Orientações gerais desta atividade:

- *edição: qualquer ide ou editor de texto*
- *compilar,executar: maven na linha de comandos (comando mvn)*

A- Exemplo de REST com Jersey

Jersey é um framework Java que implementa **Java API for RESTful Web Services** (JAX-RS) para a construção de serviços **RESTful**.

https://eclipse-ee4j.github.io/jersey/

 # versão rápida, HelloWorld diretamente na linha de comandos

1- num terminal, execute:

```
mvn archetype:generate -DarchetypeArtifactId=jersey-quickstart-grizzly2 \
-DarchetypeGroupId=org.glassfish.jersey.archetypes -DinteractiveMode=false \
-DgroupId=sd -DartifactId=simple-service -Dpackage=sd.rest1 \
-DarchetypeVersion=3.0.3
```

(Em caso de erro, confirme que tem versões de maven e java em conformidade com o recomendado nesta UC.
Em alguns casos, poderão fazer o upgrade manual à ferramenta maven (ver zona do moodle sobre maven).

2- depois entrar na pasta gerada

veja os ficheiros:

```
simple-service/
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── sd
    │           └── rest1
    │               ├── Main.java
    │               └── MyResource.java
    └── test
        └── java
            └── sd
                └── rest1
                    └── MyResourceTest.java
```

Analise o ficheiro de configuração com as dependências do projeto: pom.xml

3- Ver classe com o serviço (resource) e também a classe que ativa um *container* para o serviço (no caso com Grizzly).

4- Build do projeto

mvn compile

reparar nos testes unitários JUnit

mvn test

5- executar
mvn exec:java

6- Aceda às operações do serviço com um cliente externo

- browser (consegue aceder apenas a operações associadas a HTTP GET, podia haver outras)
- linha de comandos (passando outras opções, podia fazer GET,POST,DELETE,PUT)

```
$ curl http://localhost:8080/myresource
```

ou para ver mais detalhes:

```
$ curl -i http://localhost:8080/myresource
```

Pode encontrar "o equivalente a um cliente" no ficheiro:
src/test/java/sd/rest1/MyResourceTest.java

B- Um olhar mais atento sobre REST

REST - *REpresentational State Transfer*

é uma arquitetura de serviços com algumas semelhanças com a norma SOAP Web Services.

Nesta arquitetura existem recursos.
Cada recurso é identificado por um URI, através do qual pode ser alvo de operações.
Tipicamente, a operação a executar num recurso é associada ao pedido/método HTTP usado (PUT, POST, GET, DELETE - para criar, alterar, obter, apagar). Isto reduz a quantidade de dados a passar pela rede (em comparação com SOAP). O próprio URI no pedido HTTP identifica o recurso.

Stateless
Um serviço REST (puro) liberta o servidor do conceito de sessão. Todos os elementos necessários para atender um pedido serão passados pelo cliente. Isto contribui para uma maior escalabilidade junto do servidor.

Formato de dados: Representação dos recursos
HTML, XML, JSON, plain text ou outros...

Serviço RESTful: obedece a todos os princípios teóricos (recursos expostos por URI, operações com métodos HTTP PUT, GET, POST e DELETE; Self-descriptive messages, Layered system)

Se um serviço usa a tecnologia mas não segue todos os princípios, é um serviço REST mas não RESTful.

C- serviço REST com troca de dados

Ativação de um serviço REST, envolvendo:
- **maven**: utilitário para controlar as dependências do código, compilação, e outros aspetos avançados
- **jersey**: para facilitar a representação dos dados num formato apropriado para o transporte (equivalente aos detalhes de marshalling)

- **grizzly**: Application Server para alojar o serviço desenvolvido. Alternativas: Apache Tomcat, Glassfish, JBoss...

Este serviço guarda a lista de alunos (com número e nome) que fazem parte de uma turma.
Existem classes que representam esses dados: Aluno, Turma.

1- Pode descarregar o código inicial daqui.

o conteúdo:

```
├── pom.xml        # configuração do projeto maven
└── src
    ├── main
    │   └── java
    │       └── sd
    │           └── rest
    │               ├── Aluno.java              # dados
    │               ├── MainAppServer.java      # classe para o container/servidor que aloja o serviço
    │               ├── Turma.java              # dados
    │               └── TurmaResource.java      # serviço para um certo tipo de recursos (turma)
    └── test
        └── java
            └── sd
                └── rest
                    └── MainAppServerTest.java   # testes unitários
```

Na pasta base, verá o `pom.xml` e a pasta src.

2- Para **compilar**: `mvn compile`

Depois de compilar verá mais uma pasta: target.

3- Para testar o serviço, pode usar os testes JUnit, executando:
$ mvn test
Se tudo correr bem, verá:

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] --
[INFO] BUILD SUCCESS

Isto ativou o serviço, fez 1 testes com operações sobre o serviço, cujas respostas são validadas.

4- Executar um servidor onde o serviço ficará disponível:
mvn exec:java
Isto ativa uma instância do application server Grizzly e faz deploy do serviço.
Para terminar, basta carregar no enter.

5- No browser, abra:

http://localhost:8001/sd/turma

Isto corresponde a uma das operações do serviço. Está a listar todos os alunos na turma.

6- Para ver respostas do serviço, pode usar a ferramenta curl.
Pode simular um cliente e pedir os dados no formato que entender. Exemplos:

```
$ curl http://localhost:8001/sd/turma
$ curl -HAccept:application/json http://localhost:8001/sd/turma
```

7- Veja o código de TurmaResource.
Note as constantes anotações, relativas a partes do URL, ao método HTTP ou ao formato de dados.

```
@Path("/add")
@POST
@Consumes({"application/json", "application/xml"})
@Produces({"application/json", "application/xml"})
public Turma addAluno(@QueryParam("numero") int numero,
@QueryParam("nome") String nome
) {
    turma.add(new Aluno(numero,nome));
    return turma;
}
```

Note que para cada **método HTTP** (GET, POST, PUT, DELETE) e cada **Path** deve haver no máximo uma operação/método no resource. Quando o path não é indicado junto à operação, significa que essa operação é executada quando chegar um pedido para o **path do próprio recurso**, assinalado no início da classe.

8.1- Tentar isto no browser:

http://localhost:8001/sd/turma/add?numero=2&nome=teste

Por que motivo não funciona?
E se o resource aceitasse pedidos GET para esta operação?

8.2- Teste (alterando o POST para GET junto à operação)... e depois consulte a (nova) turma...

É uma forma simples de criar uma nova instância.
Note que, nesta experiência, o uso do GET é contra os princípios RESTful (a criação de recursos devia ser apenas com PUT, mas para efeitos de teste, é mais simples simular o GET diretamente com o browser - em todo o caso, pode implementar serviços REST e não RESTful, desde que tenha a noção dessa opção).

Nota: Se houver testes JUnit para a operação que acabou de alterar... É normal que `mvn test` denuncie um erro por não encontrar a operação via POST - pode reverter a situação ou ajustar a classe de testes, se necessário.

D - Clientes noutra Linguagem

1- Podemos agora ver outras tecnologias diferentes.

Por exemplo, para o serviço REST com a turma, podia usar também este cliente Python3.

(e existem outros módulos/pacotes para serviços REST e JSON)

E - Outros Serviços via REST

1- Consulte informação sobre os serviços REST de Location API of Bing Maps.
Objetivo: Queremos dar um endereço (parcial), de Portugal, e obter uma resposta com concelho, distrito e coordenadas da *BoundingBox* (retângulo que inclui aquela região), para um ou mais resultados correspondentes aquela morada.

2- exercício: Construa uma aplicação cliente para este serviço, que permita ler uma morada e descobrir a sua possível localização em Portugal, como explicado acima.

Para usar a Location API é necessária uma chave de *developer* do Bing Maps. Pode indicar esta chave temporária:
AvMkksfWKfV8cIY_V8p97ySFDqOzZdwjnhO7k8raL1CNU5ZpcrF9dQ_38NHXoEwL

Para ter uma ideia da resposta, veja este exemplo de pesquisa por "*evoramonte*", clicando aqui (note o URL).

Pretende-se, para cada resultado, mostrar o concelho, distrito e *boundingBox*.