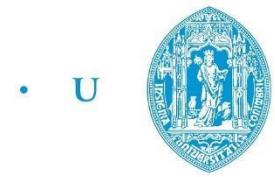
2022/23



FCTUC FACULDADE DE CIÊNCIAS
E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Trabalho 2

INTEGRAÇÃO DE SISTEMAS

João António Correia Vaz – 2019218159
joaovaz@student.dei.uc.pt
Gonçalo Tavares Antão Folhas Ferreira – 2019218159
tferreira@student.dei.uc.pt

Índice

Introdução

No segundo trabalho da cadeira, foi proposto ao grupo desenvolver uma aplicação reativa (servidor) de modo a expor alguns serviços possíveis na mesma. Juntamente também era necessário desenvolver uma aplicação cliente que pudesse interagir com a primeira. Foram usadas ferramentas tais como Reactor Flux e expressões lambda em java para o desenvolvimento das aplicações mencionadas. A estrutura do projeto assenta ainda numa base de dados de forma a podermos guardar e aceder a dados de forma reativa com WebFlux, uma framework de Spring.

A pasta zipada que foi enviada contém o presente relatório, a pasta servidor e a pasta webclient. Para testar é necessário executar primeiro o servidor (correr o DemoApplication.java) e depois o cliente (App.java). É necessário também estar associado à base de dados e que esta tenha informação, de forma que, quando forem executadas operações, existam outputs fidedignos.

Base de Dados

Como foi dito no capítulo anterior, tivemos a necessidade de criar uma base de dados que armazenasse os professores e os alunos gerados e onde fosse possível estabelecer uma relação many-to-many entre ambas as entidades. Assim, a nossa base de dados foi criada de forma manual com ajuda da plataforma Onda e do pgAdmin4.

Deste modo, existem 3 tabelas sendo que uma delas é apenas para estabelecer a relação entre as outras duas. Temos a tabela Teacher, que guarda os dados dos professores criados, e a tabela Student, que guarda os dados para os alunos criados, e, por fim, temos a tabela Student_teacher que guarda por cada linha o id de um professor e de um aluno de modo a criar uma ligação entre ambos. Esta tabela apesar de existir no modelo físico, não existe no modelo relacional por apenas ser necessária para a concretização da relação existente entre aluno e professor. Apresentamos abaixo ambos os diagramas.

Figura 1 - Modelo Conceptual da Base de Dados

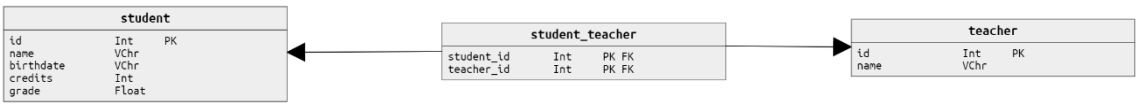


Figura 2 – Modelo Físico da Base de Dados

Teacher

A classe Teacher apresenta apenas 2 parâmetros: o id e o nome do professor. É feito um mapeamento desta classe de forma a associá-la à tabela correspondente. A esta também estão associados um repositório e um controlador onde estão definidas as operações que podem ser efetuadas, sendo elas do tipo GET, POST, PUT e DELETE. Estas operações incluem, por exemplo, a obtenção de todos os professores, a filtragem de um professor por id, a criação de um professor, a atualização da informação de um professor e, por fim, a eliminação um professor. As operações foram desenvolvidas com suporte de Flux (operações com vários dados) e Mono (operações com apenas uma entrada de dados).

Student

A classe Student apesenta vários parâmetros como o id, o nome, a data de nascimento, o número de créditos de cada estudante e a média do aluno. Assim, tal como na classe Teacher, é feito um mapeamento para a tabela na base de dados. O controlador apresenta as mesmas opções mencionada no Teacher e foram mais uma vez desenvolvidas com suporte de Flux e Mono de forma a garantir a obtenção os dados.

Servidor

A aplicação do servidor encontra-se distribuída em 3 partes como fora dito. Estas são: os modelos, os repositórios e os controladores. Possuímos ainda o DemoApplication que é o programa principal que, posteriormente, executa os outros. Dependendo da resposta do cliente são exigidas execuções de diferentes controladores. Para controlarmos o que acontece no servidor foi ainda feito um mecanismo de logging que indica que função é executada e quando é que ela foi chamada. Assim, é possível manter um controlo sobre o que acontece no servidor sendo que as chamadas do cliente são contínuas, isto é, toda a execução deste ocorre sequencialmente pelo que é preciso visionar os logs no terminal para saber exatamente que operações foram chamadas do lado do servidor. Os logs não são guardados em nenhum ficheiro de texto, pelo que a sua visualização é apenas possível no terminal.

Em termos estruturais mantivemos algo semelhante ao fornecido como exemplo no enunciado visto que a funcionalidade encontra-se presente e não é necessário fazer alterações significativas à mesma para obter o resultado pedido.

Cliente

A parte do Cliente encontra-se subdividida em 2 secções fundamentais: O ficheiro App.java, onde irão correr os webclients para executar as tarefas pedidas, e os modelos necessários para obter informação com os webclients, isto é, as classes Student, Teacher e StudentTeacher (relação entre student e teacher) e 2 classes extra que estruturam a informação necessária para alguns exercícios.

Relativamente aos modelos não há muito a apontar, visto serem os mesmos modelos disponibilizados no servidor. Contudo, existem 2 novos modelos: o TeacherCount e o StudentFullInfo, criados exclusivamente para os exercícios 10 e 11 respetivamente. O TeacherCount compreende apenas parâmetros sobre o nome do professor em questão e o seu número de alunos. Por outro lado, o StudentFullInfo apresenta toda a informação que o Student tem mais a lista de professores associados ao aluno.

O App.java contém uma execução contínua de webclients que fazem os acessos à base de dados diretamente e recolhem a informação com base nas definições pretendidas, isto é, com o suporte de filtros, sorts e maps. Deste modo, podemos selecionar a informação que queremos, evitando carregar a informação toda da base de dados. Isto facilita na procura de informação assim como no tratamento de dados para escrever nos ficheiros.

No início da execução do cliente é criada a pasta output onde serão guardados os resultados de cada webclient. Caso se execute o programa outra vez é feita uma verificação da existência da pasta output. Caso exista, esta e todos os seus conteúdos são eliminados, caso não exista, procede-se à criação da mesma seguida da execução do código. É dado um thread.sleep de 1.5 segundos à main de forma a permitir que estas operações sejam completadas. O output de cada webclient é gerado para o devido ficheiro texto com a nomenclatura ex*{nº do ex}*.txt e armazenado na pasta anteriormente criada.

Sobre o uso dos webclients, recorremos principalmente ao map() para fazer as pesquisas e retornar o valor encontrado e posteriormente escrever este para os ficheiros texto na secção do subscribe(). Clarificar que isto foi principalmente usado nos exercícios mais difíceis (10 e 11), uma vez que em alguns exercícios conseguíamos fazer logo a operação toda dentro do subscribe.

Recorremos ainda ao uso de ArrayLists para a realização dos exercícios que pediam cálculos como o da *standard deviation* e da média, uma vez que era necessário armazenar os valores todos para posteriormente a serem utilizados nestes cálculos.

Em relação à otimização temos de mencionar que o cliente apresenta um fluxo contínuo (sem menu) em que cada webclient trabalha na sua própria thread. Contudo, não funcionam de forma paralela, visto que para implementar isto seria necessário recurso a um semáforo para controlar os acessos. Contudo, podemos mencionar o facto de usarmos um thread.sleep de 6 segundos no fim da main para permitir que todas as threads corram neste intervalo e executem. Assim, após o término das mesmas, o sleep da main termina e o programa acaba. Existe ainda um sleep de 1 segundo entre 2 webclients uma vez que durante a testagem se viu necessário adicionar algum tempo para permitir a não concorrência de acessos. Podemos ainda referir o uso de filters nos webclients de forma a limitar os resultados de pesquisa e por sua vez diminuindo o dataset obtido, havendo por isso otimização na procura.

Conclusão

Para concluir, este projeto foi importante para perceber o funcionamento de algumas estruturas reativas e aplicações das mesmas. Foi possível interligar um cliente e servidor com uma base de dados de forma reativa e obter dados através de chamadas de webclients, também de forma reativa. Na nossa opinião que, no geral o projeto correu bem, a organização do grupo foi boa e os conceitos sugeridos foram bem adquiridos. Algumas funcionalidades poderiam ter corrido melhor no seu desenvolvimento. Fomos confrontados com alguns problemas que exigiram pesquisa externa e às vezes questionar os professores da PL mas pensamos que após a conclusão deste trabalho o nosso conhecimento sobre Modelos de Programação Reativos ficou bem estabelecido.

Webgrafia

https://spring.io/guides/gs/reactive-rest-service/

https://www.digitalocean.com/community/tutorials/spring-webflux-reactive-programming

https://howtodoinjava.com/spring-webflux/spring-webflux-tutorial/

https://www.digitalocean.com/community/tutorials/logger-in-java-logging-example

https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html

https://www.baeldung.com/reactor-core

https://www.codingame.com/playgrounds/929/reactive-programming-with-reactor-3/Flux

https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html