Sistemas Distribuídos e Paralelos – Projecto Final

Identificação do Grupo de Trabalho

30002248 - Luís Carlos Silvério

30002299 - Ricardo Cardoso

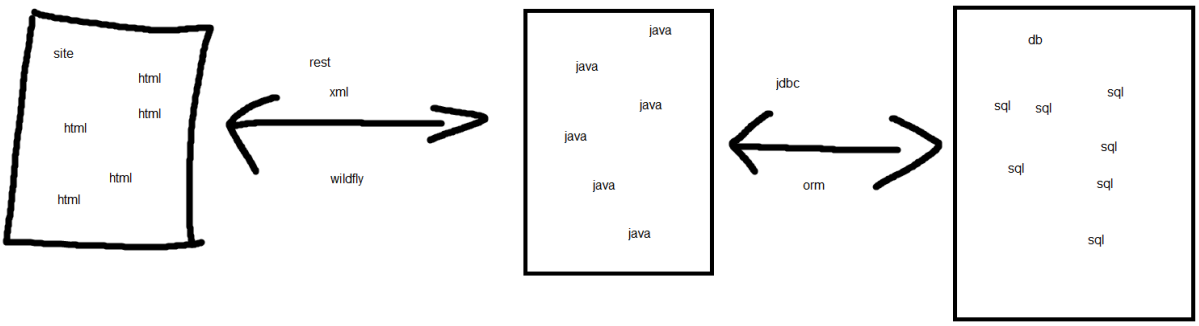
30002953 - Maria Inês Parreira

30002715 - Gonçalo Caldeirinha

Decisões Arquitecturais Tomadas

O projecto no início foi relativamente simples de visualizar, devido ao projecto de GSR ser vagamente semelhante (implementar um servidor com uma Webapp e Tomcat).

Portanto logo nas primeiras discussões entre o grupo foi montado um seguinte mock-up no MS Paint:

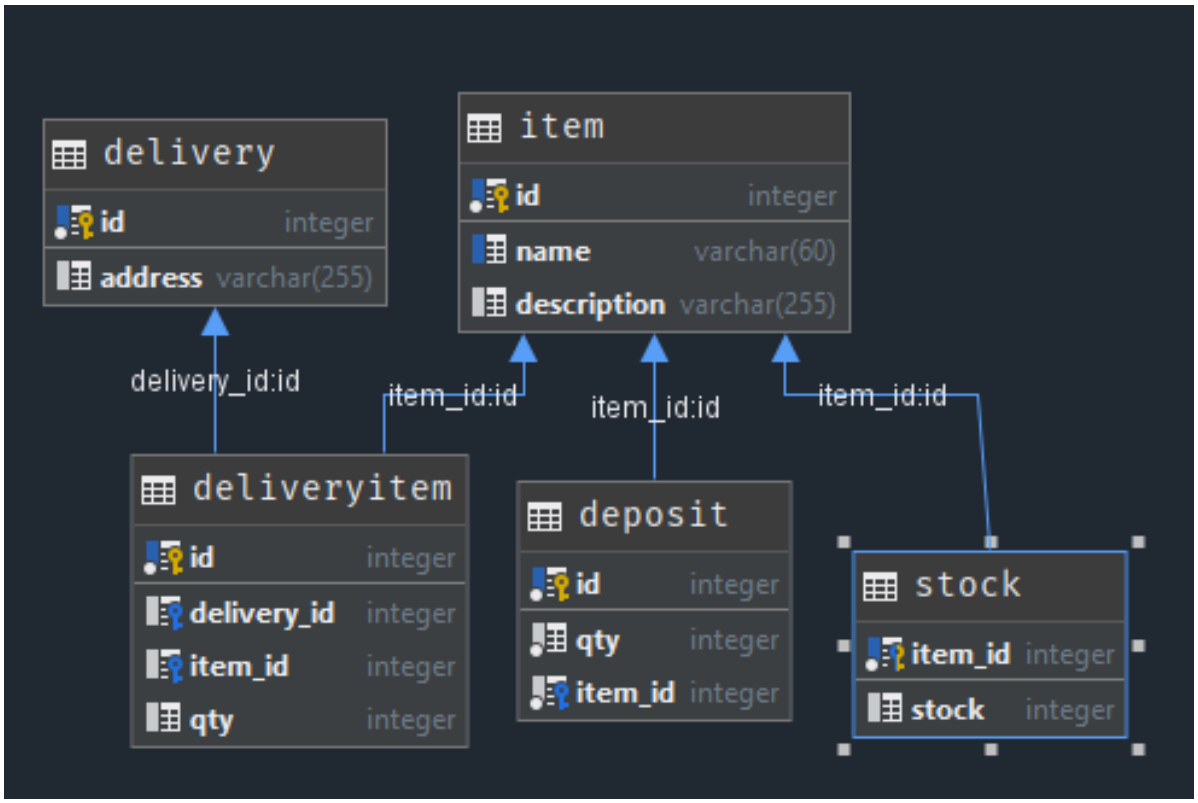


Depois decidimos usar todas as tecnologias que tinham apoio assegurado (Wildfly e PostgreSQL), isto porque nenhum dos membros do grupo tinha experiência prévia com Java a este nível.

A Rest API comunica com strings json tanto em entrada como em saída apenas.

Database

A base de dados foi implementada facilmente, apenas com o seguinte drawback de termos de fazer deliveries que associassem vários items a cada uma, para isso fizemos uma tabela relacional que liga items a deliveries.



Existem algumas constraints a ter em conta, na tabela dos items o nome é UNIQUE, devido a não podermos registar mais do que um item com o mesmo nome.

Nginx e Webserver

Decidimos usar nginx para o nosso servidor web, ao início consideramos apache, mas nginx foi a nossa última escolha devido a ser mais recente.

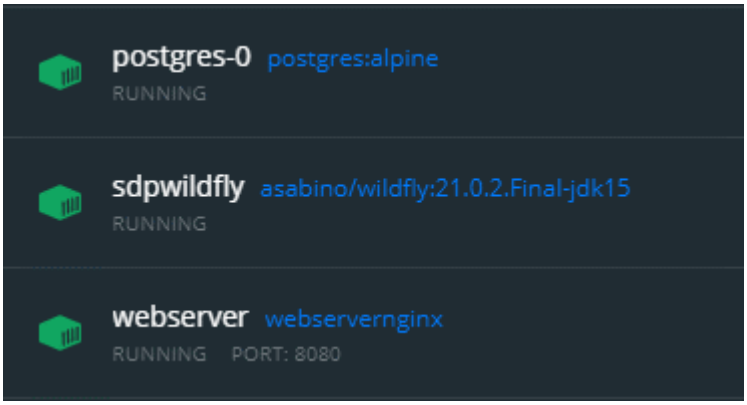
Quando implementámos o webserver, tinhamos problemas de network error, apesar de dentro do bash conseguirmos dar ping ao container do wildfly. Isto porque o nginx não estava ainda configurado. Depois configurámos um reverse proxy no nginx.conf.

Basicamente, os requests vão para o nginx e a unica porta aberta é a do nginx, o que nginx depois faz é ver o request e diferencia se é um request da API ou não. Se o request for default manda o static file normal (html), ou seja se o request não comecar com /api.

No caso da API, quando é feito um request a partir do webserver para o middleware, é reescrito a directoria para remover o /api.

Docker

Existem 3 containers do Docker neste projecto a correr numa network que chamámos de net nos nossos testes, mas o nome é arbitrário.



A comunicação entre eles funciona dentro da network, e apenas o webserver tem comunicação exterior na porta 8080.

Os comandos que usámos foram os seguintes:

```
docker build . -t postgres:alpine
docker run --network net --name postgres-0 postgres:alpine
docker build . -t asabino/wildfly:21.0.2.Final-jdk15
docker run --name sdpwildfly --network net asabino/wildfly:21.0.2.Final-jdk15
docker build . -t webservernginx
docker run --name webserver --network net -d -p 8080:80 webservernginx
```

Sendo o . no docker build a directoria onde estão os dockerfiles.

Listagem de Operações Suportadas pela API REST

Todas as operações funcionam pelo curl, usando http://localhost:8080/api/, devido ao reverse proxy.

Items

`http://localhost:8080/api/Items`

API para a consulta, registo, alteração e remoção de items e das suas descrições. Toda implementada na classe ItemRESTApi e na tabela item do SQL.

- GET

- Função doGet na classe Java.
- Basta aceder à static page, o conteúdo irá aparecer na tabela.
- Consulta os items da base de dados.

- POST

- Classe doPost na classe Java.
- Acessível através de um post exemplo com um formato como:
- `{"name": "copper ore", "description": "1g"}`
- Onde name e description são as chaves.

- PUT

- Classe doPut na classe Java.
- Acessível através de um put exemplo com um formato como:

- `{"name": "copper ore", "description": "2g"}`
- Onde name e description são as chaves.

- DELETE

- Classe doDelete na classe Java.
- Acessível através de um delete exemplo formatado como:
- `{"name": "copper ore"}`
- Onde name é a chave para o item que queremos apagar.
- O model apaga o registo do item **apenas** se o stock for 0.

Stock

`http://localhost:8080/api/Stock`

API para o registo dos depósitos e para consulta dos mesmos. Mostra apenas os items que estão em stock (ou seja quantidade superior a zero). Toda implementada na classe StockRESTApi e na tabela stock do SQL.

- GET

- Função doGet na classe Java.
- Basta aceder à static page, o conteúdo irá aparecer na tabela.
- Consulta o stock da base de dados.

- POST

- Classe doPost na classe Java.
- Acessível através de um post exemplo com um formato como:
- `{"name": "copper ore", "qty" : 30}`
- Onde name e qty são as chaves para o nome do item e quantidade.

Deliveries

`http://localhost:8080/api/Delivery`

API para o registo das entregas e para consulta das mesmas. Mostra as entregas num formato de id – morada – item:quantidade. O id é necessário ser mostrado devido à funcionalidade de mudança de morada das entregas. Cada entrega pode conter mais do que um item, e eles são mostrados pelo formato item-1 : quantidade, item-2: quantidade no webserver.

- GET

- Função doGet na classe Java.
- Basta aceder à static page, o conteúdo irá aparecer na tabela.
- Consulta as deliveries da base de dados.

- POST

- Classe doPost na classe Java.
- Acessível através de um post exemplo com um formato como:
- `{"address": "undercity", "items": [{"name" : "peacebloom", "qty": 10}, {"name" : "copper ore", "qty": 30}]}`
- Onde address é a chave para a morada, e items é uma array com chaves interiores de nome e quantidade.

- Um item não pode ser entregue se ele não for previamente registado ou se não tiver em stock.

- PUT

- Classe doPut na classe Java.
- Acessível através de um post exemplo com um formato como:
- `{"id": 33, "address": "dalaran"}`
- Onde id é a chave para a entrega, e address é a chave para modificar a morada.