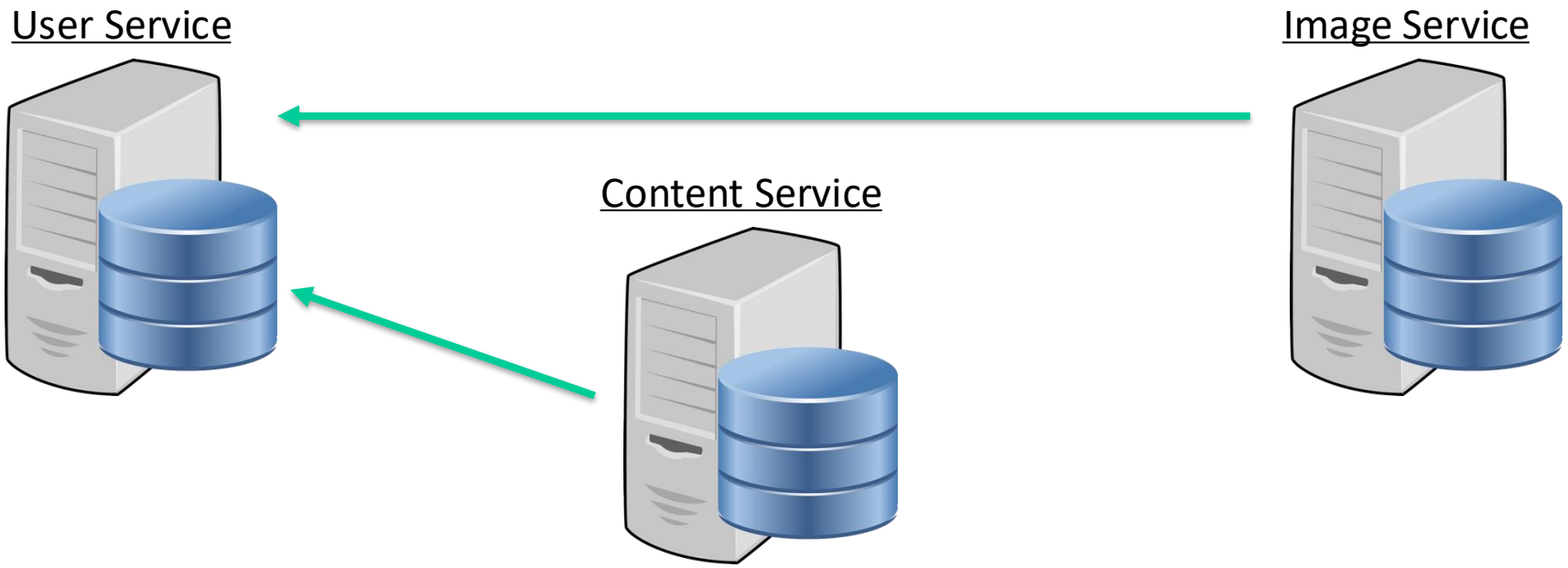
Interações no Sistema: Clientes



Os utilizadores podem interagir diretamente com todos os serviços:

e.g., criar o seu proprio utilizador, guardar uma image, criar um post.

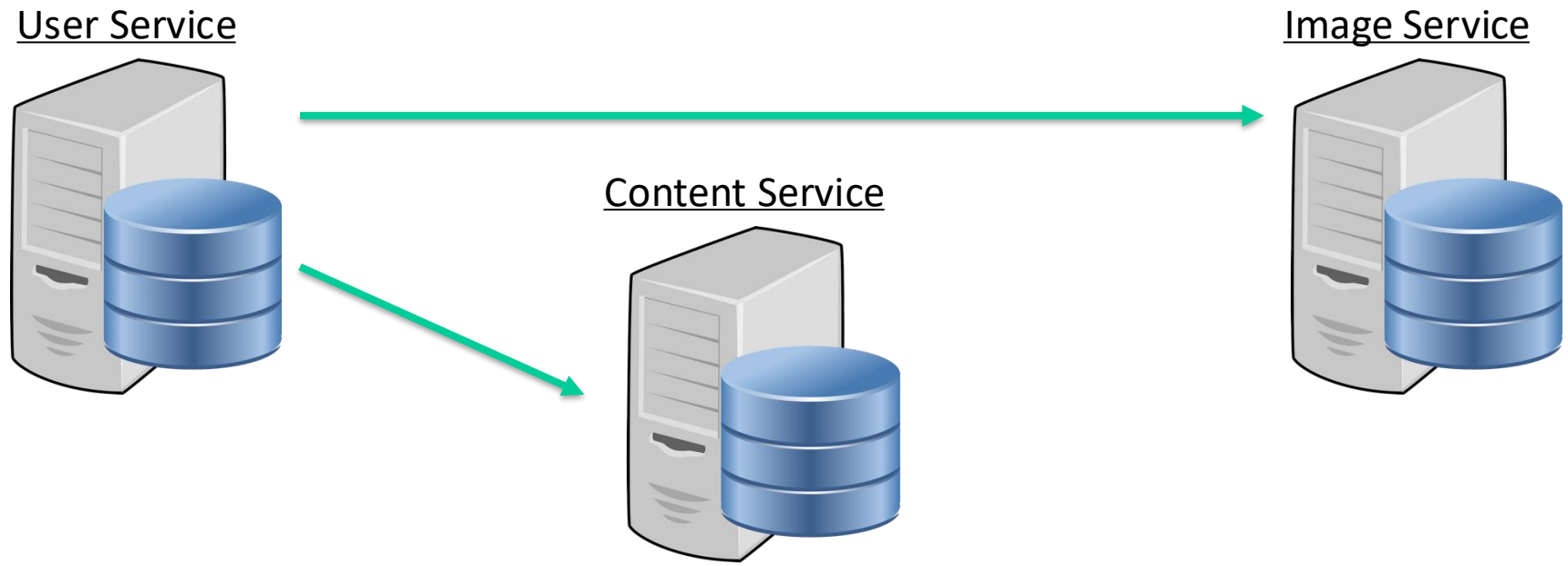
Interações no Sistema: Entre serviços



Os servidores também vão ter de interagir diretamente entre si:

O Image Service e o Content Service vão, por exemplo, ter de autenticar utilizadores, sendo o Users Service o único que pode suportar essa operação.

Interações no Sistema: Entre serviços

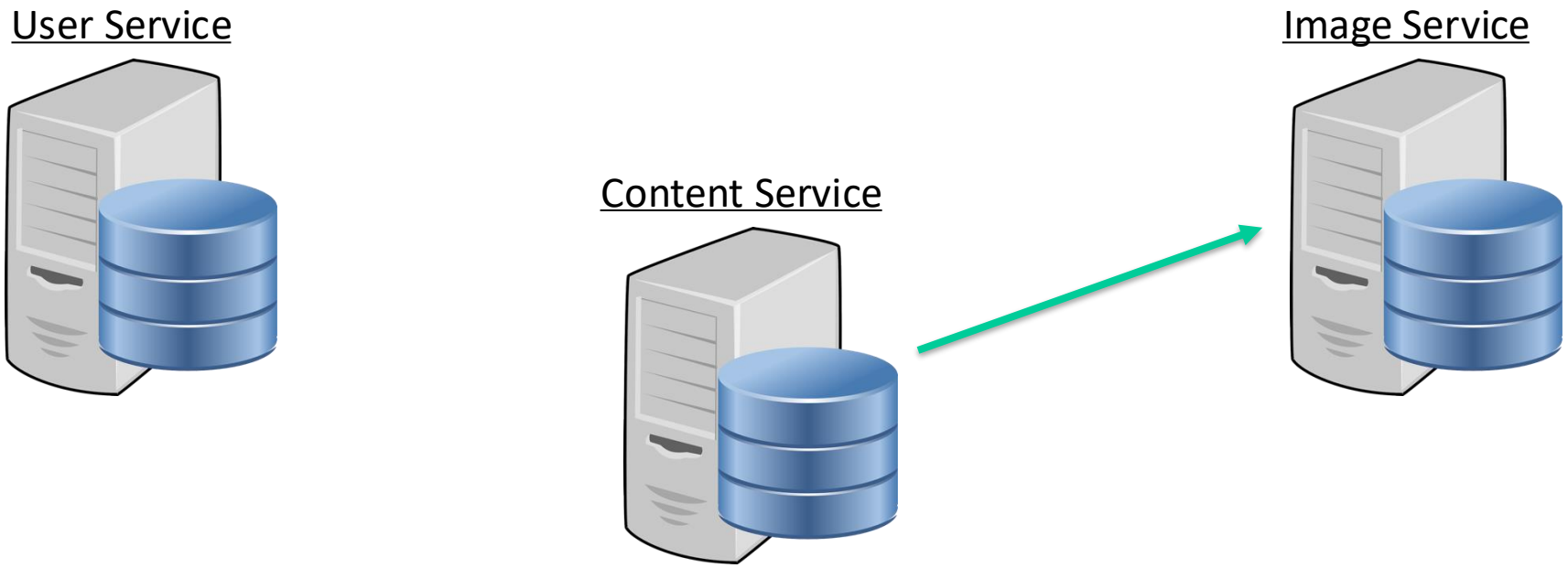


Os servidores também vão ter de interagir diretamente entre si:

O User Service vai ter de interagir com o Content Service e o Image Service quando um utilizadore é apagado:

- Os posts desse utilizador continuam a existir mas deixam de estar associados a um utilizador.
- Os upvotes/downvotes desse utilizador devem deixar de existir.
- Se o utilizador tinha uma imagem como avatar, essa imagem deve ser apagada.

Interações no Sistema: Entre serviços



Os servidores também vão ter de interagir diretamente entre si:

O Content Service deve interagir com o Image Service:

- Quando um post que refere um conteudo multimedia no Image Service é apagado (de forma a apagar essa imagem também).

Interações Com Base de Dados

- A API que vos foi fornecida do Hibernate executa todas as operações individualmente numa transação.
- Isso pode não ser util visto que permite accessos concorrentes disruptivos.
- E.g.; duas operações de edição de um User concorrentes que modificam o mesmo objeto (mas campos diferentes)
- Op1-Read, Op2-Read, Op1-Update(password),
  Op2-Update(e-mail).
- Na execução acima a mudança de password pode-se perder.
- Para resolver isto precisam de executar leituras e escritas de uma operação no contexto de uma única transação.

Interações Com Base de Dados

- A API que vos foi fornecida do Hibernate executa todas as operações individualmente numa transação.
- Isso pode não ser util visto que permite accessos concorrentes disruptivos.
- E.g.; duas operações de edição de um User concorrentes que modificam o mesmo objeto (mas campos diferentes)
- Op1-Read, Op2-Read, Op1-Update(password), Op2-Update(e-mail).
- N…rder.
- P…s de u…

Para conseguirem isto vão ter de adicionar novas operações à classe de suporte do Hibernate, nomeadamente operações para começar uma transação (e obter a referência para a mesma), ler, modificar e apagar entidades no contexto dessa transação, e fazer commit a uma transação.

REGRAS

- Interfaces REST/gRPC fornecidas no Código base no clip.
- Não podem modificar as Interfaces excepto para:
  - Adicionar parametros opcionias a funções.
  - Adicionar novos endpoints para suportar funcionalidades pedidas.
- Podem utilizar **e adaptar** todo o código que foi fornecido ao longo das aulas práticas.
- Os diferentes serviços vão ser executados em containers distintos, pelo que não podem assumir, por exemplo, que a lógica do serviço Content pode consultar diretamente a base de dados do serviço de Users.

GOALS

In the end of this lab you should be able to:

- Understand what is required for the first project.
- Understand some of the semantics of the operations.
- **Know how to use the automated tester to test your solution.**

TESTER:

O Tester já se encontra disponível, e vai ser evoluído para cobrir todas as funcionalidades do enunciado.

Página do tester: https://smduarte.github.io/sd2425-tester

- O tester funciona criando uma ou mais instâncias da imagem docker do vosso código e executando operações sobre os vários servidores (e verificando o retorno de operações).
- O tester requer que todos os servidores possam ser descobertos pelo mecanismo de multicast (Discovery)
- Como qualquer bom processo de testes, alguns testes executam operações que devem ter sucesso e outras geram operações que devem falhar (e verifica os erros retornados).

TESTER (COMO EXECUTAR):

A página do tester descreve como executar o tester. Na generalidade devem usar os scripts fornecidos com a API do Código (zip no clip) `test-sd-tp1.sh` e `test-sd-tb2.bat`

A forma mais simples de executar o tester é (em linux/mac)

- `sh test-sd-tp1.sh -image <node-da-imagem-docker>`

Opções adicionais:

- -log OFF|ALL|FINE : mostra informação adicional nos logs.
- -sleep <valor em segundos> : modifica os tempos de espera entre operações do servidor (default é 15 segundos).
- -test <numero do teste> : salta os testes até chegar ao teste solicitado.s

TESTER (EXEMPLO DE EXECUÇÃO):

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
2a )     Testing Discovery Service [ listening for announcements... ]
----------------------------------------------------------------
Starting: Users : [1] REST Servers, [0] GRPC Servers
Started: users-1 : java --add-opens java.base/java.lang=ALL-UNNAMED -cp /home/sd/* fctreddit.impl.server.rest.Us
ersServer
fa584049ea1a92e3928db5ecb0b2d60f3ae24a6297de7383766e97d7d5b1cf41
Discovery started @ /226.226.226.226:2266
INFO: Users     http://172.20.0.3:8080/rest

Found: Users
OK
killByImage(sd2425-tp1-api-jleitao): Removing container: [/users-1]
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
3a )     Testing Users Service [ createUser ]
----------------------------------------------------------------
Starting: Users : [1] REST Servers, [0] GRPC Servers
Started: users-1 : java --add-opens java.base/java.lang=ALL-UNNAMED -cp /home/sd/* fctreddit.impl.server.rest.Us
ersServer
7a49c400e038e71620a024b903f80e75f8614871ad7769fe32cb19c363ae3eeb
---------- Creating valid users
Generate some users...
Request: POST http://172.20.0.3:8080/rest/users
Header: Accept = [application/json]
Header: Content-Type = [application/json]
Entity: User [email=ocie.lindgren@balistreri.org, userId=ocie.lindgren, fullName=Ocie Lindgren, password=zb1s8u4
y, avatarUrl=null]
```

Teste 2a: Verifica se o servidor de Users com API REST e possivel de descobrir por multicast. Neste caso teve sucesso.

TESTER (EXEMPLO DE EXECUÇÃO):

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
2a )     Testing Discovery Service [ listening for announcements    ]
----------------------------------------------------------------------
Starting: Users : [1] REST Servers, [0] GRPC Servers
Started: users-1 : java --add-opens java.base/java.lang=ALL-UNNAMED -cp /home/sd/* fctreddit.impl.server.rest.Us
fa584049ea1a92e3928db5ecb0b2d60f3ae24a6297de7383766e97d7d5b1cf41
Discovery started @ /226.226.226.226:2266
INFO: Users     http://172.20.0.3:8080/rest

Found: Users
OK

killByImage(sd2425-tp1-api-jleitao): Removing container: [/users-1]
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
3a )     Testing Users Service [ createUser ]
----------------------------------------------------------------------
Starting: Users : [1] REST Servers, [0] GRPC Servers
Started: users-1 : java --add-opens java.base/java.lang=ALL-UNNAMED -cp /home/sd/* fctreddit.impl.server.rest.Us
ersServer
7a49c400e038e71620a024b903f80e75f8614871ad7769fe32cb19c363ae3eeb
---------- Creating valid users
Generate some users...
Request: POST http://172.20.0.3:8080/rest/users
Header: Accept = [application/json]
Header: Content-Type = [application/json]
Entity: User [email=ocie.lindgren@balistreri.org, userId=ocie.lindgren, fullName=Ocie Lindgren, password=zb1s8u4
y, avatarUrl=null]
```

Notem que cada teste lança servidores necessários para suportar o teste.

TESTER (EXEMPLO DE EXECUÇÃO):

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
2a )     Testing Discovery Service [ listening for announcements... ]
----------------------------------------------------------------------
Starting: Users : [1] REST Servers, [0] GRPC Servers
Started: users-1 : java --add-opens java.base/java.lang=ALL-UNNAMED -cp /home/sd/* fctreddit.impl.server.rest.Us
ersServer
fa584049ea1a92e3928db5ecb0b2d60f3ae24a6297de7383766e97d7d5b1cf41
Discovery started @ /226.226.226.226:2266
INFO: Users     http://172.20.0.3:8080/rest

Found: Users
OK

killByImage(sd2425-tp1-api-jleitao): Removing container: [/users-1]
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
3a )     Testing Users Service [ createUser ]
----------------------------------------------------------------------
Starting: Users : [1] REST Servers, [0] GRPC Servers
Started: users-1 : java --add-opens java.base/java.lang=ALL-UNNAMED -cp /home/sd/* fctreddit.impl.server.rest.Us
ersServer
7a49c400e038e71620a024b903f80e75f8614871ad7769fe32cb19c363ae3eeb
---------- Creating valid users
Generate some users...
Request: POST http://172.20.0.3:8080/rest/users
Header: Accept = [application/json]
Header: Content-Type = [application/json]
Entity: User [email=ocie.lindgren@balistreri.org, userId=ocie.lindgren, fullName=Ocie Lindgren, password=zb1s8u4
y, avatarUrl=null]
```

E esses servidores são removidos no final de cada teste.

TESTER (EXEMPLO DE EXECUÇÃO):

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
2a )     Testing Discovery Service [ listening for announcements... ]
----------------------------------------------------------------
Starting: Users : [1] REST Servers, [0] GRPC Servers
Started: users-1 : java --add-opens java.base/java.lang=ALL-UNNAMED -cp /home/sd/* fctreddit.impl.server.rest.Us
ersServer
fa584049ea1a92e3928db5e
Discovery started @ /22
INFO: Users     http://

Found: Users
OK

killByImage(sd2425-tp1-api-jleitao): Removing container: [/users-1]
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
3a )     Testing Users Service [ createUser ]
----------------------------------------------------------------
Starting: Users : [1] REST Servers, [0] GRPC Servers
Started: users-1 : java --add-opens java.base/java.lang=ALL-UNNAMED -cp /home/sd/* fctreddit.impl.server.rest.Us
ersServer
```

O input de operações é mostrado, e em caso de erro é reportado o valor esperado e qual o recebido. Analisem o input e o erro para idenficarem problemas na vossa implementação.

```
---------- Creating valid users
Generate some users...
Request: POST http://172.20.0.3:8080/rest/users
Header: Accept = [application/json]
Header: Content-Type = [application/json]
Entity: User [email=ocie.lindgren@balistreri.org, userId=ocie.lindgren, fullName=Ocie Lindgren, password=zb1s8u4
y, avatarUrl=null]
```

GOALS

In the end of this lab you should be able to:

- Understand what is required for the first project.
- Understand some of the semantics of the operations.
- Know how to use the automated tester to test your solution.

- BONUS: Como evitar repetir código em cada invocação REST de forma a conseguir lidar com erros de comunicação transientes.

RE-EXECUTION OF A REST REQUEST

```
public Result<String> createUser(User user) {

    for(int i = 0; i < MAX_RETRIES ; i++) {
        try {
            Response r = target.request()
                    .accept( MediaType.APPLICATION_JSON)
                    .post(Entity.entity(user, MediaType.APPLICATION_JSON));


            int status = r.getStatus();
            if( status != Status.OK.getStatusCode() )
                return Result.error( getErrorCodeFrom(status));
            else
                return Result.ok( r.readEntity( String.class ));

        } catch( ProcessingException x ) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException e) {
                //Nothing to be done here.
            }
        }
        catch( Exception x ) {
            x.printStackTrace();
        }
    }
    return Result.error(  ErrorCode.TIMEOUT );
}
```

In lab 3 we saw this example, in which we surround each invocation with a for loop.

RE-EXECUTION OF A REST REQUEST

```
public Result<String> createUser(User user) {

    for(int i = 0; i < MAX_RETRIES ; i++) {
        try {
            Response r = target.request()
                    .accept( MediaType.APPLICATION_JSON)
                    .post(Entity.entity(user, MediaType.APPLICATION_JSON));

            int status = r.getStatus();
            if( status != Status.OK.getStatusCode() )
                return Result.error( getErrorCodeFrom(status));
            else
                return Result.ok( r.readEntity( String.class ));

        } catch( ProcessingException x ) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException e) {
                //Nothing to be done here.
            }
        }
        catch( Exception x ) {
            x.printStackTrace();
        }
    }
    return Result.error(  ErrorCode.TIMEOUT );
}
```

target.request().accept(MediaType.APPLICATION_JSON)

Returns a request **Builder** prepared to execute an HTTP request.

RE-EXECUTION OF A REST REQUEST

```
public Result<String> createUser(User user) {

    for(int i = 0; i < MAX_RETRIES ; i++) {
        try {
            Response r = target.request()
                    .accept( MediaType.APPLICATION_JSON)
                    .post(Entity.entity(user, MediaType.APPLICATION_JSON));


            int status = r.getStatus();
            if( status != Status.OK.getStatusCode() )
                return Result.error( getErrorCodeFrom(status));
            else
                return Result.ok( r.readEntity( String.class ));

        } catch( ProcessingException x ) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException e) {
                //Nothing to be done here.
            }
        }
        catch( Exception x ) {
            x.printStackTrace();
        }
    }
    return Result.error(  ErrorCode.TIMEOUT );
}
```

Post and Put methods typically receive an **Entity** as an argument.

RE-EXECUTION OF A REST REQUEST

```
public Result<String> createUser(User user) {

    for(int i = 0; i < MAX_RETRIES ; i++) {
        try {
            Response r = target.request()
                    .accept( MediaType.APPLICATION_JSON)
                    .post(Entity.entity(user, MediaType.APPLICATION_JSON));


            int status = r.getStatus();
            if( status != Status.OK.getStatusCode() )
                return Result.error( getErrorCodeFrom(status));
            else
                return Result.ok( r.readEntity( String.class ));

        } catch( ProcessingException x ) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException e) {
                //Nothing to be done here.
            }
        }
        catch( Exception x ) {
            x.printStackTrace();
        }
    }
    return Result.error(  ErrorCode.TIMEOUT );
}
```

Each REST operation is effectively executed when the corresponding function is invoked on the Builder:

post

put

get

delete

RE-EXECUTION OF A REST REQUEST

```
public Result<String> createUser(User user) {

    for(int i = 0; i < MAX_RETRIES ; i++) {
        try {
            Response r = target.request()
                    .accept( MediaType.APPLICATION_JSON)
                    .post(Entity.entity(user, MediaType.APPLICATION_JSON));


            int status = r.getStatus();
            if( status != Status.OK.getStatusCode() )
                return Result.error( getErrorCodeFrom(status));
            else
                return Result.ok( r.readEntity( String.class ));

        } catch( ProcessingException x ) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException e) {
                //Nothing to be done here.
            }
        }
        catch( Exception x ) {
            x.printStackTrace();
        }
    }
    return Result.error(  ErrorCode.TIMEOUT );
}
```

We can therefore abstract the invocation into a pair of parameters—the Builder and, potentially, the Entity (in the case of Post and Put)—and have helper functions that can perform the invocation loop by receiving these parameters and returning a Response.

RE-EXECUTION OF A REST REQUEST

```
private Response executeOperationPost(Builder req, Entity<?> e) {
    for (int i = 0; i < MAX_RETRIES; i++) {
        try {
            return req.post(e);
        } catch (ProcessingException x) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException ex) {
                // Nothing to be done here.
            }
        } catch (Exception x) {
            x.printStackTrace();
        }
    }
    return null;
}
```

RE-EXECUTION OF A REST REQUEST

```
private Response executeOperationPost(Builder req, Entity<?> e) {
    for (int i = 0; i < MAX_RETRIES; i++) {
        try {
            return req.post(e);
        } catch (ProcessingException x) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException ex) {
                // Nothing to be done here.
            }
        } catch (Exception x) {
            x.printStackTrace();
        }
    }
    return null;
}
```

We'll need four helper functions — one for each type of REST operation.

RE-EXECUTION OF A REST REQUEST

```
private Response executeOperationPost(Builder req, Entity<?> e) {
    for (int i = 0; i < MAX_RETRIES; i++) {
        try {
            return req.post(e);
        } catch (ProcessingException x) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException ex) {
                // Nothing to be done here.
            }
        } catch (Exception x) {
            x.printStackTrace();
        }
    }
    return null;
}
```

In the case of POST (and likewise PUT), we'll receive two arguments: the request Builder and the Entity to be sent.

RE-EXECUTION OF A REST REQUEST

```
private Response executeOperationPost(Builder req, Entity<?> e) {
    for (int i = 0; i < MAX_RETRIES; i++) {
        try {
            return req.post(e);
        } catch (ProcessingException x) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException ex) {
                // Nothing to be done here.
            }
        } catch (Exception x) {
            x.printStackTrace();
        }
    }
    return null;
}
```

The for loop remains the same as we had before.

RE-EXECUTION OF A REST REQUEST

```
private Response executeOperationPost(Builder req, Entity<?> e) {
    for (int i = 0; i < MAX_RETRIES; i++) {
        try {
            return req.post(e);
        } catch (ProcessingException x) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException ex) {
                // Nothing to be done here.
            }
        } catch (Exception x) {
            x.printStackTrace();
        }
    }
    return null;
}
```

Except that the only thing we try to do is return the Response generated by invoking the post method (parameterized with the Entity).

RE-EXECUTION OF A REST REQUEST

```
private Response executeOperationPost(Builder req, Entity<?> e) {
    for (int i = 0; i < MAX_RETRIES; i++) {
        try {
            return req.post(e);
        } catch (ProcessingException x) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException ex) {
                // Nothing to be done here.
            }
        } catch (Exception x) {
            x.printStackTrace();
        }
    }
    return null;
}
```

If we exhaust all attempts, we'll have to return null.
This possibility must be handled by whoever calls this function.

RE-EXECUTION OF A REST REQUEST

```
public Result<String> createUser(User user) {

    Response r = executeOperationPost(target.request().accept(MediaType.APPLICATION_JSON),
            Entity.entity(user, MediaType.APPLICATION_JSON));

    if (r == null)
        return Result.error(ErrorCode.TIMEOUT); // We were not able to obtain a response in the maximum time;

    int status = r.getStatus();
    if (status != Status.OK.getStatusCode())
        return Result.error(getErrorCodeFrom(status));
    else
        return Result.ok(r.readEntity(String.class));

}
```

RE-EXECUTION OF A REST REQUEST

```
public Result<String> createUser(User user) {

    Response r = executeOperationPost(target.request().accept(MediaType.APPLICATION_JSON),
            Entity.entity(user, MediaType.APPLICATION_JSON));

    if (r == null)
        return Result.error(ErrorCode.TIMEOUT); // We were not able to obtain a response in the maximum time;

    int status = r.getStatus();
    if (status != Status.OK.getStatusCode())
        return Result.error(getErrorCodeFrom(status));
    else
        return Result.ok(r.readEntity(String.class));

}
```

We can now rewrite the createUser function without needing the loop to repeat the request in case of a transient communication failure.

RE-EXECUTION OF A REST REQUEST

```
public Result<String> createUser(User user) {

    Response r = executeOperationPost(target.request().accept(MediaType.APPLICATION_JSON),
            Entity.entity(user, MediaType.APPLICATION_JSON));

    if (r == null)
        return Result.error(ErrorCode.TIMEOUT); // We were not able to obtain a response in the maximum time;

    int status = r.getStatus();
    if (status != Status.OK.getStatusCode())
        return Result.error(getErrorCodeFrom(status));
    else
        return Result.ok(r.readEntity(String.class));

}
```

We try to obtain the Response by executing this method and passing the request Builder and the Entity as arguments.

RE-EXECUTION OF A REST REQUEST

```
public Result<String> createUser(User user) {

    Response r = executeOperationPost(target.request().accept(MediaType.APPLICATION_JSON),
            Entity.entity(user, MediaType.APPLICATION_JSON));

    if (r == null)
        return Result.error(ErrorCode.TIMEOUT); // We were not able to obtain a response in the maximum time;

    int status = r.getStatus();
    if (status != Status.OK.getStatusCode())
        return Result.error(getErrorCodeFrom(status));
    else
        return Result.ok(r.readEntity(String.class));

}
```

If the response is null, we know that the number of attempts has been exhausted, and we send a corresponding ErrorCode.

RE-EXECUTION OF A REST REQUEST

```
public Result<String> createUser(User user) {

    Response r = executeOperationPost(target.request().accept(MediaType.APPLICATION_JSON),
            Entity.entity(user, MediaType.APPLICATION_JSON));

    if (r == null)
        return Result.error(ErrorCode.TIMEOUT); // We were not able to obtain a response in the maximum time;

    int status = r.getStatus();
    if (status != Status.OK.getStatusCode())
        return Result.error(getErrorCodeFrom(status));
    else
        return Result.ok(r.readEntity(String.class));
}
```

Otherwise, we can process the Response received from the server as we did before.

RE-EXECUTION OF A REST REQUEST

```
private Response executeOperationGet(Builder req) {
    for (int i = 0; i < MAX_RETRIES; i++) {
        try {
            return req.get();
        } catch (ProcessingException x) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException e) {
                // Nothing to be done here.
            }
        } catch (Exception x) {
            x.printStackTrace();
        }
    }
    return null;
}
```

RE-EXECUTION OF A REST REQUEST

```
private Response executeOperationGet(Builder req) {
    for (int i = 0; i < MAX_RETRIES; i++) {
        try {
            return req.get();
        } catch (ProcessingException x) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException e) {
                // Nothing to be done here.
            }
        } catch (Exception x) {
            x.printStackTrace();
        }
    }
    return null;
}
```

In the case of the Get (and likewise Delete) operation, the code is similar, but we only need one argument — the request Builder.

RE-EXECUTION OF A REST REQUEST

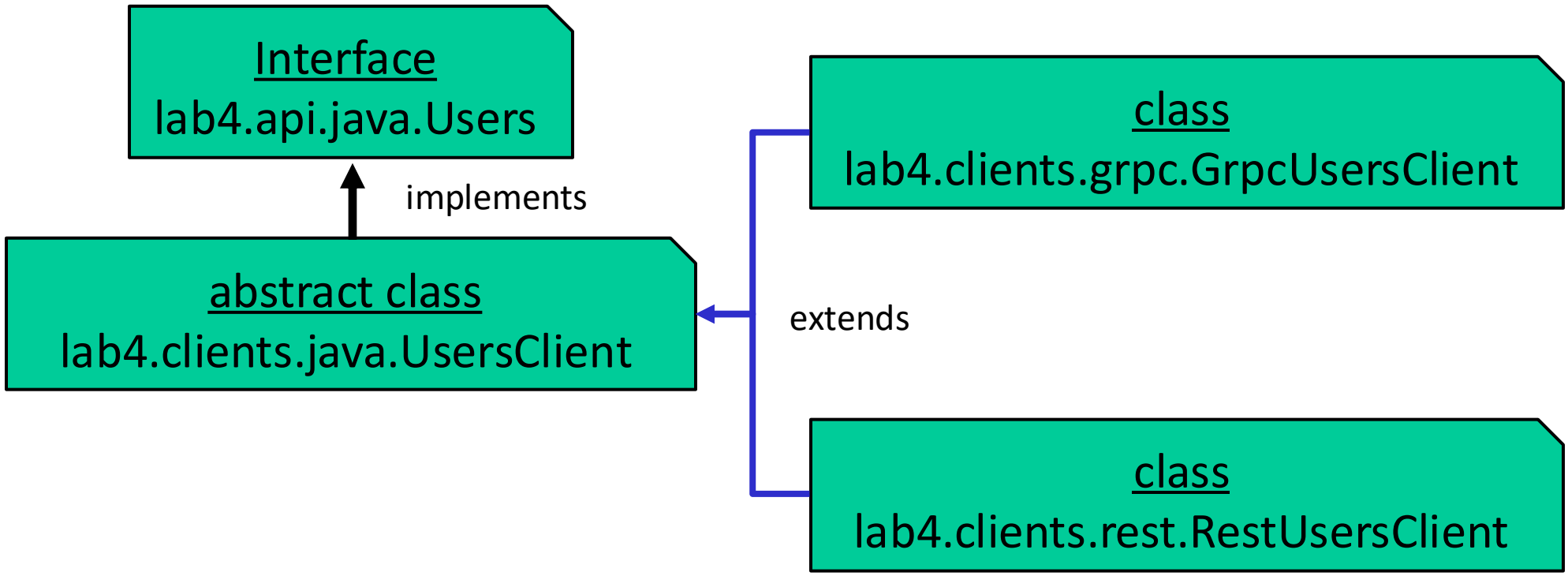
```
private Response executeOperationGet(Builder req) {
    for (int i = 0; i < MAX_RETRIES; i++) {
        try {
            return req.get();
        } catch (ProcessingException x) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException e) {
                // Nothing to be done here.
            }
        } catch (Exception x) {
            x.printStackTrace();
        }
    }
    return null;
}
```

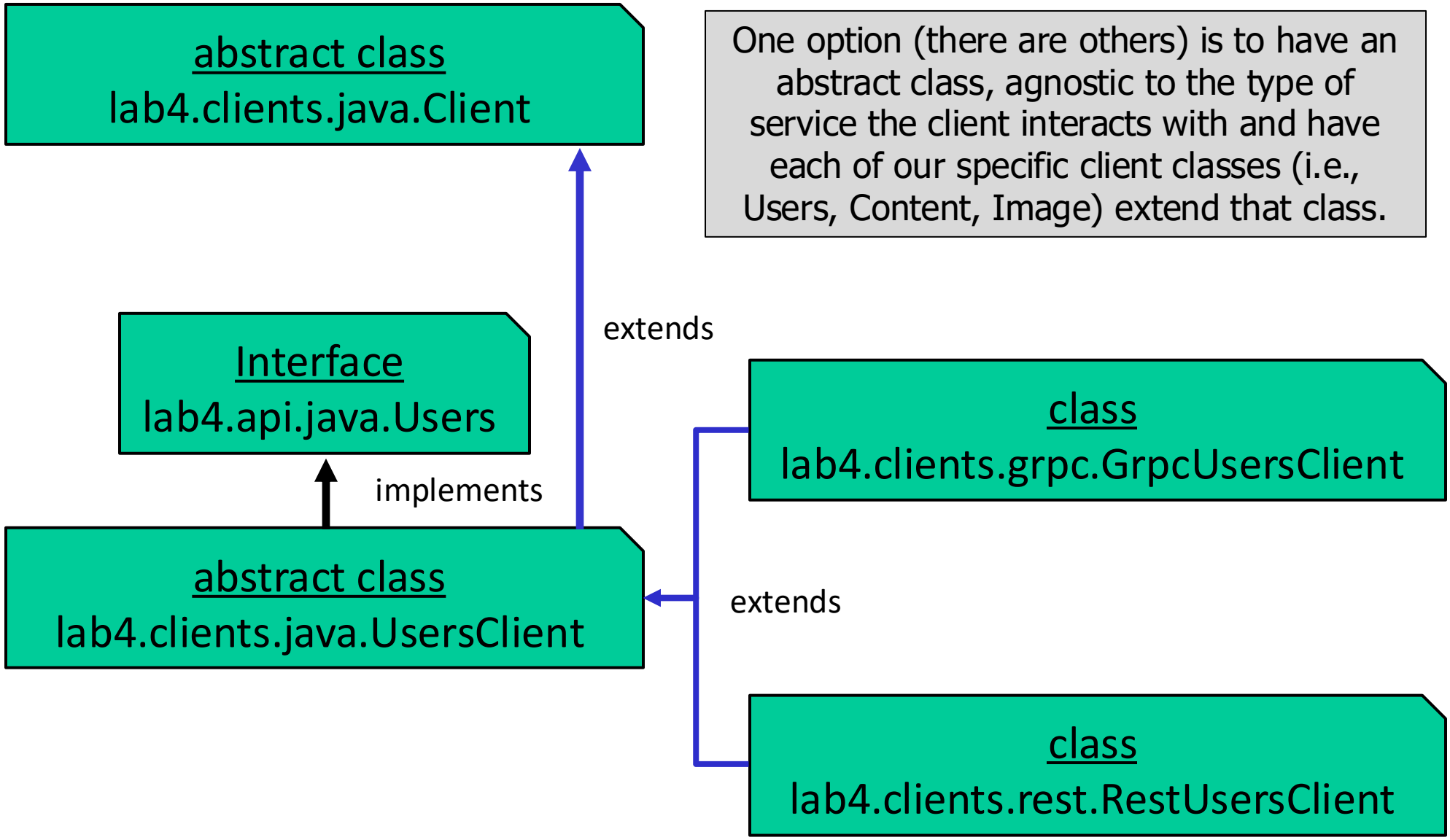
And obviously, the operation we perform based on that request is different!

Where to Define These Auxiliary Functions?

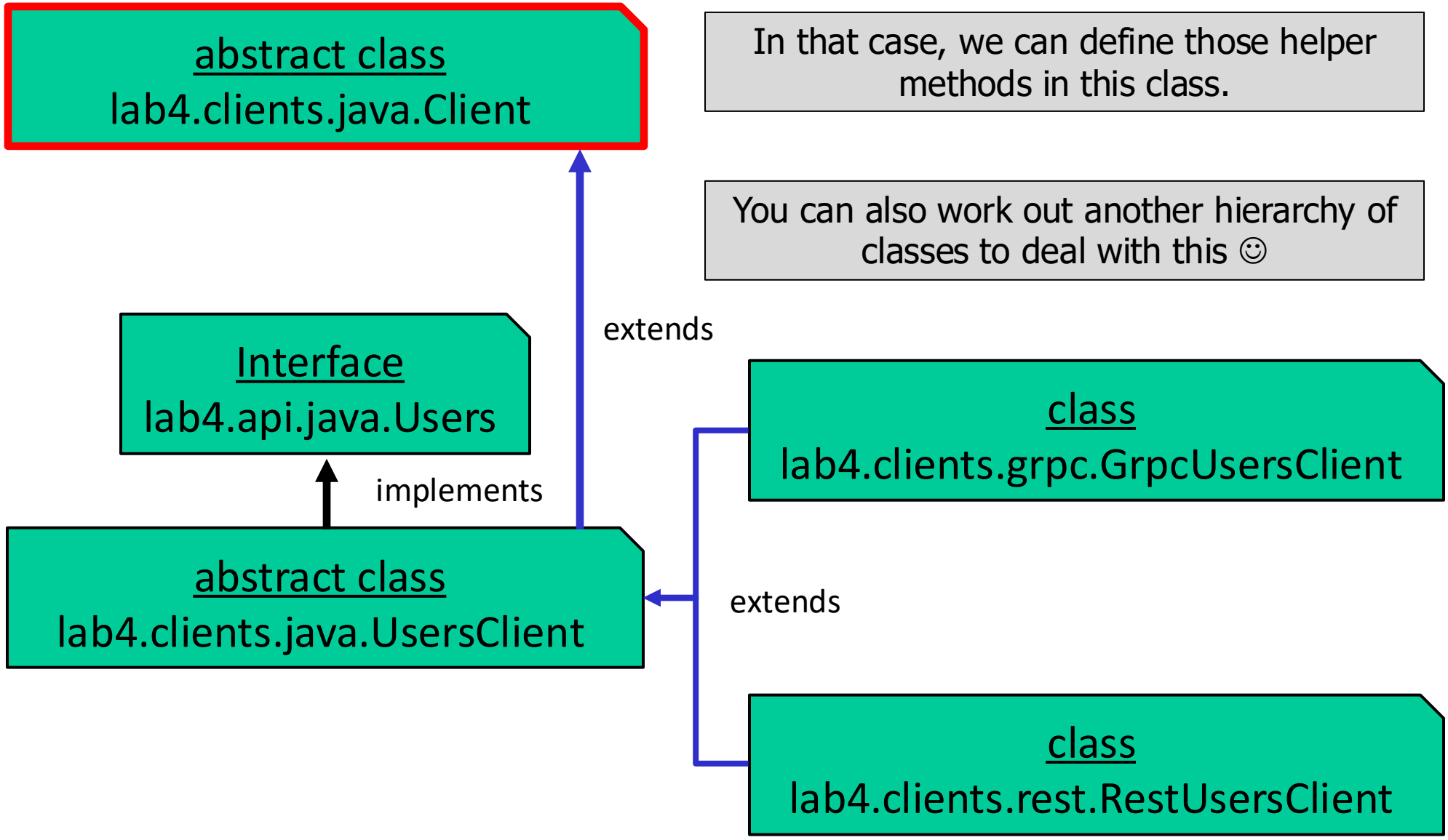
Last week we saw this class model to implement the REST and gRPC clients for the Users service. But in the project, we'll have clients for Users, Content, and Image (all of which will need this functionality).



Where to Define These Auxiliary Functions?



Where to Define These Auxiliary Functions?



RE-EXECUTION OF A REST REQUEST

```
public Result<String> createUser(User user) {

    Response r = executeOperationPost(target.request().accept(MediaType.APPLICATION_JSON),
            Entity.entity(user, MediaType.APPLICATION_JSON));

    if (r == null)
        return Result.error(ErrorCode.TIMEOUT); // We were not able to obtain a response in the maximum time;

    int status = r.getStatus();
    if (status != Status.OK.getStatusCode())
        return Result.error(getErrorCodeFrom(status));
    else
        return Result.ok(r.readEntity(String.class));

}
```

This piece of code is also very similar in all functions (with the key difference being if there is a value returned or not). So, you can also externalize to a function using the same technique.

EXERCISE

1. Verify that you can use the Tester
2. Work in your project ☺