Introdução

Este projeto consiste numa aplicação Java, representando um servidor baseado no protocolo HTTP[1], para consulta e navegação de um subconjunto da informação disponível na base de dados *Northwind*.

O servidor deve aceitar pedidos HTTP, com método GET, para os recursos identificados por um conjunto de URI[2] e, em caso de sucesso, o resultado desses pedidos é uma representação em HTML[3] com a informação do recurso.

Para a concretização desta aplicação foi usada a biblioteca JDBC, para a ligação à base de dados SQL Server *Northwind*, e por um conjunto de classes e respectivos métodos, denominado *webfast*, cujo objectivo é o de providenciar uma API[4] de representação da informação em HTML.

Âmbito

1. O servidor consiste numa aplicação Java, usando o *package* com.sun.net.httpserver, presende no *Java Runtime Environment.*

2. O servidor deve aceitar pedidos http, com método GET, para os recursos identificados pelos URI apresentados em seguida. Em caso de sucesso, o resultado dos pedidos HTTP é uma representação em HTML com a informação do recurso, abaixo definida.

    - /employees – lista com todos os *employees.*
    - /employees/{eid} – propriedades do *employee* com chave **eid**.
    - /employees/{eid}/managees – lista com os *subordinados* do *employee* com chave **eid**.
    - /employees/{eid}/orders – lista com as ordens do *employee* com chave **eid***.*
    - /orders – lista com todas as ordens.
    - /orders/{oid} – propriedades da ordem com chave **eid**.
    - /orders/{oid}/details – lista com detalhes da ordem com chave **oid**.
    - /orders/{oid}/details/{pid} – propriedades do *order detail* com chave (**oid**, **pid**).

[1] *Hypertext Transfer Protocol*
[2] *Uniform Resource Identifier*
[3] *HyperText Markup Language*
[4] *Application Programming Interface*

- /customers – lista com todos os clientes.
- /customers/{cid} – propriedades do cliente com chave **cid**.
- /customers/{cid}/orders – lista de ordens do cliente com chave **cid**.
- /products – lista com todos os produtos.
- /products/{pid} – propriedades do produto com chave **pid**.
- /products/{pid}/orders – lista de *order details* para o produto com chave **pid**.

3. O resultado de cada pedido, em caso de sucesso, é uma representação HTML do recurso. Estas representações têm os seguintes requisitos:

    - Quando a representação HTML contém uma lista de itens, cada item deve incluir uma ligação para o URI com a informação detalhada desse item.
    - As representações devem também conter ligações para outros URI, de forma a garantir a navegabilidade entre todos os recursos, de acordo com o grafo apresentado na figura 1.

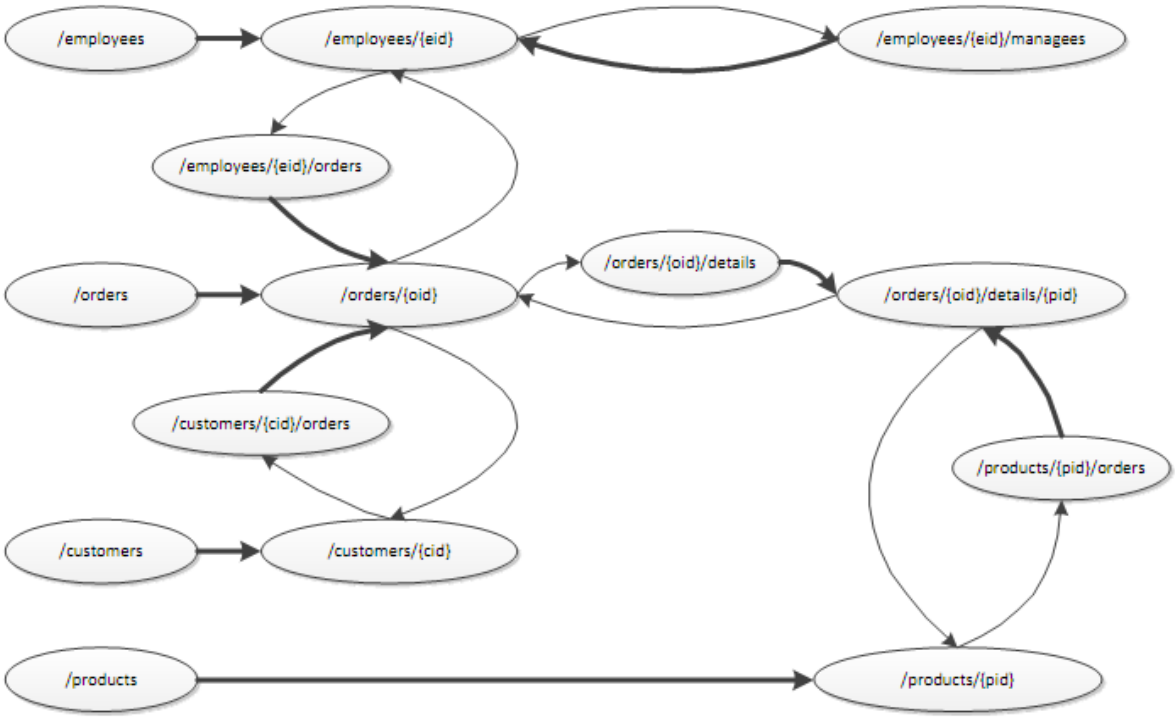

Figura 1: Grafo com as ligações entre recursos. Os arcos com maior espessura representam um conjunto de ligações.

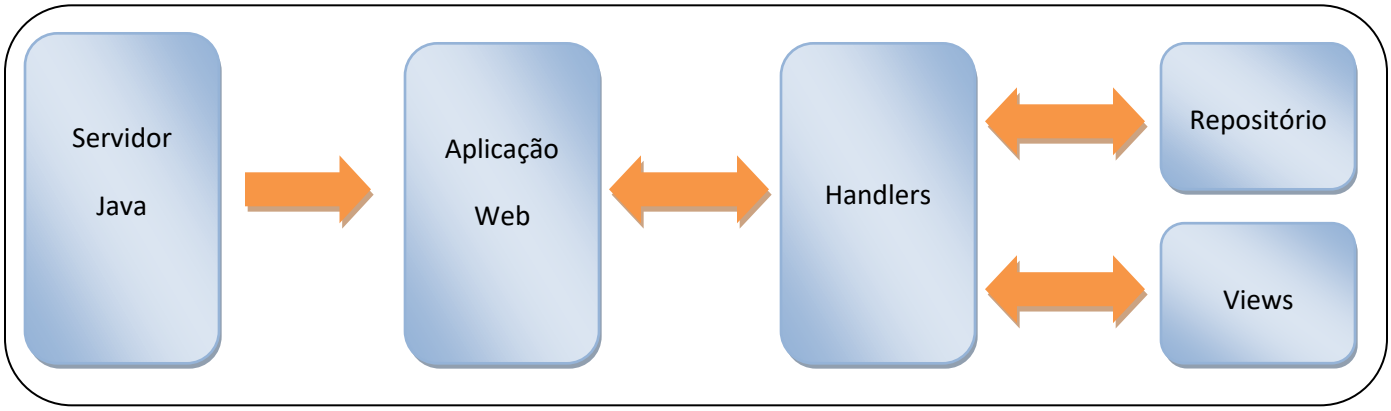

Figura 2: Arquitectura da solução implementada.

Como é possível observar na figura 2, a aplicação é constituída por cinco blocos principais de código, os quais são explicados se seguida.

i. Servidor Java – Bloco responsável pelo arranque do servidor e registo de aplicações.

ii. Aplicação Web – Bloco responsável pelo atendimento do pedido http, escolhendo qual o *handler* que trata o pedido recebido, recebendo a resposta do mesmo, devolvendo ao *browser*.

iii. Handlers – Bloco onde é processado o pedido recebido, obtendo os dados solicitados no pedido, através de um repositório, disponibilizado os mesmos para uma vista.

iv. Mappers – Bloco responsável pela gestão dos dados, disponibilizado mecanismos para consulta e edição de dados.

v. Views – Bloco responsável pela construção da resposta numa representação HTML, com base nos dados recebidos.

Servidor Java

Os passos de arranque do servidor são: registo de um porto para um *listening socket*, registar a aplicação *Web* que o servidor disponibiliza e iniciar o servidor.

Para receber pedidos HTTP, é utilizada uma instância de HttpServer, onde é indicado o endereço IP[5] e porto onde o servidor irá receber as ligações TCP e encaminhar os pedidos das mesmas para o(s) *handler(s)* registados na instância de HttpServer.

No caso concreto deste servidor, apenas foi registada uma aplicação, ou seja, um *handler*, que tratará todos os pedidos efectuados.

[5] *Internet Protocol*

Aplicação Web

A aplicação *Web* registada no servidor Java funciona como um *handler*. Esta aplicação consiste numa classe – MainAppWeb - que implementa a interface HttpHandler, que o obriga à definição do método handle, tendo como argumento uma variável do tipo HttpExchange.

É nesta variável passada ao método 'handle' que se encontram encapsulados o pedido efectuado pelo cliente, via *browser*, e a resposta a esse pedido que será posteriormente gerada pela aplicação, de forma a mostrar os conteúdos solicitados pelo cliente no *browser*. A classe HttpExchange disponibiliza também um conjunto de métodos para obtenção de informação do pedido do cliente, assim como para a construção da resposta.

O objectivo desta classe é o de escolher qual o *handler* que irá ser responsável pelo tratamento do pedido HTTP, sendo a decisão tomada com base no URI e método – GET ou POST.

Neste bloco foi utilizado o padrão *Strategy,* pois conforme o URI e método, será criada uma instância de uma classe que extende a classe abstracta AbstractRequestHandler, que sabe compor especificamente a informação solicitada pelo cliente, para cada um dos recursos identificados na figura 1.

Ao ser obtida a instância da classe, será então invocado o método 'write' com a finalidade de construção de uma resposta e obtenção do código de resposta HTTP a ser devolvido ao *browser*.

```
ByteArrayOutputStream mem = new ByteArrayOutputStream();
PrintStream printer = new PrintStream(mem);

// 500 - Internal Server Error
int httpCode = 500;

try {
      httpCode = rh.write(exch.getRequestBody(), printer, exch.getResponseHeaders(),
reqPath);
} catch (SQLException e) {
      error(exch, 500, e.getMessage());
}

printer.flush();
printer.close();

// first sends the header response
exch.sendResponseHeaders(httpCode, 0);
// Sends body response
mem.writeTo(exch.getResponseBody());
mem.flush();
mem.close();
exch.close();
```

Figura 3 - Extracto de código da obtenção da resposta HTTP.

O que se encontra representado na figura 3 é a invocação do método 'write' da instância da classe que implementa a interface IHttpHandler, dada pela variável 'rh', obtendo na variável 'httpCode' o *status code* http a ser entregue ao cliente, juntamente com toda a informação que foi obtida e processada numa representação HTML, que se encontra na variável 'mem'.

No caso de o URI se encontrar incorrectamente definido, dá-se a situação onde não é possível obter qualquer *handler,* sendo necessário retornar essa informação para o *browser*. Para tal, existe um mecanismo que trata estes casos - assim como os casos onde ocorre um erro interno no servidor -, que consiste no método 'error', cuja implementação se encontra na figura 4.

```
private void error(HttpExchange exch, int httpCode, String message) throws
IOException{
      // first sends the header response
      exch.sendResponseHeaders(httpCode, 0);
      // Sends body response
      PrintStream out = new PrintStream(exch.getResponseBody());
      out.print(message);
      out.flush();
      out.close();
      exch.close();
      return;
}
```

Figura 4 - Implementação do método error.

Um *handler* consiste numa classe que extende a classe abstracta AbstractRequestHandler, sendo responsável pela obtenção dos dados associados ao recurso definido pelo URI do pedido HTTP, entregando os mesmos a uma View que irá compor uma representação HTMLExiste um *handler* para cada recurso existente na aplicação, excepção feita aos recursos dados pelo URI "/employees/{eid}/orders" e "/customers/{cid}/orders", onde o handler que atende estes pedidos é o mesmo do URI "/orders", uma vez que a única diferença que existe consiste na informação listada.

Para obtenção da informação, os *handlers* utilizam um repositório específico para o tipo de dados que pretendem mostrar, através da classe RepositoryFactory, podendo efectuar interrogações[6] sobre o mesmo, com vista à obtenção dos recursos solicitados no pedido.

Essa informação obtida é passada a uma View específica, para o recurso em causa, através do construtor da mesma, para que seja construída a representação HTML. Finalmente é invocado o método 'print' da View para que seja colocado na resposta a representação obtida, conforme será detalhado posteriormente neste relatório.

Se não houver qualquer erro que leve ao lançamento de uma excepção, no fim é retornado o código 200, que será enviado na resposta ao *browser*, indicando que o pedido foi efectuado com sucesso.

[6] Termo em Inglês é *queries.*

```
public class EmployeesHandler extends AbstractRequestHandler{

	final IRepository<String, Employee> repo;

	public EmployeesHandler() throws SQLException{
		this.repo = RepositoryFactory.getRepository(Employee.class);
	}

	@Override
	public int write(InputStream requestBody, PrintStream responseBody,
			Headers responseHeaders, String requestPath) throws IOException,
			SQLException {

		new EmployeesView(repo.loadAll(),
requestPath.split("/")[2]).print(responseBody, 0, null);
		return 200;
	}

}
```

Figura 5 - Exemplo de implementação de Handler.

Na figura 5, onde se encontra representado um exemplo da implementação de um *handler*, neste caso, o *handler* que é responsável pelo pedido de listagem dos *employees* existentes, é possível verificar que o repositório de dados é obtido através da classe RepositoryFactory, passando a classe do recurso que está a ser tratado. Os dados são passados à View EmployeesView, invocando o método 'loadAll' do repositório, sendo posteriormente invocado o método 'print' da instância criada.

As classes que extendem a classe abstracta AbstractRequestHandler podem ser divididas em três subgrupos: *handlers* de listagem de todos os registos de uma entidade, *handlers* de listagem de um registo de uma entidade e *handlers* de listagem de detalhes sobre um registo de uma entidade, o que significa que existirá um *handler* de cada subgrupo para cada uma das cinco entidades – *customer, employees, order, orderdetails* e *product.*

Repositório

Neste bloco encontram-se especificadas as classes que permitem a obtenção da informação da base de dados *Northwind*, e sobre a qual serão efectuadas consultas. Estas classes, que implementam a interface IRepository, providenciam métodos para consulta dos recursos para cada uma das entidades existentes, conforme descrito de seguida.

```
public interface IRepository<K, T> {

	T loadById(K key) throws SQLException;

	Iterable<T> loadWhere(String whereClause) throws SQLException;

	Iterable<T> loadAll() throws SQLException;

	void put(T obj) throws SQLException;

}
```

Figura 6 - Interface IRepository

A figura 6 demonstra a interface IRepository, que as classes concretas de cada tipo de recurso implementam, sendo possível identificar os quatro métodos existentes para interacção com a base de dados:

i. **loadById**

- Objectivo: Obter um único registo de um tipo T, com base no seu identificador único de um tipo K.

- Input : Chave única, de um tipo genérico K, identificadora de um objecto do tipo genérico T, correspondente a uma entidade na base de dados.

- Output: Objecto do tipo T, identificado pela variável de *input*.

ii. **loadWhere**

- Objectivo: Obter um conjunto de registos de um tipo T, com base numa cláusula.

- Input: *String* com a cláusula que permite restringir o resultado da interrogação à base de dados. Como exemplo: Obter todos os *Customers* onde o campo *City* seja igual ao parâmetro de entrada.

- Output: Iterável de objectos de tipo T, resultante da interrogação efectuada.

iii. **loadAll**

- Objectivo: Obtenção de todos os registos de um tipo T.

- Output: Iterável de todos os objectos existentes do tipo T.

iv. put

- Objectivo: Inserção de um objecto de tipo T na base de dados.

- Input: O objecto de tipo T a inserir na base de dados.

As diversas classes que implementam a interface acima descrita encapsulam uma instância de IDataMapper, sendo através desta instância que é efectuada a comunicação com a base de dados para obtenção da informação, seguindo o padrão *Adapter,* conforme é visível na figura 7, que mostra a implementação da classe ProductRepository.

```
public class ProductRepository implements IRepository<Integer, Product>{

	private IDataMapper<Integer, Product> prodMapper;
	private final IDBContext ctx;

	public ProductRepository(String connectionUrl) throws SQLException{
		ctx = new DBContextSingleConnection(connectionUrl);
		prodMapper = new ProductDataMapper(ctx);
	}

	@Override
	public Iterable<Product> loadAll() throws SQLException {
		return prodMapper.load();
	}

	@Override
	public Product loadById(Integer key) throws SQLException {
		return prodMapper.loadById(key);
	}

	@Override
	public Iterable<Product> loadWhere(String whereClause) throws
SQLException {
		return prodMapper.where(whereClause);
	}

	@Override
	public void put(Product obj) throws SQLException {
		// TODO Auto-generated method stub
	}
}
```

Figura 7 - Implementação da classe ProductRepository

De salientar, que os *mappers* utilizados nos repositórios, são os que foram apresentados e estudados nas aulas ao longo do semestre e que foram desenvolvidos na segunda série de exercícios desta unidade curricular.

Estes *mappers* têm a particularidade de implementarem o padrão *Lazy Load*, ou seja, o resultado da execução de uma interrogação à base de dados é obtido apenas no momento em que se está a iterar sobre este.

RepositoryFactory

Sempre que o servidor recebe um novo pedido, é criada uma nova instância de um determinado *handler*, que necessita de um repositório para obter informação na base de dados. Para evitar a constante criação de uma instância de uma classe repositório, que implica um conjunto de operações que, em conjunto, se tornam dispendiosas, foi definida a classe RepositoryFactory. Esta classe contém uma variável privada estática do tipo HashMap que tem como chaves objectos do tipo Class e aceita como valores objectos IRepository.

Para que um *handler* obtenha o repositório para o tipo de recurso de que é responsável, esta classe disponibiliza o método estático 'getRepository' que tem como parâmetro um objecto do tipo Class, retornando o objecto IRepository que se encontra associado à chave para o objecto passado como argumento.

Desta forma, os repositórios apenas são instanciados uma vez, aquando a primeira chamada ao método 'getRepository'.

Views

Neste bloco de *software* encontram-se definidas as diferentes Views que efectuam a representação HTML do recurso indicado pelo URI do pedido HTTP, existindo, à semelhança dos *handlers*, uma View para cada recurso possível de ser requisitado.

Uma View consiste numa classe que extende a classe HtmlView do *package* webfast, e que através de objectos desse mesmo *package* - tais como HtmlA, HtmlTable e HtmlTr, correspondendo a representações Java dos elementos HTML âncora[7], tabela[8] e linha de tabela[9], respectivamente – compõe a representação HTML de um determinado recurso.

[7] Termo em Inglês é *anchor.*
[8] Termo em Inglês é *table.*
[9] Termo em Inglês é *table row.*

Assim, verifica-se que a implementação de uma View é efectuada com base no padrão *Composite*, uma vez que qualquer um dos objectos utilizados extendem a classe abstracta HandlerOutputComposite que implementa a interface HandlerOutput.

```
public class ProductsView extends HtmlView{

	public ProductsView(Iterable<Product> products, String controller){
		head().title("Products");

		HtmlBody body = body()
		.heading(1, "List of Products")
		.hr();

		HtmlTable table = body.table();

		HtmlTr tr = table.tr();
		tr.th().text("Id");
		tr.th().text("Product Name");

		for(Product p : products){
			tr = table.tr();
			tr.td().text("" + p.getProductID());
			tr.td().a(controller + "/" +
p.getProductID()).text(p.getProductName());
		}
	}
}
```

Figura 8 - Implementação da classe ProductsView

Esta representação será passada como resposta ao *browser*, através do objecto do tipo PrintStream que o *handler* irá passar como argumento ao método 'print' da classe abstracta HandlerOutputComposite, como anteriormente mostrado na figura 5.