Relatório técnico do Projeto de LS

Autores: 44796 Rui Cunha
45373 Pedro Ribeiro
46109 João Cravo

Relatório para a Unidade Curricular de Laboratório de Software

2 – Julho – 2021

Introdução

A fase 1 da unidade curricular de Laboratório de Software tem como objetivo o desenvolvimento de um sistema, apoiado por uma base de dados, que faça a gestão de utilizadores, desportos, rotas e atividades. Um utilizador é definido por um nome, email(único) e identificador, um desporto é identificado por um nome, uma descrição e um identificador, uma rota tem duas localidades, uma inicial e a final, a distância entre elas e um identificador, e por fim, uma atividade tem o identificador do utilizador, a sua duração, a data, o identificador de rota(opcional) e um identificador. Um utilizador e um desporto podem conter várias atividades.

A aplicação funciona à base de comandos, e nesta fase, todos os métodos dos comandos se enquadram numa de três possibilidades: POST (criação de elementos de domínio e a sua adição à base de dados), GET (procura e obtenção desses elementos) e EXIT que fecha a aplicação.
Para além dos métodos, os comandos apresentam também uma path e parâmetros. A path define o recurso no qual o comando é executado e os parâmetros são pares" name=value" ou sequências deles separados por "&".

Para a fase 2, foram adicionados headers à estrutura que têm como principal objetivo incluir diferentes formatos de visualização dos dados e estruturas desenvolvidos na primeira fase. Existe também a adição de dois novos comandos.

Para a fase 3, foi criado um server HTTP que é capaz de receber e executar os comandos GET das fases anteriores. Com a ajuda do Postman é possível efetuar os pedidos e receber o resultado nos formatos também já explorados nas fases anteriores (Json, texto e Html). A arquitetura do servidor e a sua navegabilidade são realizadas com base no esquema dado pelos docentes, sendo assim o "esqueleto" do servidor definido e implementado.

Para fase final de projeto, o objetivo é implementar comandos POST na interface HTTP. Mas, devido a inconsistências relacionadas com as implementações, foi indicado pelos docentes a remodelação de código de modo a obter um código mais explícito, eficiente e simples, que conforme o tempo restante, possa ajudar na implementação da fase 4. No caso deste projeto, a implementação foi mudada, ou seja, código base das fases passadas foi alterado a todos os níveis e a fase 4, a implementação dos POST's não foi realizada.

Modelação da base de dados

Modelação conceptual

A aplicação apresenta o seguinte modelo EA:

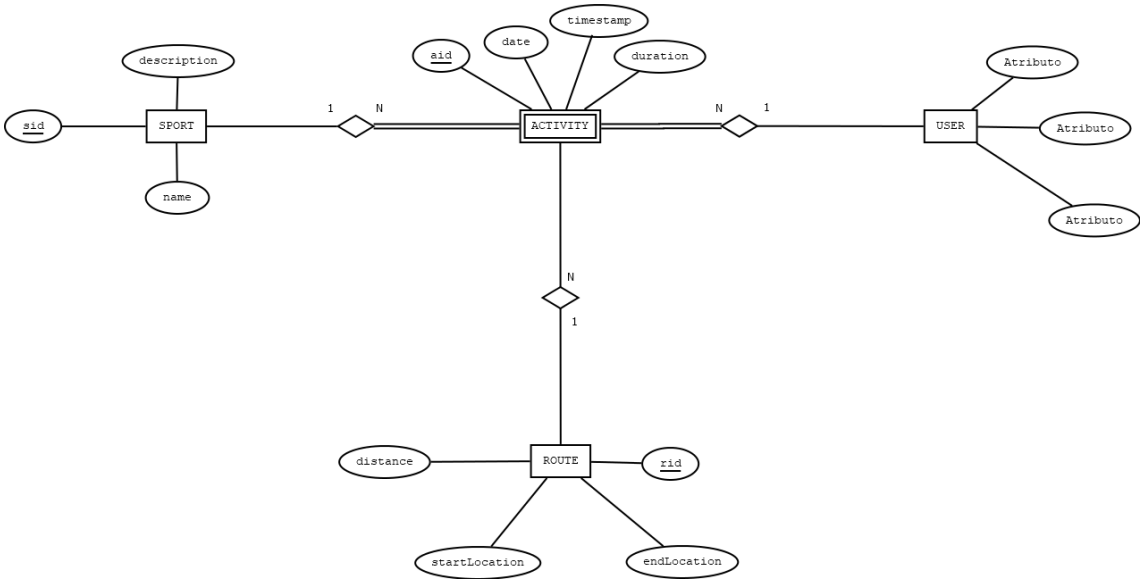

Figura 1: Modelo EA

Modelo Relacional

PK: chave primária

FK: chave estrangeira

CC: chave Candidata

User(<u>uid</u>,name,email)

PK: uid

CC: email

Restrições:

- Email tem que ser único

Sport(<u>sid</u>,name,description)

PK: sid

Route(<u>rid</u>,startLoc,endLoc,distance)

PK: rid

Restrições:

- Distância expressa em kms.

Activity(<u>aid</u>,duration,date,uid,rid,sid,timestamp)

PK: aid

FK: uid chave estrangeira de User.id, rid chave estrangeira de Route.rid, sid chave estrangeira de Sport.sid

Restrições:

- Data tem que estar no formato yyyy:mm:dd
- Duração tem que estar no formato hh:mm:ss.fff
- Quando existe timestamp, a atividade foi “apagada”

Modelação física

O modelo físico está presente em:

(Diretoria do Projeto) \2021-2-LI42D-G10\src\test\sql\createDBUser.sql

Processamento de comandos

A interface CommandHandler apresenta um único método(execute) que executa comandos SQL, recebendo como parâmetros um CommandRequest. Um ComamandRequest tem como parâmetros um mapa, que contém a informação já extraída de Path e parâmetros (Parameters), que também têm um mapa. A informação já extraída de Path encontra-se no primeiro parâmetro do CommandRequest e intitula-se de RouteResult. A obtenção da RouteResult está na classe Router, mais especificamente no método findRoute, onde caso encontremos uma Route(na nossa aplicação denominada RouteHandling, para diferenciar da Route do modelo) que seja validada(método e path validados), é criada uma nova RouteResult com o handler dessa Route e o mapa resultante da extração de Path. A validação de Path é feita na classe PathTemplate, e usando a Path como string, usamos o split e vamos comparar, primeiramente se a string de Path e PathTemplate tiverem tamanho igual (se for diferente o match é inválido), e depois adicionando num mapa a key(string, por exemplo: users/1, a key seria users e o value seria 1) e o value, conforme a existência ou não de dados postos pelo utilizador(nesta fase, os id's postos) e caso haja, esses dados serão values e as keys as Strings antes. Caso não haja esses dados, apenas as keys serão postas e os values ficaram a null. No fim do método comparamos esse mapa com o mapa de Path e caso sejam iguais, existe match.Por fim há o enumerado method, que contém os 3 valores: POST, GET e EXIT.

Para a segunda fase, é adicionado dois novos métodos, o DELETE, que na verdade vai atualizar a tabela com uma timestamp que indica a remoção de um elemento e o OPTIONS que apresenta uma lista dos comandos disponíveis.

Foi também alterada a estrutura do CommandRequest, no qual foi adicionado um componente opcional, os headers. A classe Headers é interpretada do mesmo modo que a Parameters (HashMap com key Value do mesmo formato que Parameters), tendo esta como função a visualização diferenciada de output nos seguintes formatos: texto, html e json, podendo ser opcionalmente visualizada em ficheiro.

Em caso de omissão de Headers, o output predefinido é de formato html.

Foi adicionado também o mecanismo de paginação que, apresentada uma lista, permite ao utilizador escolher quais resultados apresentar. Esta paginação é realizada dentro de cada comando, pois só tem em conta os parâmetros passados e, visto que são parâmetros a sua extração e verificação é feita de forma idêntica à primeira fase.

Para a fase 3, foi adicionado o comando LISTEN que tem como parâmetro um porto de onde vai receber pedidos. O comando LISTEN apresenta um parâmetro e tal como nos outros comandos, vai ser processado da mesma maneira. Quanto aos comandos GET processados por este LISTEN, esses já vão ser processados de maneira diferente e existem 2 classes que tratam disso. Para saber a path de modo a termos o comando, teremos que usar o request, e o

método requestURI que devolve a path (em String). O tipo do pedido (classe Method) vem no método getMethod do request. Para executar o comando teremos que instanciar a classe App de modo a usar os métodos já existentes para obter o CommandHandler e assim, realizar o comando desejado. Aqui também, com base nos headers, é definido o formato da resposta. No caso de não haver handler, aparece uma mensagem a informar.

Encaminhamento dos comandos

A classe Router, primeiramente adiciona todas as Routes possíveis, adicionamos uma Route nova à lista de Routes e o findRoute que verifica se existe uma Route com o Path e o método passados.

Os parâmetros da Path são extraídos na Path, onde ao termos a path em string, usamos o split para separar por "/" e guardar num Hashmap, onde a key é a string e o value é o id, ou caso não haja id's ficará a null.

Foi adicionada uma nova Route na segunda fase (DELETE).

Foi adicionada novamente uma Route para o comando LISTEN.

Nas classes que gerem o servidor HTTP, a path é extraída do URI e armazenada de modo a poder ser chamada pelos métodos da App que inicializam os handlers para a execução dos comandos. Para casos de paginação, os parâmetros são passados pela queryString de modo a saber em que página nos encontramos.

Gestão de ligações

A conexão começa e acaba dentro do método execute do CommandHandler, uma vez que existe apenas um comando por conexão, é criada e encerrada na execução de cada comando.

Para a fase 3, existe a ligação ao porto desejado para o comando LISTEN (porto 8080), com o auxílio da classe Server.

Na fase final e a pedido dos docentes, foram implementados try-with-resources para a resolução das conexões de modo a garantir que, caso haja problemas com a conexão, sejam devidamente tratados e apresentados ao utilizador.

Acesso a dados

Onde possível, foram utilizados PreparedStatements de modo a proteger contra SQL-Injection.

Na segunda fase do trabalho foi criada a Classe Element e Text, que ajudam a processar os dados e a organizá-los de modo a ter uma estrutura idêntica a HTML. Conforme dito no guia de

desenvolvimento e apoio de HTML, foi seguido o modelo proposto e nestas classes residem os métodos que auxiliam a realização desse modelo. Foi escolhida a opção 2 (guia de HTML).

A classe CommandResult agora apresenta mais métodos onde são impressos os resultados nos formatos pedidos, tendo desta forma acesso visual aos dados pretendidos.

A classe CommandResult apresenta agora (fase 3) métodos que representam resultados que estarão nas tabelas (no browser quando o servidor está ligado). Os métodos que tinham como função mostrar a visualização dos dados nos formatos, foram ligeiramente modificados, incluindo agora tipos de retorno de modo a usarmos o seu conteúdo e transformá-lo em bytes para poder ser apresentado em HTML no browser.

A classe Element, lida também com a criação do "modelo" a ser visualizado, ou seja, a estrutura do HTML é processada nesta classe, tal como a criação dos recursos do HTML. O processamento das âncoras que permitem o navegar fluido da aplicação também é processado aqui, respeitando as regras de ligações impostas no enunciado e garantindo que as conexões existem e estão operacionais.

Para a fase final, esta foi a etapa mais extensa do trabalho. Todos os comandos foram implementados com tratamento de erros e PreparedStatements de modo a evitar SQL-Injection.

Inicialmente, foram criados DataMappers, para auxiliar falta de abstração na implementação prévia. Os problemas que levaram à utilização destas estruturas foram o facto de em cada comando, era adicionado uma nova entidade de domíno com os valores do ResultSet a uma lista, e, desta forma, não nos era possível obter cada campo na entidade de domínio, tendo sido desenvolvido previamente código extenso e confuso na implementação dos formatos e da fase 3, uma vez que as entidades só estavam acessíveis por via do método toString. Com o novo DataMapper é possível obter os campos desejados de cada entidade.

Foram adicionadas vistas para cada comando. As vistas têm por objetivo realizar verificação de erros, mapear cada resultado do comando utilizado.

O CommandResult foi simplificado, e transformado numa interface implementada por cada CommandView. O CommandResult é utilizado em todo o projeto de modo a obtermos os resultados de cada comando na forma que é mais indicada ao pedido efetuado à aplicação. Passou a ser a nossa ferramenta mais importante na simplificação e ordenação do código efetuado e agora remodelado, até à fase 3.

Após a reorganização, surgiu uma nova package, visual, é responsável pela representação gráfica dos resultados ao utilizador.

Ambas as classes HTTP e Element apresentavam métodos bastante longos e repetitivos e com uma implementação muito próxima do hard-code, e também muito confusa. Estas foram as classes mais alteradas. Começando pela classe PrintHtmlCommands, esta tinha, por exemplo, criações de novas tabelas em métodos diferentes e, ao apresentar os resultados no browser, estes não contavam com a presença de headers, sendo difícil ao utilizador perceber o que significavam os resultados apresentados. Foram agora, implementados métodos genéricos para a criação de tabelas, assim como para a criação de headers, sendo agora explícito o tipo de dados que o utilizador vê. Para os comandos GetUserById e GetRouteById, anteriormente, o comando para a realização tinha sido alterado de forma minimalista, ou seja, um CommandResult continha uma lista com uma mistura de várias entidades de domínio, as

tabelas de apresentação encontravam-se juntas e sem informação relativa a headers. Um dos maiores problemas de tal abordagem, é o facto de ao realizar estes comandos fora do âmbito do LISTEN, seriam apresentados dados a mais que não eram relevantes ao comando, e mesmo na apresentação no formato (por exemplo, json ou texto), não havia informação sobre os campos, sendo apenas apresentados dados “crus”. Os comandos adicionado santes(GetSportsByUserId e GetSportsByRouteId) e não utilizado devido à implementação explícita em cima, foram agora utilizados, e as tabelas de apresentação de dados estão agora em ordem, ou seja, cada entidade está agrupada na sua tabela e devidamente apresentada ao utilizador.Estes novos comandos ajudam a relacionar informação em comum entre Routes e Sports ou entre Sports e Users com a ajuda intermédia de Activity.

Ainda nesta classe foi criado o método executeCommands, que utilizando o mesmo príncipio da App, executa um comando para os casos em que é necessário imprimir mais que uma tabela, o que contrasta com a implementação anterior, onde apenas existia um comando misturado com várias entidades. Este comando permite a obtenção de um novo CommandResult que vai ter as informações necessárias numa lista para criar uma nova tabela.

A classe Element, embora não tenham existido mudanças de lógica significativas, foi possível obter uma implementação mais geral, um método genérico que redirecionaria para o tipo de impressão de tabela da classe acima mencionada(printHtmlCommands). Esta generalização foi especialmente necessária, pois o código continha imensos métodos que apenas diferiam num conjunto muito pequeno de linhas.

A package model conta com a adição da classe Erro que vai ajudar no processamento de erros.

Processamento de erros

Para a deteção de erros, foi criada a classe ParameterCheck que vai verificar erros de inserção de valores (parâmetros da path ou parâmetros). No caso da existência de erros, é retornado um CommandResult, que, como recebe um Object de parâmetro, vai ter adicionado à sua lista (presente em CommandResult) uma String que indica a falha na transação (que nem chega a começar), e como existe um método print que imprime as informações contidas no CommandResult, em caso de insucesso na transação aparece uma mensagem que anuncia o insucesso. Caso a transação fosse bem-sucedida, eram mostradas na consola as informações da transação (por exemplo, um GET /users/1 mostraria as informações do User com o identificador 1).

Devido à utilização do mecanismo try-catch, os erros na introdução de comandos são automaticamente tratados na classe App.

Nesta fase final do projeto, foi finalmente abordado o processamento de erros. Começando pelo modelo, foi adicionada uma classe Erro que apresenta em String ao utilizador, a mensagem de erro, não sendo mensagem de sistema, mas sim uma mensagem personalizada que indica o problema de uma maneira “user-friendly”. Foi adicionada uma vista também para o erro. O processamento de erros foi especialmente utilizado nos comandos POST onde tende a existir maior volume de erros, devido à inserção dos parâmetros pelo utilizador. Foram verificados erros, tais como chaves duplicadas, parâmetros não inseridos nos formatos corretos. Caso existam erros, estes nas implementações dos comandos são adicionados a uma lista que, após a realização dos comandos , terá os seus conteúdos imprimidos e mostrados ao utilizador.

Avaliação crítica

Os objetivos estabelecidos foram cumpridos, mas podia haver melhorias especialmente na área da Path e PathTemplate, ou seja, melhorar o método de extração dos id's de modo a ser mais intuitivo e melhorar código na verificação do match. A validação de parâmetros poderia estar mais desenvolvida, adicionando exceções e mais verificações. Nos comandos, poderiam ter sido utilizadas macros para cada campo.

Os objetivos principais para a fase 2 foram cumpridos, mas o trabalho pode ser melhorado. Por exemplo, limpeza de código nos comandos, ou algoritmos mais bem estruturados. Poderia haver um processamento de erros mais desenvolvido.

Os objetivos principais na fase 3 foram cumpridos, mas existe muito espaço para melhoria. Principalmente no processamento de erros, terá que ser melhorada a interface de modo a mostrar ao utilizador algo mais esclarecedor e não com mensagens de erros pré-definidas na linguagem. Algumas correções sugeridas pelos docentes na altura da demonstração já foram realizadas, mas ainda sobram umas melhorias por fazer, como por exemplo, mostrar mensagens de sucesso nos comandos POST. Outra melhoria pode ser nas tabelas HTML, mostrando estas o que significa cada campo e não apenas a informação "crua" de cada tipo de dados.

Para a fase final, embora a implementação de pedidos POST não tenha sido implementada, cremos que os objetivos estipulados pelos docentes para a melhoria do projeto foram todos atingidos e, implementados corretamente.