FAQ

Laboratórios de Informática III
2021/2022

Última atualização: 2021-12-10 14:05

1. Como deve estar organizado o repositório?

Na raiz do repositório, deve existir uma diretoria por cada guião lançado. A raiz do repositório terá a seguinte estruturação:

- guiao-1/
- guiao-2/
- guiao-3/

Todos os elementos que compõem a resolução de um guião devem estar dentro da respetiva diretoria. Ou seja, a composição da diretoria “guiao-1” será:

- guiao-1/
    - docs/
    - entrada/
    - libs/
    - obj/
    - src/
    - Makefile

Esta estruturação será idêntica para as diretorias de cada um dos outros guiões. Para além disso, esta estruturação não impede que os grupos acrescentem outros elementos necessários à elaboração do seu trabalho. Exemplos disso são a diretoria “saida/”, a diretoria “report/” para quem quiser aí colocar/organizar o relatório, um ficheiro README.md, etc.

De notar que a Makefile deve estar dentro da respetiva diretoria de guião. Ou seja, terão uma Makefile para cada guião. O executável gerado por cada Makefile deve ser colocado dentro da respetiva diretoria de guião. **Exemplo:** a Makefile dentro da diretoria “guiao-1/” deve gerar o executável “guiao-1” e colocá-lo nessa mesma diretoria.

Cada Makefile só deve gerar o executável do respetivo guião. **Exemplo:** a Makefile do guião 2 não é responsável por criar um executável para o guião 1.

2. O guião 2 deve executar o passo de validação dos dados?

O guião 1 incide sobretudo na validação dos dados e esse não é o foco do guião 2. Como tal, ao carregar os dados para memória, a execução do passo de validação é opcional. Ou seja, podemos partir do princípio que os ficheiros fornecidos como input para o guião 2 não têm dados inválidos e é isso que vai ser considerado na avaliação do guião 2.

Uma sugestão é utilizar como input para o guião 2 os ficheiros produzidos pelo vosso programa do guião 1.

Contudo, é boa prática construírem os módulos do guião 2 (um dos principais focos deste guião) de forma a que seja fácil trocar a parte da implementação que efetua a leitura dos dados. Há várias formas de cumprir com isto.

Uma sugestão mais concreta é, durante a elaboração do guião 2, planear o módulo de leitura de forma a que este consiga acomodar uma função de leitura que efetue a validação e outra função de leitura que não efetue este passo. Se estas duas funções tiverem assinaturas idênticas, é fácil trocar a que é executada.

O esforço colocado numa implementação mais flexível representa tempo e esforço poupado no guião 3, em que o passo de validação terá uma importância mais relevante.

3. O que representa um colaborador de um repositório?

No trabalho de LI3, um utilizador é considerado um colaborador de um repositório se for o **autor** (author_id) ou o ***committer*** (commiter_id) de pelo menos um *commit* do repositório em questão.

4. O ficheiro de input dos comandos deve ser validado?

Podem assumir que o ficheiro de comandos dado como input é válido. Isto é, não tem linhas vazias ou comandos inválidos do ponto de vista sintático (ex: argumentos em falta para uma determinada query) ou semântico (ex: datas impossíveis).

É valorizado o trabalho de implementar algum mecanismo de validação de comandos. Contudo, este esforço deve ser colocado em segundo plano face à finalização do projeto.

Se optarem por dedicar tempo a esta tarefa, algumas sugestões são:

- Produzir um ficheiro de output vazio para um comando vazio
- Descartar comandos com datas inválidas (ou outros dados)

Estas implementações são trabalho adicional. Mais uma vez, este esforço é certamente valorizado. Se tiverem tomando decisões diferentes às sugestões, existe sempre espaço para discutirmos essas partes durante a defesa do trabalho prático.

5. O campo "*updated_at*" dos repositórios é a data do último *commit*?

Não necessariamente. Não existe garantia de que um repositório não seja atualizado depois da data do seu último *commit*. O que acontece é que os dados podem não cobrir a ação

que levou a essa atualização (ex: atualização do texto descritivo de um repositório depois da data do último *commit* registado).

Portanto, quando se pretende saber a inatividade de um repositório com base nos seus *commits*, a lista de *commits* deve ser consultada. Ou seja, não se deve utilizar a informação do campo "*updated_at*" para calcular a inatividade.

6. Como deve ser interpretado o ficheiro de *input* e organizado o *output*?

Na execução do programa, este deve receber como argumento o nome do ficheiro de texto onde estão descritas as *queries* a executar. Considerando um executável com o nome `guiao-2` e um ficheiro de texto `commands.txt` que contém os comandos pretendidos, a execução ocorreria da seguinte forma:

```
./guiao-2 commands.txt
```

De notar a diferença entre as noções de **comando** e ***query***. Um comando, que corresponde a uma linha do ficheiro de texto de *input*, indica a *query* que se pretende executar.

No mesmo ficheiro de texto, a mesma *query* pode ser solicitada várias vezes sob a forma de diferentes comandos.

O exemplo seguinte corresponde a um ficheiro possível de comandos e está presente no enunciado do guião 2:

```
1
5 100 2010-01-01 2015-01-01
6 5 Python
7 2014-04-25
8 3 2016-10-05
9 4
6 3 Haskell
10 50
```

Este ficheiro tem 8 linhas e, portanto, tem 8 comandos para execução. Os comandos 3 e 7 (linhas 3 e 7) correspondem ambos à execução da *query* 6 com a particularidade de terem parâmetros diferentes para a sua execução.

É esperado que, após a execução do programa para este ficheiro de comandos, sejam produzidos 8 ficheiros de *output* em que os seus nomes seguem a forma:

`command1_output.txt`
`command2_output.txt`
…
`command8_output.txt`

Cada um destes ficheiros de *output* terá o resultado produzido pela respetiva *query*. A título de exemplo, o ficheiro **`command2_output.txt`** terá o resultado da execução do **2º comando** (presente na linha 2) da *query* 5 com os argumentos **`100`**, **`2010-01-01`** e **`2015-01-01`**.